

Submitted by
Dipl.-Ing. Andreas
Schörghumer, BSc

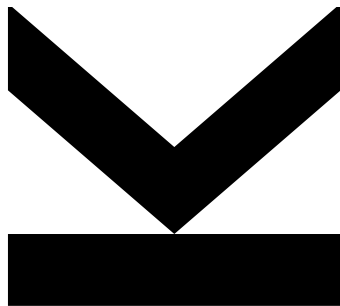
Submitted at
Institute for
System Software

Supervisor and
First Evaluator
o.Univ.-Prof.
Dipl.-Ing. Dr. Dr.h.c.
Hanspeter Mössenböck

Second Evaluator
Univ.-Prof.
Dipl.-Ing. Dr.
Martin Pinzger

July 2021

Data Analysis and Error Analytics in Large-Scale Heterogeneous Software Systems



Doctoral Thesis
to obtain the academic degree of
Doktor der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

Abstract

Today's software systems continuously collect monitoring data during operation, which can be used for assessment of system states, error analytics and overall data analysis. Due to the increasingly large amounts of data, manual analyses are infeasible and automated tools must be developed. While many approaches have been proposed in both research and industry that work in the context of a single system, the area of analyzing data from a multi-system environment has not yet been strongly focused, despite all its potential benefits such as combining multi-system data to create powerful models or identifying errors and patterns across multiple systems. In this thesis, we thus present our work on analyzing multi-system monitoring data, which consists of three separate approaches that each tackle specific tasks.

The first part comprises a crash analysis of the processes of the different multi-system topologies. Based on the processes' software technologies, we investigate the crash behavior with the goal to identify error-prone technologies and failures across multiple systems. In the evaluation, where we use over one year's worth of monitoring data of over 500 software systems from our industry partner, we show the feasibility and usefulness of our approach.

The second part of this thesis covers a multi-system event prediction with the main objective of predicting performance-related service slowdown events based on infrastructure monitoring time series. Using our sophisticated preprocessing framework, we extract different datasets and train various machine learning models, including several multi-system models that utilize data from different systems. We evaluate our approach on monitoring data covering 20 days of 57 software systems from our industry partner, which reveals a subpar prediction performance. In a detailed discussion and an additional evaluation of synthetic data, we identify possible reasons and limitations of our approach.

In the third part, we present a feature-based time series clustering approach. We create a set of clustering methods and automatically compare them using labeled data, where we can then choose one of the top-performing methods for clustering unlabeled data to extract common patterns across different software systems. We also propose a run-time cost model to assess the computational costs in addition to the clustering quality. The evaluation comprises the UCR time series archive as well as two infrastructure monitoring datasets from our industry partner, covering thousands of time series of hundreds of systems. The results reveal interesting insights and demonstrate the usefulness of our approach.

Kurzfassung

Moderne Softwaresysteme sammeln während des Betriebs kontinuierlich Monitoring-Daten, die zur Beurteilung von Systemzuständen, zur Fehleranalytik und zur Gesamtdatenanalyse genutzt werden können. Aufgrund der immer größer werdenden Datenmengen sind manuelle Analysen nicht mehr durchführbar, weshalb automatisierte Werkzeuge entwickelt werden müssen. Während sowohl in der Forschung als auch in der Industrie viele Ansätze vorgeschlagen wurden, die im Kontext eines Einzelsystems funktionieren, wurde bisher noch kein ausreichender Fokus auf eine Multisystemumgebung gesetzt, trotz aller potenziellen Vorteile wie der Kombination von Multisystemdaten zur Erstellung leistungsfähiger Modelle oder der Identifizierung von Fehlern und Mustern über mehrere Systeme hinweg. In dieser Arbeit stellen wir daher unsere Arbeit zur Analyse von Multisystem-Monitoring-Daten vor, die aus drei separaten Ansätzen besteht, welche jeweils spezifische Aufgabenstellungen behandeln.

Der erste Teil umfasst eine Crash-Analyse von Prozessen der verschiedenen Multisystemtopologien. Basierend auf den Softwaretechnologien dieser Prozesse untersuchen wir das Absturzverhalten mit dem Ziel, fehleranfällige Technologien und Ausfälle über mehrere Systeme hinweg zu identifizieren. Die Auswertung von mehr als einem Jahr an Monitoring-Daten von über 500 Softwaresystemen unseres Industriepartners zeigt die Praxistauglichkeit und Nützlichkeit unseres Ansatzes.

Der zweite Teil dieser Arbeit befasst sich mit der Vorhersage von leistungsbezogenen Service-Slowdown-Ereignissen auf Basis von Infrastruktur-Monitoring-Zeitreihen. Mit Hilfe unseres leistungsfähigen Preprocessing-Frameworks extrahieren wir verschiedene Datensätze und trainieren diverse Machine-Learning-Modelle, darunter mehrere Multisystemmodelle, die Daten von unterschiedlichen Systemen nutzen. Wir evaluieren unseren Ansatz anhand von Monitoring-Daten, die 20 Tage von 57 Softwaresystemen unseres Industriepartners umfassen, wobei wir eine unterdurchschnittliche Vorhersageleistung feststellen müssen. Im Rahmen einer ausführlichen Diskussion und einer zusätzlichen Auswertung von synthetischen Daten identifizieren wir mögliche Gründe sowie Grenzen unseres Ansatzes.

Im dritten Teil stellen wir einen merkmalsbasierten Zeitreihen-Clustering-Ansatz vor. Wir erstellen Clustering-Methoden und vergleichen diese automatisch mit Hilfe gelabelter Daten, aus denen wir eine der besten Methoden für das Clustering von nicht gelabelten Daten wählen können, um gemeinsame Muster über verschiedene Softwaresysteme hinweg zu extrahieren. Zusätzlich präsentieren wir ein Laufzeitkostenmodell, um neben der Clustering-Qualität auch die Rechenkosten zu bewerten. Die Evaluierung umfasst das UCR-Zeitreihenarchiv sowie zwei Infrastruktur-Monitoring-Datensätze unseres Industriepartners, die tausende Zeitreihen von hunderten Systemen beinhalten. Die Ergebnisse zeigen interessante Einblicke und demonstrieren die Nützlichkeit unseres Ansatzes.

Acknowledgments

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

I would like to thank my supervisor Prof. Hanspeter Mössenböck for offering me the chance to pursue a PhD and to work at the Christian Doppler Laboratory on Monitoring and Evolution of Very-Large-Scale Software Systems. Thank you for trusting and encouraging me throughout all the years I worked there. My gratitude also goes to Prof. Martin Pinzger, who readily agreed to be my second examiner.

I would also like to thank Herwig Moser, Hans Kohlreiter, Wolfgang Beer and Thomas Natschläger from our industry partner Dynatrace for all the meetings and their valuable input.

My utmost gratitude goes to all colleagues who I worked with, especially to my former team members Peter Chalupar and Mario Kahlhofer, without whom I would most likely not have had the perseverance to finish my PhD. Thank you for your constant support, your great work and all the funny and memorable moments.

Special thanks go to my fellow PhD colleague Markus Weninger, with whom I had countless great and fruitful discussions, and who always listened to me in case I had to complain about something. Thank you also for the constant supply of quality memes that always kept me smiling when things did not go as planned.

Many thanks to Prof. Paul Grünbacher, who constantly supported and encouraged me throughout my entire PhD. Thank you for your readiness to help and for your everlasting buoyant disposition.

Lastly, I would like to thank all my friends and especially my family. With their unconditional help and support, they enabled me to pursue my wish to study computer science and ultimately to finish my PhD, for which I am eternally grateful. Thank you ever so much!

Contents

1 Introduction	1
1.1 Motivation	1
1.2 Scientific Contributions	2
1.3 Outline	3
2 Background	5
2.1 Dynatrace	5
2.2 Data Formats	5
2.2.1 JSON	5
2.2.2 YAML	6
2.2.3 CSV	7
2.2.4 InfluxDB	7
2.3 Multi-System Infrastructure Monitoring Data	7
2.3.1 System	8
2.3.2 Topology	8
2.3.3 Events	10
2.3.4 Time Series	11
2.3.5 Data Collection and Storage	11
2.4 Machine Learning	13
2.4.1 Basics	13
2.4.2 Data Imbalance	15
2.4.3 Supervised Learning	16
2.4.3.1 Training and Testing	16
2.4.3.2 Variants of Training and Testing	17
2.4.3.3 Evaluation Metrics	17
2.4.3.4 Random Forests	20
2.4.4 Unsupervised Learning	21
2.4.4.1 Evaluation Metrics	23
2.4.4.2 t-distributed Stochastic Neighbor Embedding (t-SNE)	26
2.4.4.3 k-Means	27
2.4.4.4 Hierarchical Clustering	28

2.4.4.5	Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH)	32
2.5	Statistical Background	33
2.5.1	Standardization	33
2.5.2	Normalization	33
2.5.3	Pearson Correlation	34
2.5.4	Wilcoxon Signed-Rank Test	34
2.5.5	Box Plots	35
3	Topology-driven Crash Analysis	37
3.1	Motivation	37
3.2	Data Requirements and Assumptions	38
3.2.1	Topology	38
3.2.2	Events	38
3.3	Approach	39
3.3.1	Tuple Creation	40
3.3.2	Tuple Merging	41
3.3.3	Ranking	42
3.3.4	Crash Property Analysis	43
3.4	Data for Evaluation	44
3.5	Evaluation	49
3.5.1	Automated Analysis	49
3.5.2	Manual Investigation	51
3.5.3	Process Communication	52
3.6	Discussion	56
3.6.1	Lessons Learned	56
3.6.2	Problems and Limitations	58
3.6.3	Threats to Validity	58
3.7	Related Work	60
3.8	Outlook	62
4	Time-Series-based Event Prediction	63
4.1	Motivation	63
4.2	Data Requirements and Assumptions	64
4.2.1	Topology	64
4.2.2	Events	65
4.2.3	Time Series	65
4.3	Data Preprocessing Framework	66
4.3.1	Requirements	68
4.3.2	Preprocessing Pipeline	68
4.3.2.1	Data Access	69

4.3.2.2	Data Selection	69
4.3.2.3	Sampling	71
4.3.2.4	Data Extraction	73
4.3.3	Scalability	80
4.4	Approach	81
4.4.1	Data Preparation	81
4.4.2	Event Prediction	82
4.5	Data for Evaluation	82
4.6	Evaluation	85
4.6.1	Clustering	93
4.6.2	Balanced Scenario	94
4.6.3	Unbalanced Scenario	103
4.6.4	Synthetic Data	110
4.6.5	Balanced Scenario Revisited	120
4.7	Discussion	123
4.7.1	Lessons Learned	123
4.7.2	Problems and Limitations	124
4.7.3	Threats to Validity	125
4.8	Related Work	126
4.8.1	Time Series Processing	126
4.8.2	Time-Series-based Frameworks	127
4.8.3	More General Frameworks	127
4.8.4	Log-Data-based Approaches	128
4.8.5	Monitoring-Data-based Approaches	129
4.8.6	Reliability Prediction	130
4.9	Outlook	131
5	Time Series Clustering	133
5.1	Motivation	133
5.2	Data Requirements and Assumptions	134
5.3	Time Series Characteristics	135
5.4	Approach	139
5.4.1	Determining Feature Set Importance	143
5.4.2	Post-Processing Feature Sets	143
5.4.3	Clustering Labeled Data	146
5.4.4	Clustering Unlabeled Data	154
5.4.5	Run-Time Cost Model	154
5.5	Data for Evaluation	156
5.5.1	UCR Archive	156
5.5.2	IMTS Archive	157
5.5.3	UCR and IMTS Datasets	158

5.6 Evaluation	163
5.6.1 Time Series Characteristics	163
5.6.2 Clustering Method Selection	165
5.6.2.1 Determining Feature Set Importance	166
5.6.2.2 Post-Processing Feature Sets	166
5.6.2.3 Clustering Labeled Data	168
5.6.2.4 Clustering Unlabeled Data	173
5.6.2.5 Run-Time Cost Model	179
5.7 Discussion	183
5.7.1 Lessons Learned	186
5.7.2 Problems and Limitations	187
5.7.3 Threats to Validity	188
5.8 Related Work	189
5.8.1 Features for Time Series	189
5.8.2 Automatic Clustering Selection	190
5.8.3 Analysis of Industrial Systems	190
5.8.3.1 Statistical Analysis	191
5.8.3.2 Applied Clustering	191
5.8.4 Run-time Costs	192
5.9 Outlook	193
6 Conclusion	195
A Background	197
A.1 Feature Importance	197
B Topology-driven Crash Analysis	199
B.1 Data Exploration	199
B.2 Evaluation Results	204
C Time-Series-based Event Prediction	221
C.1 Data Exploration	221
C.2 Evaluation Results	223
D Time Series Clustering	251
D.1 Permutation Analysis Feature	251
D.2 Data Exploration	252
D.3 Evaluation Results	255
D.3.1 Variant Differences	255
D.3.2 Clustering Unlabeled Data	255
Bibliography	301

Chapter 1

Introduction

1.1 Motivation

Monitoring data of software systems provide a rich source of information. Especially with the recent advances in hard- and software technologies, an abundance of data is recorded, which can no longer be analyzed manually. Instead, automated approaches, tools and frameworks are required for processing and analysis, and many researchers as well as practitioners have proposed valuable solutions throughout the years. However, analyzing data from *multiple, independent systems* has not yet been a strong focus of research, despite all the potential benefits.

For instance, we can use this kind of data to identify errors across multiple systems and possibly even fix them in the affected systems if they share a common root cause. In addition, we could collect all such error and failures in general to populate a multi-system fault database, which would greatly help debugging, fixing and quality assurance, especially when dealing with recurring incidents, regardless of the system in which they happened. This is not the only advantage of multi-system data. For example, assume that we want to train some machine learning model which requires certain amounts of data to build a reliable model. If we only have data of a single system but the amount is insufficient, we cannot proceed any further. On the other hand, if we have similar or comparable data from several systems, we can use this to our advantage to merge this multi-system data, which then allows us to create our intended model (of course, the data might not be sufficiently similar, which would lead to an invalid/unusable multi-system model). Moreover, we could not only handle insufficient system data but also potentially apply our model on yet unseen, new systems, where no historic data is available in the first place. Another benefit that multi-system data can bring is the possibility to identify common patterns across different systems. If we know that some systems share certain characteristics, we could create models and tools specifically designed for these characteristics, which could then be used in all affected systems without having to develop such a model/tool for each single system separately.

In this thesis, we thus address the topic of analyzing data from a multi-system environment. Our goal is to implement approaches that can leverage such data and the corresponding benefits introduced above, where the results can either directly be used or provide a basis for further analyses. The next section covers an overview and the scientific contributions of these approaches.

1.2 Scientific Contributions

Throughout this project, we created and implemented three main approaches that are all focused on multi-system data. All approaches are described in detail in the following chapters, so we only provide a brief overview of their topics and list the corresponding scientific publications. The main focus and novelty of all our approaches compared to existing work is the integration of the multi-system environment, i.e., we specifically address multi-system problems, scenarios and challenges, which has not yet been done to the extent we propose in this thesis.

The first part describes our topology-driven multi-system process crash analysis, where we aimed to automatically find common crashes across multiple systems using the software technology information provided by processes. Publications:

- Andreas Schörgenhumer, Mario Kahlhofer, Hanspeter Mössenböck, and Paul Grünbacher. “Using Crash Frequency Analysis to Identify Error-Prone Software Technologies in Multi-System Monitoring”. In: *Proceedings of the 18th IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2018, pp. 183–190. DOI: [10.1109/QRS.2018.00032](https://doi.org/10.1109/QRS.2018.00032)

The second part contains our multi-system event prediction approach, where the goal was to leverage infrastructure monitoring time series to predict performance-related service slowdown events. Work on this topic also includes our comprehensive multi-system preprocessing framework. Publications:

- Andreas Schörgenhumer, Mario Kahlhofer, Peter Chalupar, Hanspeter Mössenböck, and Paul Grünbacher. “Using Multi-System Monitoring Time Series to Predict Performance Events”. In: *Proceedings of the 9th Symposium on Software Performance*. GI Softwaretechnik-Trends, 2018, pp. 55–57. URL: http://pi.informatik.uni-siegen.de/stt/39_3/01_Fachgruppenberichte/SSP18/SchoergenheimerKahlhoferChalupar+18.pdf
- Andreas Schörgenhumer, Mario Kahlhofer, Peter Chalupar, Paul Grünbacher, and Hanspeter Mössenböck. “A Framework for Preprocessing Multivariate, Topology-Aware Time Series and Event Data in a Multi-System Environment”. In: *Proceedings of the 19th IEEE International Symposium on High Assurance Systems Engineering*. IEEE, 2019, pp. 115–122. DOI: [10.1109/HASE.2019.00026](https://doi.org/10.1109/HASE.2019.00026)
- Andreas Schörgenhumer, Mario Kahlhofer, Paul Grünbacher, and Hanspeter Mössenböck. “Can We Predict Performance Events with Time Series Data from Monitoring Multiple Systems?” In: *Companion of the 10th ACM/SPEC International Conference on Performance Engineering*. ACM, 2019, pp. 9–12. DOI: [10.1145/3302541.3313101](https://doi.org/10.1145/3302541.3313101)
- Andreas Schörgenhumer, Mario Kahlhofer, Peter Chalupar, Hanspeter Mössenböck, and Paul Grünbacher. “On the Difficulties of Supervised Event Prediction based on Unbalanced Real-World Data in Multi-System Monitoring”. In: *Proceedings of the 10th Symposium on Software Performance*. GI Softwaretechnik-Trends, 2019, pp. 38–40. URL: http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Schoergenheimer.pdf

The third part details our feature-based time series clustering approach, where we wanted to identify common time series clusters within the monitoring data of multiple systems. Besides

an automatic ranking of clustering methods, we also provide a run-time cost model to assess computational costs in addition to the clustering quality. Publications:

- Andreas Schörgenhumer, Paul Grünbacher, and Hanspeter Mössenböck. “Selecting Time Series Clustering Methods based on Run-Time Costs”. In: *Proceedings of the 11th Symposium on Software Performance*. Accepted for publication. GI Softwaretechnik-Trends, 2020. URL: https://www.performance-symposium.org/fileadmin/user_upload/palladio-conference/2020/Papers/SSP2020_paper_1.pdf
- Andreas Schörgenhumer, Thomas Natschläger, Paul Grünbacher, Mario Kahlhofer, Peter Chalupar, and Hanspeter Mössenböck. “An Approach for Ranking Feature-based Clustering Methods and its Application in Multi-System Infrastructure Monitoring”. In: *AI-Enabled Software Development and Operations (AI4DevOps) of the 47th Euromicro Conference on Software Engineering and Advanced Applications*. Accepted for publication. IEEE, 2021

1.3 Outline

In this section, we provide an outline and the structure of this thesis. [Chapter 2](#) contains all the relevant and necessary background information for all following chapters. It primarily serves as a reference chapter, and readers who are already familiar with certain background topics can simply skip them. Afterwards, our three main topics follow: [Chapter 3](#) describes our topology-driven crash analysis, [Chapter 4](#) our time-series-based event prediction and [Chapter 5](#) our time series clustering approach. To guide the reader through the thesis, each of these three chapters is structured into the following sections:¹

- *Motivation*: This sections presents an introduction and motivates what we try to accomplish and implement in the corresponding chapter.
- *Data Requirements and Assumptions*: None of our approaches are limited to only one sort of dataset, which is why we describe all requirements and assumptions that must be fulfilled in order to apply the approach.
- *Approach*: In this section, we detail the approach itself and how it works given the data requirements and assumptions from the previous section.
- *Data for Evaluation*: Here, we present the datasets that we actually use to evaluate our approach. All datasets introduced in this section fulfill all of the requirements and assumptions, i.e., they can be seen as an “instantiation” of the theoretical data.
Evaluation: Using the data from the previous section, we evaluate our approach and present detailed results.
- *Discussion*: This section contains general points of discussion, lessons learned as well as problems and limitations we encountered, and threats to validity.
- *Related Work*: In this section, related literature is discussed, including differences to our approach and work that could prove valuable for future extensions and improvements.
- *Outlook*: The last section provides a brief overview of future work.

¹[Chapter 4](#) and [Chapter 5](#) have one additional section just before the approach (the preprocessing frame work and the time series characteristics, respectively) to achieve a clearer structure and better separation.

Finally, **Chapter 6** summarizes and concludes this thesis. There is also an appendix for the background chapter and for each of the three main topics, which contains further information regarding data exploration and supplementary results. The appendix is mainly intended to provide a more complete overview of the evaluated data and results for interested readers, but it is not necessary for the main content of this thesis.

Chapter 2

Background

This chapter covers everything required to understand the following chapters and also serves as a reference for the terminology used throughout this thesis. First, we provide a short introduction of our industry partner Dynatrace. We continue by describing the different data formats we used for our practical work. We then introduce one of the core parts of this thesis: the multi-system infrastructure monitoring data, including detailed information on systems, their topology, events and time series. Afterwards, a brief introduction to machine learning follows, with a strong focus on the methods, models and algorithms we used in our work. Finally, we also present some important statistical features.

2.1 Dynatrace

Our industry partner Dynatrace is a software company who provides application performance management, digital experience management, digital business analytics, artificial intelligence for operations and infrastructure monitoring [105]. Their customers range from small businesses to large-scale enterprises, covering a wide area of domains, which includes e-commerce, the automotive industry or enterprise resource planning. For our cooperation, infrastructure monitoring is the most important part, where the goals are to monitor and analyze the entire software system of a customer, thereby collecting data of all relevant system components. Dynatrace provides a plethora of monitoring data, however, we only need a few selected parts thereof, which we present in detail in [Section 2.3](#).

2.2 Data Formats

In this section, we cover all important data formats that we use throughout this thesis for storing different kinds of data.

2.2.1 JSON

JSON [40] (JavaScript Object Notation) is a plain-text data format with the main goal of object serialization and language-independent data exchange. Its two core building blocks are objects (key-value pairs within braces `{...}`) and arrays (ordered sequences of values within square brackets `[...]`), where the values can be `null` (no value), boolean, numbers, strings or again objects and arrays, allowing arbitrary nesting levels to represent complex object structures.

The example in [Listing 2.1](#) shows how a configuration object for extracting data of some sensor could look like in the JSON format. Within the root object, there are five key-value pairs: "data", "functions", "skipInvalid", "threshold" and "output". The last three have simple values attached (boolean, number, string), "functions" has an array with three strings as values, and "data" points to another, nested object which contains the two key-value pairs "ids" (an array of numbers) and "meta" (yet another object with a string "type" and a number "freq").

```
{
  "data": {
    "ids": [
      123,
      124,
      217
    ],
    "meta": {
      "type": "sensor",
      "freq": 240
    }
  },
  "functions": [
    "MIN",
    "MAX",
    "AVG"
  ],
  "skipInvalid": true,
  "threshold": 0.75,
  "output": "C:\\eval\\data"
}
```

Listing 2.1: JSON example of a configuration file to extract some sensor data.

2.2.2 YAML

YAML [\[14\]](#) (YAML Ain't Markup Language) is a plain-text data format primarily designed for readability by humans, programming language independence and ease of use. In contrast to JSON, YAML prioritizes easy human interaction over the simplicity of generating or parsing files. It also includes a native support for the three basic primitives, namely mappings (dictionaries/maps), sequences (arrays/vectors/lists) and scalar values, which are an integral part of the main YAML structure. The key-value pairs of mappings can either be written as a comma-separated list within braces `{ . . }`, or they can also be indented and listed below each other for improved readability. Analogously, the values of sequences can either be comma-separated and wrapped in square brackets `[. .]` or indented with a leading dash and listed below each other. While YAML files are not restricted to any specific scenario or domain, the authors highlight configuration or log files, communication between processes and persisting objects (e.g., for cross-language data exchange) as most common and fitting use cases. As of version 1.2, YAML is a superset of JSON.

[Listing 2.2](#) shows the YAML version of the JSON example of [Listing 2.1](#). Since the braces are replaced with simple indentations and quotation marks are no longer necessary for the identifiers, the YAML version is more readable and also much shorter. Note that the quotation marks for the string values are only required for the `output` identifier, since it contains escape characters. However, using quotes for all strings further enhances readability as it makes it more clear that the values are strings.


```

data:
  ids:
    - 123
    - 124
    - 217
  meta:
    type: "sensor"
    freq: 240
  functions:
    - "MIN"
    - "MAX"
    - "AVG"
skipInvalid: true
threshold: 0.75
output: "C:\\eval\\data"

```

Listing 2.2: YAML example of a configuration file to extract some sensor data.

2.2.3 CSV

CSV [163] (Comma-Separated Values) is a simple plain-text data format that stores records in each line of the file, separated by commas. Due to its simplicity, the CSV format is widely used in various domains and is a typical export format to store tabular data.

In Figure 2.1, a small example is shown how a typical use case could look like. Continuing the example of Listing 2.1, some sensor values were recorded and listed in a standard table as shown in Figure 2.1a. When exporting the data to the CSV format, the results could look as shown in Figure 2.1b.

id	type	MIN	MAX	AVG
123	sensor	12.1	99.9	24.5
124	sensor	12.2	57.2	14.9
217	sensor	33.7	45.0	38.3

(a) Ordinary data table.

```

id,type,MIN,MAX,AVG
123,sensor,12.1,99.9,24.5
124,sensor,12.2,57.2,14.9
217,sensor,33.7,45.0,38.3

```

(b) The same data represented as CSV.

Figure 2.1: Example of some extracted sensor data first represented in a table view (left) and then in the CSV format (right).

2.2.4 InfluxDB

InfluxDB [84] is a database with the main purpose of storing time series. It was specifically designed to cope with high writing and querying loads, while allowing high compression rates as well as easy data queries using an SQL (Structured Query Language)-like format. InfluxDB can handle millions of data points per second and access them efficiently by utilizing indexed time series clusters, supporting a precision up to nanoseconds but also the option to downsample older data, i.e., to merge data to a lower precision to mitigate storage shortage over time.

2.3 Multi-System Infrastructure Monitoring Data

In this section, we present the core data structure of this thesis. We start by giving an introduction of the systems and then continue with details on a system's topology, the

occurring events and the time series we collect. Finally, we describe how all the different data was recorded and how it is stored.

2.3.1 System

A *system* is any kind of software system. Its size can start from one simple computer and may range up to large clusters with thousands of servers. The domain can also be arbitrary. It could be a travel booking platform, an online shop or a maintenance system in the automotive sector. A system consists of various monitored hardware and software *components/entities* that are connected via the *topology* and may be associated with *events* and *time series*, the combination of which we refer to as *infrastructure monitoring data*. We describe each of these three main parts in the following sections. Note that we have these kinds of data from multiple, independent systems at our disposal, forming our *multi-system environment*, for which a general overview is shown in [Figure 2.2](#).

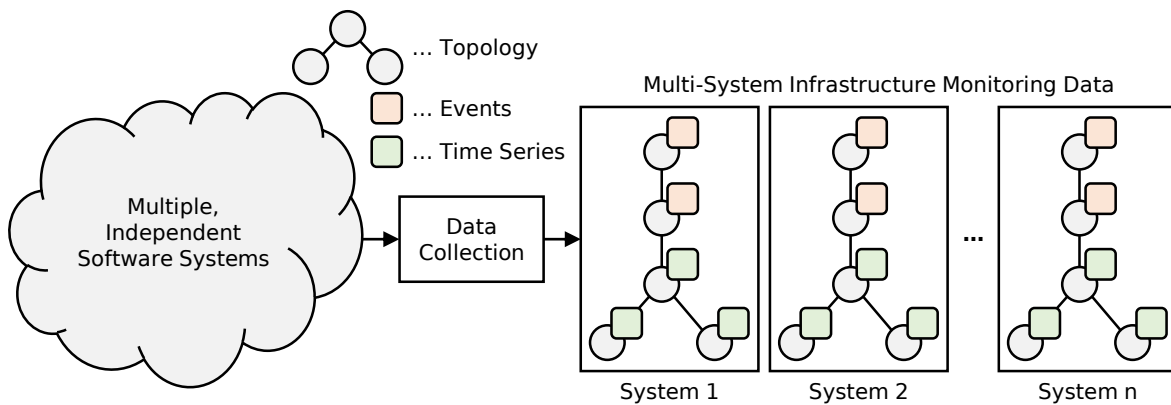


Figure 2.2: Overview of our multi-system environment, where infrastructure monitoring data (topologies, events, time series) of various, independent systems is collected.

2.3.2 Topology

The topology of a system stores its components, how they are linked and different kinds of metadata. Components are physical or abstract entities, and each one is uniquely identified (ID), has an associated type, a lifetime (two timestamps when the entity was first seen and when it was last seen) and, depending on the type, additional data. The most important feature is the type. It determines to which other types a component may be connected, and which time series and events can be recorded. Dynatrace monitors many different component types, however, in this thesis, the following subset suffices:

- *Service*: This type represents abstract components that represent the business logic of the system, e.g., a database access or a web request. Services are executed by process groups that consist of potentially multiple processes (e.g., for handling load balancing), i.e., a service entity is connected to one or more process entities.
- *Process*: This type represents process entities that execute services. The *software technologies* (ST) are an important data structure that is attached to every process. It manages a list of all technologies that the process used or uses, where each list entry contains the following information: the technology's type (e.g., Java), the edition (e.g., OpenJDK), the version (optional, e.g., 11.0.1) and the timestamp when it was last seen

on the process, i.e., the point in time until the technology was considered active. An example of such software technologies is shown in [Table 2.1](#). Here, a process was running a Tomcat server on Java until both technologies were replaced with newer versions. Processes can communicate with each other, and they run on a host, i.e., a process entity is connected to zero or more process entities and to one host entity.

Type	Edition	Version	Active Until
Java	OpenJDK	1.8.0_121	1495535781954
Tomcat	-	7.0.65.0	1495535781954
Java	OpenJDK	1.8.0_131	1499218873887
Tomcat	-	8.0.44.0	1499218873887

Table 2.1: Example software technology properties of a process, taken from [\[157\]](#). The - character represents missing (optional) fields. The timestamps are in milliseconds of UTC (Coordinated Universal Time).

- *Host*: Entities of this type represent the processing units of the system, which can be the actual computers or servers, or virtual machines. They run the processes (and indirectly the services) and are one of the three main components for collecting infrastructure monitoring time series. Hosts can have multiple disks and network interfaces, i.e., a host entity is connected to zero or more disks and to zero or more network interfaces.
- *Disk*: This type represents data storage devices of hosts, and it is the second main component type for collecting infrastructure monitoring time series. One disk entity is normally linked to exactly one host entity.[\[1\]](#)
- *Network*: This type represents network interfaces of hosts and captures all incoming and outgoing data traffic. It is the third of the three main component types for collecting infrastructure monitoring time series. One network entity is linked to exactly one host entity.

It might happen that some entities are not connected at all, even though we expect them to be linked, for instance, there might be no host for a disk component. Reasons for such cases are either monitoring configurations by system administrators that explicitly disable the collection of component data, or it could be because of data extraction errors or data loss.

The topology can be represented with a graph, where we show an example of a small system in [Figure 2.3](#). In total, there are seven components: Two service entities $S1$ and $S2$ are executed by process $P1$, which runs on host $H1$. This host has two disks $D1$ and $D2$ as well as one network interface $N1$ attached. The time lines next to the entities indicate their lifetimes. The entire system was monitored from timestamp t_1 until t_2 . The host, its network interface and disk $D1$ were active during the entire time. Service $S1$ was active from the start on but was last seen at timestamp $S1_2$ (e.g., the service finished its task). Service $S2$ started somewhere in the middle at timestamp $S2_1$ and ran until $S2_2$. The process that executed the services also finished at around the same time (timestamp $P1_2$). Lastly, disk $D2$ was attached to the host at timestamp $D2_1$ and remained active until the end of the system’s monitoring period.

The above example is a particularly small system with only a single host, e.g., from monitoring a single personal computer. Naturally, there are much larger systems with poten-

¹In extremely rare cases, a disk might be interlinked with multiple hosts, for example, when the same disk is attached to different hosts during its observed lifetime.

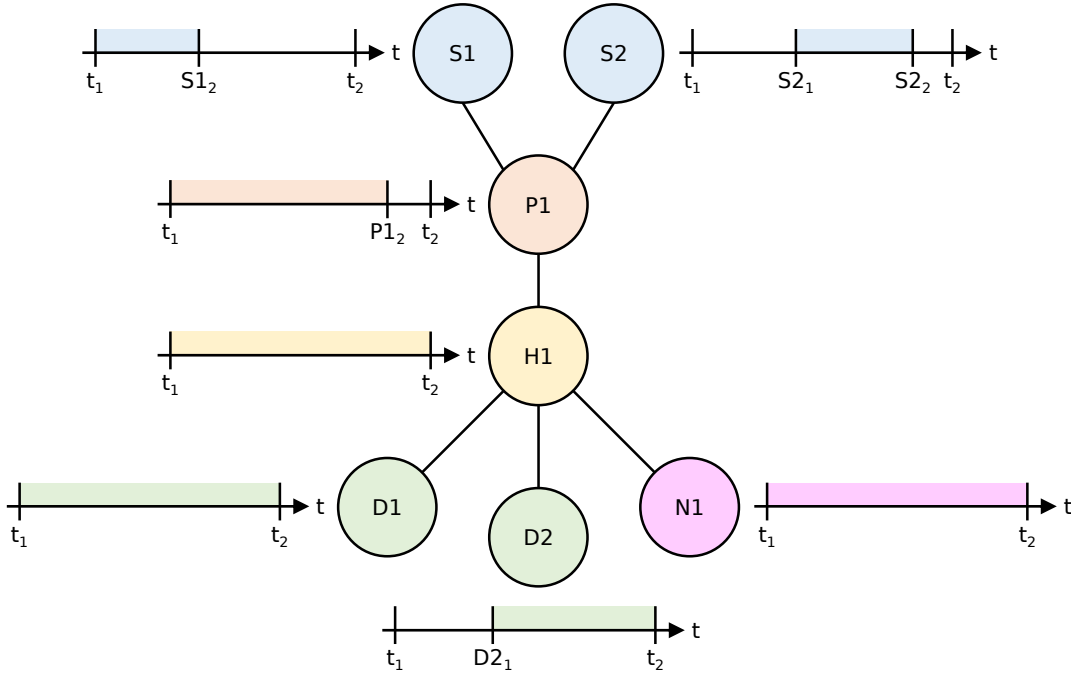


Figure 2.3: Example system as graph visualization including component lifetimes.

tially thousands of entities, making the detailed visualization significantly more challenging. Moreover, the components of the resulting graph do not necessarily need to be all connected. If, for instance, services are executed on processes that only run on a single host and there is no communication between these hosts, then the graph will contain multiple subgraphs (one for each host), i.e., a system can be a collection of unconnected graphs.

2.3.3 Events

Events occur at a specified point in time² and can be either expected (e.g., a process restart) or anomalous. Every event has a unique identifier (ID) and stores the type (defines the category and kind), the entity on which it occurred and the time of the occurrence. Similarly to the component types, Dynatrace collects a plethora of event types, but again, we only need to focus on a few selected ones in this thesis:

- *Process crash*: The anomalous events of this type occur on process entities and, as the name suggests, indicate that the corresponding process crashed. A process crash event can carry additional information on the crash details, such as the name of the error or exception message, the fault location or the process signal.
- *Service (response) slowdown*: This anomalous event occurs on service entities and is heuristically created by comparing the average service response time with the expected baseline and checking for any negative deviation.

Figure 2.4 shows the two service entities and the connected process from the same system as presented in **Figure 2.3**. Now, two events occurred. Somewhere during the lifetime of service *S2*, a service slowdown was identified and an appropriate event *E1* was created at timestamp

²More specifically, some events actually have two timestamps: a minimum and a maximum. These are primarily events that are based on heuristics which work with time windows and allow a more fuzzy definition of when an event occurred. For simplicity, we use the minimum timestamp as the event occurrence.

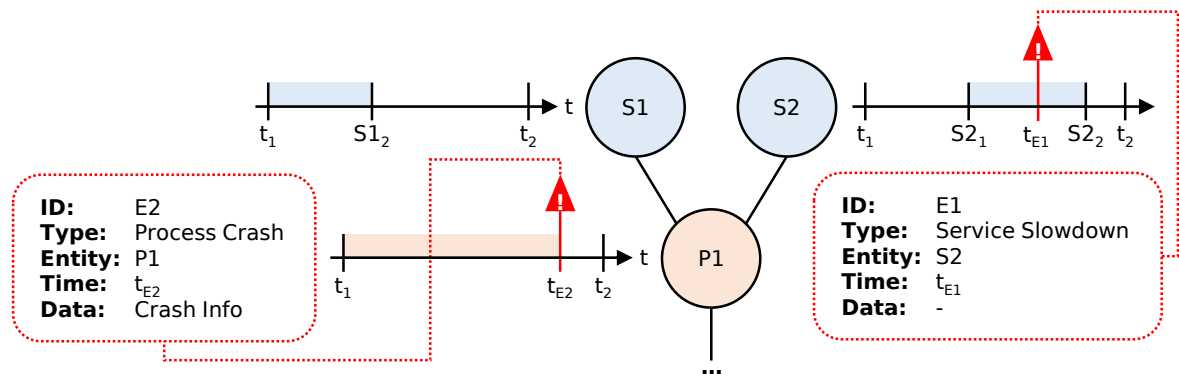


Figure 2.4: Two events occurring in the example system from [Figure 2.3](#)

t_{E1} . Later on, process $P1$ crashed, which led to the event $E2$ at time t_{E2} and, in turn, to the termination of the process since no restart was initiated.

2.3.4 Time Series

A time series \mathbf{x} is a list/vector of n consecutive values of the form $(x_t \mid t \in [1, n])$, where t is called the timestamp and x_t its associated value (typically, $x_t \in \mathbb{R}$). In our multi-system infrastructure monitoring environment, the timestamps are evenly spaced, which means that $t_{i+1} - t_i = \Delta t \forall i \in [1, n - 1]$, although this is not a general requirement for time series. Yet again, Dynatrace monitors much more time series than we use in this thesis. [Table 2.2](#) shows all the series we collect for our component types,³ including a unique identifier (ID), a short description and the unit of measurement. Each time series *kind*, which we also call *metric*, is mapped to exactly one component type, and each individual time series is mapped to exactly one component. For example, we collect the CPU Idle (H-01) metric at components of type *Host*, and for some concrete host $H1$, this could result in the time series \mathbf{x}_{H1} . In total, we have 34 metrics at our disposal: 11 host series, 13 disk series and 10 network interface series. All these time series are evenly spaced and internally sampled every ten seconds, i.e., $\Delta t = 10\text{sec}$. However, when exporting the metrics (cf. [Section 2.3.5](#)), they are only available to us in one-minute resolution ($\Delta t = 1\text{min}$), where the values are averaged over six ten-second samples.

[Figure 2.5](#) shows the same example as [Figure 2.3](#) but now with time series attached to the entities. In the example, the host only provides a single metric, namely CPU Idle (H-01), i.e., the available utilization of the central processing unit. The time series is missing some values, which is indicated by the question mark symbols. For network $N1$, two metrics are collected: the total number of received bytes (N-01) and how many packets were transmitted (N-04). We are also interested in the time series monitoring the amount of used disk space (D-02). There are two disk entities, but only disk $D2$ has data available (bounded by the entity's lifetime). Disk $D1$ is missing the entire time series, for instance, due to a data recording error.

2.3.5 Data Collection and Storage

In this section, we describe how we collect the data from Dynatrace and how we store it afterwards. [Figure 2.6](#) gives an overview. In the left, the multi-system infrastructure monitoring is shown with a cloud symbol, where single systems and their data are represented by a single component graph (cf. [Section 2.3.2](#)), including symbols for events (labeled E) and time series (labeled T). We access this multi-system environment via a REST [\[54\]](#) (representational state

³Note that services and processes can also have time series. However, we do not record them.

Component Type	ID	Time Series Kind/ Metric	Unit	Short Form
Host	H-01	CPU Idle	Percent (%)	CPU Idle
	H-02	CPU System	Percent (%)	CPU System
	H-03	CPU Load	Ratio	CPU Load
	H-04	CPU User	Percent (%)	CPU User
	H-05	CPU IO Wait	Percent (%)	CPU IO Wait
	H-06	Page Faults	Per second	Page Faults
	H-07	Memory Available	Percent (%)	Mem. Avail. %
	H-08	Memory Available	Byte	Mem. Avail.
	H-09	Memory Used	Byte	Mem. Used
	H-10	Swap Available	Byte	Swap Avail.
	H-11	Swap Used	Byte	Swap Used
Disk	D-01	Disk Available	Byte	Disk Avail.
	D-02	Disk Used	Byte	Disk Used
	D-03	Disk Available	Percent (%)	Disk Avail. %
	D-04	Read Bytes	Bytes per second	Read Bytes
	D-05	Written Bytes	Bytes per second	Written Bytes
	D-06	Read Operations	Per second	Read Ops.
	D-07	Write Operations	Per second	Write Ops.
	D-08	Read Time	Millisecond	Read Time
	D-09	Write Time	Millisecond	Write Time
	D-10	Utilization Time	Percent (%)	Util. Time
	D-11	Queue Length	Count	Queue Length
	D-12	Inodes Available	Percent (%)	Inodes Avail. %
	D-13	Inodes Total	Count	Inodes Total
Network	N-01	Bytes Received	Bytes per second	Bytes Rec.
	N-02	Bytes Sent	Bytes per second	Bytes Sent
	N-03	Received Packets	Per second	Rec. Pkts.
	N-04	Sent Packets	Per second	Sent Pkts.
	N-05	Received Packets Dropped	Per second	Rec. Pkts. Drop.
	N-06	Sent Packets Dropped	Per second	Sent Pkts. Drop.
	N-07	Received Packet Errors	Per second	Rec. Pkt. Err.
	N-08	Sent Packet Errors	Per second	Sent Pkt. Err.
	N-09	Receiving Utilization	Percent (%)	Receiving Util.
	N-10	Sending Utilization	Percent (%)	Sending Util.

Table 2.2: Infrastructure monitoring time series. The *Short Form* is simply an abbreviated form of the full metric name, including the unit where necessary to uniquely identify a metric.

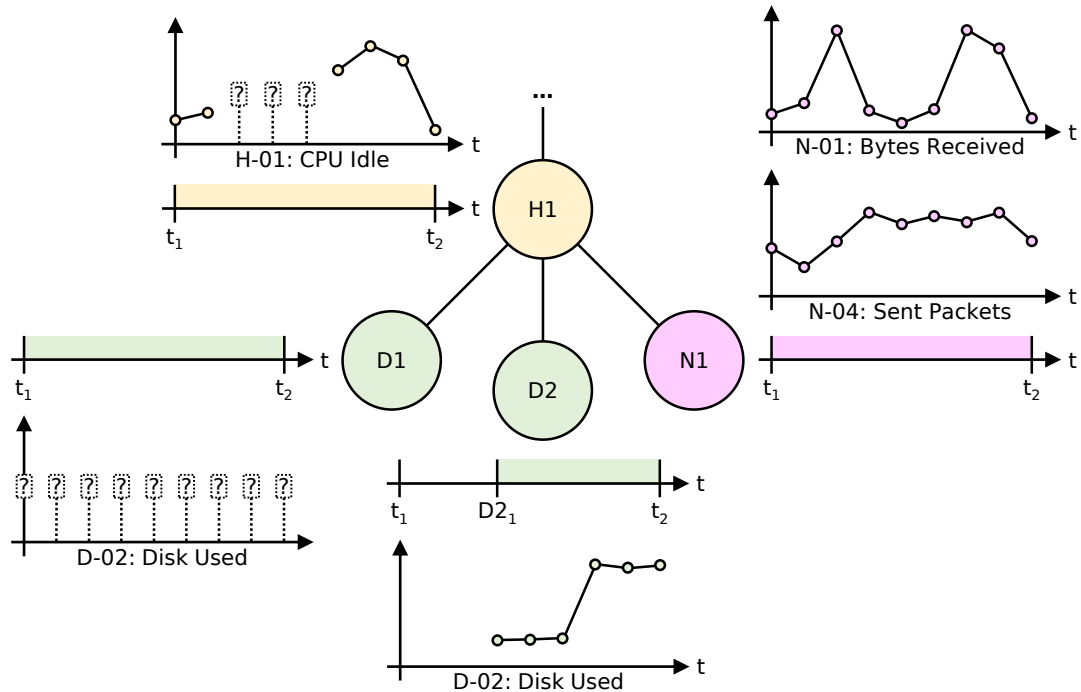


Figure 2.5: Example system from [Figure 2.3](#) with time series.

transfer) API (application programming interface) and extract all the data described in the previous sections. The time series and events are written to an InfluxDB time series database,⁴ whereas the topology is stored in separate JSON files, one for each system.⁵

2.4 Machine Learning

The topic of machine learning covers algorithms and models that automatically “learn” from a given set of sample data with the goal of making predictions on unseen data (supervised task) or extracting interesting or useful information, structures or patterns (unsupervised task). In this section, we first cover the basics of machine learning and common terms, and then we continue with details on data imbalance, supervised learning and unsupervised learning. It must be noted that machine learning is a huge scientific field and that we only focus on the aspects necessary to understand our approaches in this thesis. For in-depth information and further details, we refer the reader to [\[76\]](#), [\[171\]](#), [\[27\]](#).

2.4.1 Basics

When talking about machine learning, *generalization* can be considered a core concept. Generalization means that a machine learning model is capable of applying insights gained from previously seen tasks to new data, i.e., common and discerning data characteristics are

⁴Of course, events themselves are not time series. However, we can treat the set of all events as a single time series by simply sorting them according to the time of their occurrences. Querying this “event time series” is convenient and fast, which is why we decided to store events this way.

⁵Strictly speaking, there are multiple JSON files for each system because we do not export the entire data all at once but rather sequentially in smaller batches to avoid increasing the load on the systems we extract data from. However, these multiple JSON files can simply be merged (add new entities, update all timestamps), which we regard as an implementation detail and not necessary in the context of this thesis.

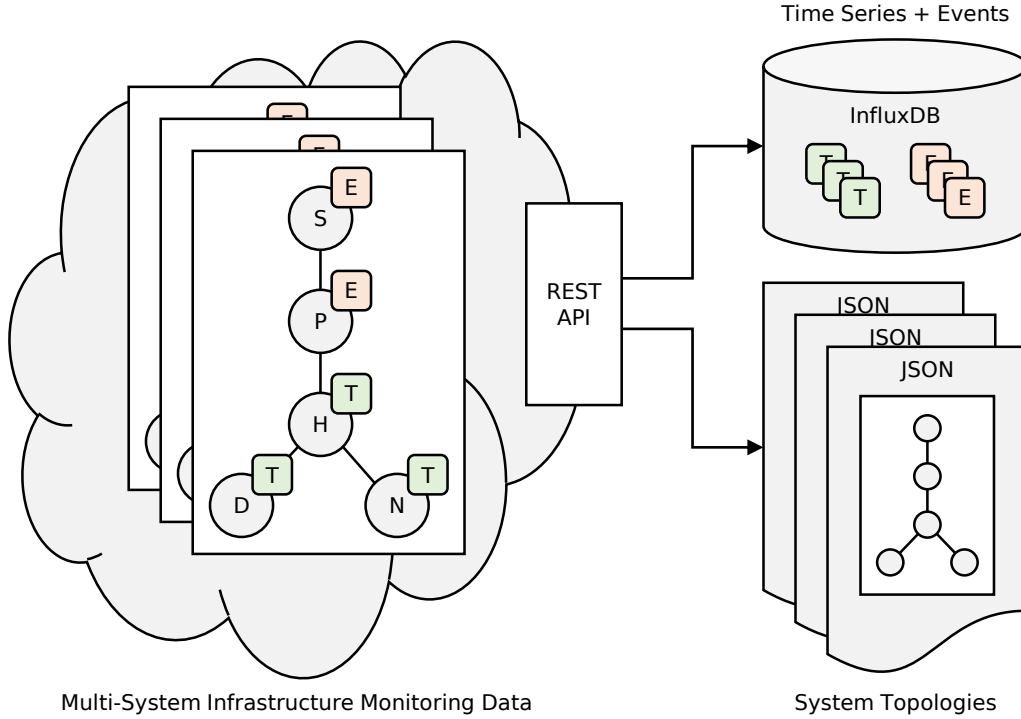


Figure 2.6: Overview of the data collection and storage process.

extracted to learn the concept of the task rather than the specific problem or the solution of this problem itself. The core part of such a task is the data which should be processed. There are just two components: the *data points* and the *labels*. Data points can be expressed as a matrix \mathbf{X} with n observations \mathbf{x}_i (also called feature vectors, samples or rows), where each \mathbf{x}_i consists of k values (also called features, measurements or columns). Labels can be expressed as a row vector with equally many rows as \mathbf{X} , and it stores the associated label y_i (also called target or output) for each observation \mathbf{x}_i . Both \mathbf{X} and \mathbf{y} are defined in [Equation 2.1](#).

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} = \begin{bmatrix} x_{11} & \cdots & x_{1k} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nk} \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \quad (2.1)$$

In *supervised learning*, both \mathbf{X} and \mathbf{y} (labeled data) are required since the goal is to learn the connection between data points and labels (cf. [Section 2.4.3](#) for details), whereas in *unsupervised learning*, only the data points \mathbf{X} (unlabeled data) are needed since the goal is to extract patterns only within the input data (cf. [Section 2.4.4](#) for details).

The data points \mathbf{X} and labels \mathbf{y} can be numerical or categorical. However, many machine learning algorithms cannot handle categorical data out of the box, so transformations such as label binarization are often part of the processing pipeline. For example, assume that \mathbf{y} contains categorical labels that are mapped to one of the two classes **Event** and **NoEvent**, i.e., each label either represents an event occurrence or, alternatively, no event occurrence. A straightforward solution would then be to encode these two classes by simply replacing **Event** with the value +1 and **NoEvent** with the value -1.

2.4.2 Data Imbalance

If the (categorical) labels in \mathbf{y} are not evenly distributed, i.e., if one label value occurs more often than others, we talk about data imbalance. For instance, assume that we have a labeled dataset of 100 samples and two classes A and B . If 50 samples were labeled as class A and 50 samples as class B , we would have a balanced dataset. However, if 90 samples were labeled as class A and 10 samples as class B , we would have an imbalanced dataset.⁶ In this case, A is called the *majority class* and B the *minority class*. Naturally, multi-class datasets (datasets with more than two classes) can also be imbalanced, in which case there are multiple majority and minority classes, depending on the actual distribution. Since imbalanced data is often encountered in real-world scenarios and most machine learning models cannot handle such data directly,⁷ various approaches have been studied to cope with the imbalance problem. The two most common techniques are called *under-* and *oversampling*.

The goal of undersampling is to drop samples from the majority class(es) until the desired class distribution is reached. A simple and straightforward implementation is random undersampling, where the samples that should be excluded are chosen at random. The advantage of this approach is its simplicity and the reduced computational cost for further processing, since the total number of samples decreases. On the other hand, a big disadvantage is the fact that information is lost since we potentially drop samples that store essential data characteristics. There are techniques which try to minimize this problem by reducing the sample size more carefully [102], such as selecting less informative samples (e.g., duplicates or overlapping samples via Tomek links [180] or nearest-neighbor-based heuristics [192]) or generating fewer synthetic samples (new samples that are not part of the original data) that are based on characteristics of the original samples (e.g., prototype generation based on clustering approaches).

Oversampling works in the opposite direction by generating more samples of the minority class(es) until the desired class distribution is reached. Of course, generating more samples means higher computational costs since the total number of samples increases. The advantage, however, is the fact that we do not drop potentially important samples but keep the entire original information. Various oversampling techniques exist, the most simple one is random oversampling, which duplicates minority samples at random. More sophisticated approaches generate new, synthetic samples based on the original data, such as SMOTE [33] (synthetic minority oversampling technique) or ADASYN [74] (adaptive synthetic sampling approach). There is also the option to generate more samples based on altered copies of the original data points, which is called *data augmentation*. Augmentation is prominent in the area of image classification and deep learning scenarios [169], but the general concept applies to any kind of data and approach. The data modifications can be arbitrary and can range from permutation or scaling to adding random noise and more. The specific kind heavily depends on the domain and on the goals the users want to achieve.

⁶Note that the thresholds for balanced and imbalanced class distributions can vary, depending on how many samples we have and which algorithm we use (some are more robust than others). For example, in the above dataset, we would probably consider a class distribution of 55/45 still balanced.

⁷Many models try to find thresholds and decision boundaries based on the available data. The higher the class imbalance is, the more important and decisive the majority class becomes. If, for example, a model were to only rely on the data of the majority class of the above example, it would still be correct 90% of the time. However, the minority class would be ignored completely, which would result in a useless model.

2.4.3 Supervised Learning

In supervised learning, the goal is to learn how the data points \mathbf{X} are linked to their corresponding labels \mathbf{y} and then to predict \mathbf{y}' on new, unseen data \mathbf{X}' . If \mathbf{y} contains categorical data, the learning task is to find a solution how to distinguish the different classes, which is called *classification* (in the special case that \mathbf{y} only contains two classes, it is called binary classification). If \mathbf{y} contains numerical, continuous data, the learning task is to identify how the magnitude of the values in \mathbf{y} correlates with the data \mathbf{X} , which is called *regression*.⁸

2.4.3.1 Training and Testing

Most supervised models learn the relationships in the data in the *training* phase and apply the learned concepts in the *testing* phase, for which we give a brief theoretical introduction that is adapted from [76]. In the training phase, the model is given a *training set* (or train set) $\mathbf{X}_{\text{train}}$ and the corresponding labels $\mathbf{y}_{\text{train}}$, which is a subset of the available data. After this phase, the parameterization vector \mathbf{w} represents the state/configuration of this trained model. The result is the model prediction defined by the function in Equation 2.2:

$$\hat{y} = g(\mathbf{x}; \mathbf{w}) \quad (2.2)$$

which yields the predicted label \hat{y} for a given feature vector \mathbf{x} and model parameterization vector \mathbf{w} . How well this functions performs is determined by the *generalization error*, for which we must define some sort of *loss* that compares the true label y with the predicted label \hat{y} . Equation 2.3 defines the quadratic loss and Equation 2.4 the zero-one loss, which are commonly used loss functions:

$$L(y, g(\mathbf{x}; \mathbf{w})) = (y - g(\mathbf{x}; \mathbf{w}))^2 \quad (2.3)$$

$$L(y, g(\mathbf{x}; \mathbf{w})) = \begin{cases} 0 & \text{if } y = g(\mathbf{x}; \mathbf{w}), \\ 1 & \text{if } y \neq g(\mathbf{x}; \mathbf{w}). \end{cases} \quad (2.4)$$

Formally, Equation 2.5 defines the above mentioned generalization error or *risk* R as the expectation E of the loss function for unseen, future data \mathbf{X}' and \mathbf{y}' , which we do not know (indicated by the dot in $R(g(\cdot; \mathbf{w}))$), but we can approximate it in the testing phase using our *testing set* (or test set) with $\mathbf{X}' \approx \mathbf{X}_{\text{test}}$ and the corresponding labels $\mathbf{y}' \approx \mathbf{y}_{\text{test}}$:

$$R(g(\cdot; \mathbf{w})) = E_{(\mathbf{X}', \mathbf{y}')} (L(y, g(\mathbf{x}; \mathbf{w}))) \approx \frac{1}{m} \sum_{i=1}^m L(y_i, g(\mathbf{x}_i; \mathbf{w})) \quad (2.5)$$

where m is the total number of samples in the test set. There are various ways to determine how well the final model performed on the test set, which we cover in Section 2.4.3.3. It is important to note that the training and test set must not overlap to avoid using test data in the training process, which would invalidate the estimated model performance.

Choosing the function g is the primary goal of the training phase, and it can be accomplished by two approaches: empirical risk minimization (ERM) and structural risk minimization (SRM). In ERM, the goal is to find a function g which minimizes the risk $R(g)$, whereas in SRM, the complexity of g is additionally taken into account. Purely minimizing the risk, which is based on the training data estimate, can lead to *overfitting* (a model with high *variance*), which happens when the function tries to capture every detail of the training data set, including noise.

⁸Note that for classification, the labels can also be numeric. However, in contrast to regression, the magnitude of the values is irrelevant since they are considered to be categorical.

Naturally, this goes against the core concept of generalization and should thus be avoided. Punishing overly complex functions that can model every detail can reduce the variance and yield more appropriate functions. However, the complexity penalty must be chosen carefully to not fall into the direct opposite direction, where only the simplest functions are considered that ignore relevant information, which would lead to *underfitting* (a model with high *bias*). Balancing the two extremes is called the bias-variance trade-off.

2.4.3.2 Variants of Training and Testing

Using training and test sets is only one of the possibilities to estimate the model performance (the quality of the model). Other variants include validation sets, which are used to optimize the model parameters (hyperparameter tuning) before the final model is fit on the test data, i.e., to avoid overfitting to the test set. [Figure 2.7](#) shows an example of both approaches. Here, a 50/50 split is used for the training and test set, and a 50/25/25 split is used for the alternative train-validation-test set approach. Of course, other splits are possible.

Data		
Training Set	Test Set	
Training Set	Validation Set	Test Set

Figure 2.7: Example data split using training and test sets (middle row) or training, validation and test sets (bottom row).

Cross-validation is another approach, where the available data is split into n parts/folds (n -fold cross-validation), and $n - 1$ parts are used for training while the remaining part is used for evaluation. There are a total of n iterations, in each of which fold i will be used as validation fold. If enough data is available, cross-validation can be merged with the train-validation-test set approach, where the cross-validation is performed on the train-validation set but the final model performance is estimated on the test set. [Figure 2.8](#) shows an example of a 5-fold cross-validation, where an additional test set was first split from the available data. Since it is a 5-fold cross-validation, there is one validation fold and four training folds in each of the five iterations.

In typical application scenarios, the available data, or more precisely, the rows of \mathbf{X} and their labels \mathbf{y} , are randomly shuffled before splitting it into the different sets to avoid that certain data characteristics only appear in the one particular set. Alternatively, the different splits can also be provided separately, e.g., if the data of interest are time series and the test set should reflect data which was recorded *after* the training data. For instance, for a 50/50 split of two weeks' worth of time series data, the training set covers week one and the test covers week two. Ultimately, it depends on the task at hand, and it is up to the user which model evaluation approach is selected and how the different splits are created.

2.4.3.3 Evaluation Metrics

After fitting a machine learning model on the validation or test data, we want to determine how well the predictions $\hat{\mathbf{y}}$ match the actual/true labels \mathbf{y} . For regression, there are various metrics which can be directly computed based on the two label sets. We list two of the most commonly used ones:

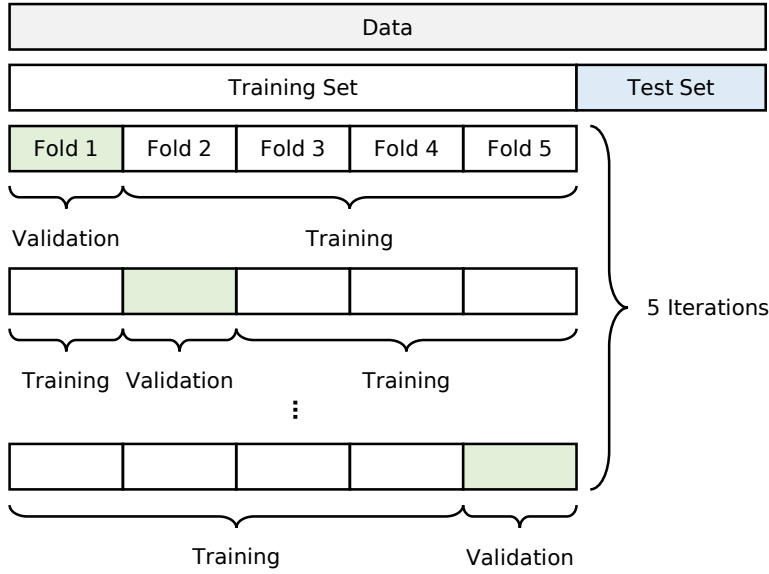


Figure 2.8: Example data split using 5-fold cross-validation with an additional test set that is only used once after the cross-validation.

- *Mean Squared Error* (MSE): This metric is the average squared deviation of the predicted labels to the true labels. It is the same as the expected risk when using the quadratic loss function (cf. [Equation 2.3](#) and [Equation 2.5](#)). The MSE is defined in [Equation 2.6](#) for predicted labels $\hat{\mathbf{y}} = (\hat{y}_1, \dots, \hat{y}_n)$ and actual labels $\mathbf{y} = (y_1, \dots, y_n)$:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (2.6)$$

- *Mean Absolute Error* (MAE): This metric is the average absolute difference between the predicted labels and the true labels. Compared to the MSE, outliers are much less punished. The MAE is defined in [Equation 2.7](#) for predicted labels $\hat{\mathbf{y}} = (\hat{y}_1, \dots, \hat{y}_n)$ and actual labels $\mathbf{y} = (y_1, \dots, y_n)$:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| \quad (2.7)$$

In classification scenarios, the *confusion matrix* lists the predicted labels and compares them to the actual labels, from which we can then calculate various evaluation metrics that help us in analyzing the performance of our model. [Table 2.3](#) shows how such a confusion matrix looks like for a given a set of c unique classes C_i within the n labels from \mathbf{y} . The columns store the actual classes of \mathbf{y} , whereas the rows store the predicted classes of $\hat{\mathbf{y}}$. The cells x_{ij} then store how often a sample was classified as C_i but was actually of class C_j . The main diagonal of the confusion matrix ($i = j$) thus holds all matching/correct predictions, and all other cells contain mismatching/incorrect predictions. If there are n labels, then the sum of the confusion matrix cells must be equal to n , more formally, $\sum_{i,j=1}^c x_{ij} = n$.

In case of binary classification, we only have two unique classes, i.e., $c = 2$ with C_1 and C_2 . For convenience, we will set $C_1 = -1$ and $C_2 = +1$ and refer to them as the *negative class* and the *positive class*, respectively (we will also call the associated samples *negative samples* and *positive samples*). This leads to a simplified, binary confusion matrix shown in [Table 2.4](#), where the previous entries x are replaced by the following four human-readable counts:

		Actual		
		$y = C_1$	\dots	$y = C_c$
Predicted	$\hat{y} = C_1$	x_{11}	\dots	x_{1c}
	\vdots	\vdots	\ddots	\vdots
	$\hat{y} = C_c$	x_{c1}	\dots	x_{cc}

Table 2.3: Confusion matrix with c classes.

- *TN = True Negative*: Counts how many negative samples have been correctly predicted as negative ($y = -1, \hat{y} = -1$).
- *FN = False Negative*: Counts how many positive samples have been incorrectly predicted as negative ($y = +1, \hat{y} = -1$).
- *FP = False Positive*: Counts how many negative samples have been incorrectly predicted as positive ($y = -1, \hat{y} = +1$).
- *TP = True Positive*: Counts how many positive samples have been correctly predicted as positive ($y = +1, \hat{y} = +1$).

		Actual	
		$y = -1$	$y = +1$
Predicted	$\hat{y} = -1$	TN	FN
	$\hat{y} = +1$	FP	TP

Table 2.4: Binary confusion matrix.

Based on these numbers, we can calculate different evaluation metrics, where we will list the ones used in this thesis in the following:

- *Accuracy (ACC)*: This metric shows how many samples out of all samples were correctly predicted. In other words, it can be used to answer the question “How often was our model correct?”. It yields values in the range $[0, 1]$ (higher is better), and it is formally defined in [Equation 2.8](#):

$$\text{ACC} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (2.8)$$

- *True Positive Rate (TPR)*: This metric (also called sensitivity or recall) shows how many samples out of all actually positive samples were correctly predicted as positive. In other words, it can be used to answer the question “How many positive samples was our model able to detect?”. It yields values in the range $[0, 1]$ (higher is better), and it is formally defined in [Equation 2.9](#):

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (2.9)$$

- *Positive Predictive Value (PPV)*: This metric (also called precision) shows how many samples out of all predicted positive samples are actually positive. In other words, it can

be used to answer the question “Out of all the predicted positive samples, how many are actually correct?”. It yields values in the range $[0, 1]$ (higher is better), and it is formally defined in [Equation 2.10](#):

$$\text{PPV} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (2.10)$$

- *False Positive Rate* (FPR): This metric (also called fall-out) shows how many samples out of all actually negative samples were incorrectly predicted as positive. In other words, it can be used to answer the question “How often did our model erroneously predict an actually negative sample as positive?”. It yields values in the range $[0, 1]$ (lower is better), and it is formally defined in [Equation 2.11](#):

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \quad (2.11)$$

- *F1 Score* (F1): This metric (also called F-measure) is the harmonic mean of the PPV and the TPR (precision and recall) and is thus a more expressive measure (both other metrics must be high in order for the F1 score to be high). It yields values in the range $[0, 1]$ (higher is better), and it is formally defined in [Equation 2.12](#):

$$\text{F1} = \frac{2}{\frac{1}{\text{PPV}} + \frac{1}{\text{TPR}}} = 2 \cdot \frac{\text{PPV} \cdot \text{TPR}}{\text{PPV} + \text{TPR}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}} \quad (2.12)$$

- *Matthews Correlation Coefficient* [\[115\]](#) (MCC): This metric takes all confusion matrix entries into account and calculates a correlation value. Unlike all other metrics, it is robust against imbalanced data [\[22\]](#) and yields values in the range $[-1, 1]$ (higher is better), where -1 represents the worst possible prediction (every sample was misclassified), 0 represents a random assignment and $+1$ represents the best possible prediction (every sample was correctly classified). It is formally defined in [Equation 2.13](#):

$$\text{MCC} = \frac{\text{TP} \cdot \text{TN} + \text{FP} \cdot \text{FN}}{\sqrt{(\text{TP} + \text{FP}) \cdot (\text{TP} + \text{FN}) \cdot (\text{TN} + \text{FP}) \cdot (\text{TN} + \text{FN})}} \quad (2.13)$$

Of course, these metrics can be extended to multi-class problems ($c > 2$). One way is to micro-average the confusion matrix counts, which means to globally sum up all counts, i.e., sum all TN, FN, FP and TP of all individual classes, and then calculate the selected metric based on these global counts. Another option is macro-averaging, where the selected metric is calculated for each individual class (C_i vs $\neg C_i \forall i \in [1, c]$), and then the resulting c metrics are averaged.

2.4.3.4 Random Forests

There are many machine learning models for supervised learning, ranging from simple decision trees and k-nearest neighbor models to support vector machines and artificial neural networks. In this thesis, we focus on random forests [\[24\]](#), which are part of *ensemble learning*, meaning that multiple models are used for the final prediction in order to improve the results. In case of a random forest, the inner models are multiple decision trees, and the final prediction is based on, e.g., an average of the predictions (for regression) or on the class that was predicted most often (for classification), although other methods are possible.

In short, a decision tree splits the samples of the input data in such a way that the classes of the labels are best separated. Starting from all samples at the root node, a discriminating

feature is selected and the samples are split into two nodes based on a chosen feature threshold. This procedure is repeated recursively until some stopping criterion is reached, e.g., the maximum tree depth is reached or the node cannot be split anymore (e.g., only contains samples from a single class). After the training phase, predictions can be made by following the different tree branches (according to the tree's feature thresholds) until a leaf node is reached. The leaf node either represents the predicted class for the current sample, or it contains probabilities of all the possible classes.

The randomness in *random* forests was introduced to reduce variance, i.e., to avoid the overfitting of individual decision trees, and it can be enabled in two main phases of the learning process. The first one is the sampling phase, where a random subset of the original data is drawn with replacement, so the different inner decision trees operate on different data. In each decision tree, the second phase is selecting how many features (again, a random subset) to use from the sampled data for making a tree split. The concrete sampling method as well as the splitting configurations are often implementation-specific and can be adjusted through various hyperparameters [131], which also include settings such as the number of inner decision trees and their maximum depth.

Another important part of a random forest is its built-in feature importance evaluation, i.e., an evaluation of the discriminating properties of features with respect to the class labels. Since the internal decision trees split the data and their labels based on discriminating features of the input data, we know after training the model which features these are, i.e., we know which of our features are more discriminating and thus more important (the labels could be separated well), and which of our features are less discriminating and thus less important (the labels could not be separated well). Figure 2.9 shows a small example of ten labeled samples (five samples are of class *A* and five of class *B*), each represented with three features f_1 , f_2 and f_3 , and a possible split by a single decision tree using the default implementation of scikit-learn [131]. This implementation relies on the so-called Gini importance (also referred to as Mean Decrease Impurity), which basically measures how good a split is, i.e., the better the classes can be separated, the more “pure” the nodes become (for more information, we refer the interested reader to [24, 108]). The example reveals that feature f_3 was selected as the most discriminating/important feature (importance value of 0.57), followed by f_1 (importance value of 0.43) and lastly f_2 (importance value of 0). In the context of this thesis, it suffices to know which features performed better than others, i.e., we do not need the exact importance values. Using the example from above, it is thus only relevant for us that the ordering/ranking is $f_3 \rightarrow f_1 \rightarrow f_2$ (from most important to least important feature).

2.4.4 Unsupervised Learning

In unsupervised learning, the goal is to extract characteristics of the data points \mathbf{X} only, i.e., no labels are required. There are two main approaches: methods that find discriminating features within the data and generative models that try to model the underlying data distribution.

For the former, the two most important types are *projection methods* and *clustering*. Projection methods aim to project or transform the input data into a (typically) low-dimensional space, where essential data characteristics, patterns and clusters can be identified more easily. They can also be useful for visualizing high-dimensional data by reducing the dimensionality to three or two components. Often, information loss is an issue that must be kept in mind, especially when the dimensionality reduction is significant. Clustering aims to identify patterns that can then be used for creating clusters of similar data, i.e., splitting the data into groups with common characteristics. Examples of projection methods include Principal Component Analysis [194] (PCA, linear dimensionality reduction) or t-distributed

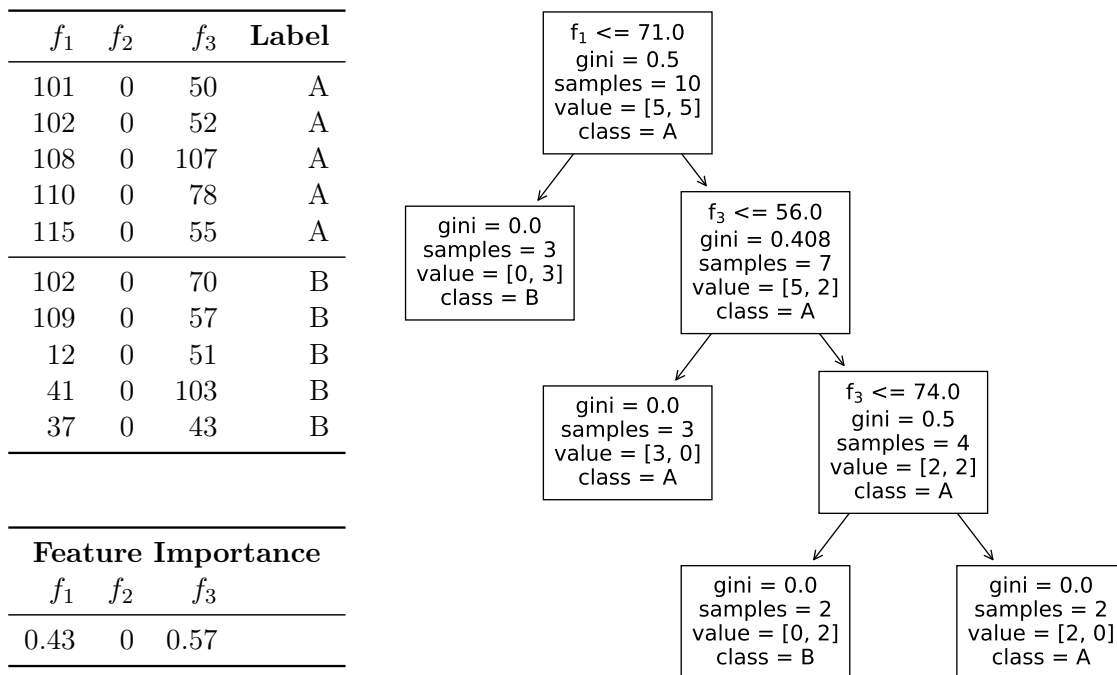


Figure 2.9: Example of a decision tree using the default scikit-learn implementation [131] and how it split ten labeled samples according to the values of three features, including their resulting feature importance values. The information displayed in a node includes the predicted *class* after the training phase, the *values* it contains (i.e., how many samples of class *A* and *B*, respectively), its total number of *samples* (equal to the sum of *value*), the *Gini* importance (cf. [24, 108] for details) and, for non-leaf nodes, the feature f_i and the condition used for splitting. When aggregated over all nodes, the (normalized) Gini importance for all three features is 0.43 for f_1 , 0 for f_2 and 0.57 for f_3 (cf. Section A.1 on p. 197 for details), which means that the ordering/ranking from most important to least important feature is $f_3 \rightarrow f_1 \rightarrow f_2$.

Stochastic Neighbor Embedding [111] (t-SNE, non-linear dimensionality reduction), and an example of clustering is hierarchical clustering.

Generative models, on the other hand, try to model the hidden, true data distribution given by \mathbf{X} , which can then be used to generate data points based on the identified, estimated distribution. The aim is to generalize and find an appropriate representation for the training data rather than simply memorizing it. Again, clustering is a common type of generative unsupervised learning. The goal of finding common patterns and groups of similar data is the same as described above, however, the means to reach this goal differ (data inspection compared to model creation). Examples include Gaussian mixture models or k-means clustering.

The theoretical background for calculating the loss and estimating the risk is more complicated than for the supervised setting. However, such detailed information is not necessary in the context of this thesis, so we refer the interested reader to [76].

2.4.4.1 Evaluation Metrics

Predominantly for clustering models, we would again like to know how well they perform, but unlike with the supervised approach, we often do not have any “ground truth” available (e.g., labels \mathbf{y} indicating the true cluster assignment of the data samples), which we could use for the performance evaluation. Instead, several measures exist that do not require labels, which are called *internal* evaluation metrics. A problem with this kind of evaluation is that the metrics themselves represent some sort of clustering order, i.e., calculating a metric for a given clustering may correlate with how this clustering was achieved. If the machine learning model extracted the clusters based on similar traits which also the evaluation metric uses for the score calculation, then chances are high that such a model will get higher scores. Conversely, if the model relies on completely different data characteristics, low scores can be expected, which, however, does not necessarily mean that the clustering is bad, only that the evaluation metric and the model’s optimization objective do not align. We list some of the commonly used metrics, which all define good clusters to be dense/compact and well separated, and bad clusters to be sparse and/or overlapping:

- *Silhouette coefficient* [147]: This metric measures for each sample how close/similar it is to objects in the same cluster (intra-cluster distance) and how far away/dissimilar it is to objects in the next nearest clusters (inter-cluster distance). It yields values in the range $[-1, 1]$ (higher is better), where -1 means the worst possible clustering, 0 means overlapping clusters and $+1$ means the best possible clustering.
- *Davies-Bouldin index* [46]: This metric measures for each cluster how similar it is to the next nearest cluster by calculating the ratio of the sum of the two cluster sizes/diameters (average distance of all objects to the cluster centroid) to their distance. It yields values in the range $[0, \text{inf})$ (lower is better), where 0 means the best possible clustering and higher values indicate worse clustering.
- *Dunn index* [52]: This metric measures the ratio of the minimum distance between clusters (inter-cluster distance) to the maximum distance within clusters (intra-cluster distance/cluster diameter). It yields values in the range $[0, \text{inf})$ (higher is better), where 0 means the worst possible clustering and higher values indicate better clustering.
- *Calinski-Harabasz index* [28]: This metric (also called Variance Ratio Criterion) measures the ratio of the dispersion within clusters (intra-cluster dispersion) and the dispersion between other clusters (inter-cluster dispersion). It yields values in the range $[0, \text{inf})$

(higher is better), where 0 means the worst possible clustering and higher values indicate better clustering.

To get an overview how the scores of the above metrics are affected by different datasets with different clusters, [Figure 2.10](#) shows ten different examples. Each example contains a dataset with color-encoded clusters, and the table next to the dataset plot lists the corresponding values of all internal evaluation metrics described above.

Note that all these metrics include the calculation of distances as well as determining cluster centroids or cluster diameters and the definition of inter- and intra-cluster distances (e.g., average distance between all sample pairs, average distances based on the centroids, smallest/largest distance of any sample pair), which is often implementation-specific and/or parameterizable [\[131\]](#). Since distances are essential, we also briefly mention a few of the commonly used distance metrics:

- *Euclidean distance*: straight line distance, formally defined in [Equation 2.14](#) for vectors \mathbf{u} and \mathbf{v} of length n :

$$d(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^n (u_i - v_i)^2} \quad (2.14)$$

- *Manhattan distance*: city block distance, formally defined in [Equation 2.15](#) for vectors \mathbf{u} and \mathbf{v} of length n :

$$d(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^n |u_i - v_i| \quad (2.15)$$

- *Cosine distance*: distance based on the cosine similarity, formally defined in [Equation 2.16](#) for vectors \mathbf{u} and \mathbf{v} of length n :

$$d(\mathbf{u}, \mathbf{v}) = 1 - \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|} = 1 - \frac{\sum_{i=1}^n u_i v_i}{\sqrt{\sum_{i=1}^n u_i^2} \sqrt{\sum_{i=1}^n v_i^2}} \quad (2.16)$$

In case we do have knowledge of the “ground truth”, e.g., labels \mathbf{y} indicating the true cluster assignment are available, we can use *external* evaluation metrics. In principle, these metrics can be compared to the ones used for evaluating classification problems in supervised machine learning, with the distinction that the cluster labels can be permuted without affecting the score, nor do they have to be exactly equal. For example, given true labels $\mathbf{y} = (0, 0, 0, 1, 1)$, all the following predicted cluster assignments (also called partitions) would be regarded as a perfect clustering: $\hat{\mathbf{y}} = (1, 1, 1, 0, 0)$ (permutation), $\hat{\mathbf{y}} = (2, 2, 2, 1, 1)$ (non-matching labels), $\hat{\mathbf{y}} = (1, 1, 1, 2, 2)$ (permutation and non-matching labels). External evaluation metrics also come with some problems. First and foremost, they require the presence of labels, which is often not the case in real-world applications. Moreover, the given labels may only indicate *one* clustering, but there might be additional or different clusters, which could also be valid. We list some metrics in the following:

- *Adjusted Rand index* [\[78\]](#) (ARI): This metric measures the similarity of two cluster assignments and is comparable to the accuracy score (cf. [Equation 2.8](#)). It is based on the Rand index (RI) but corrected for chance, so the ARI yields values between $[-1, 1]$ (higher is better), where -1 means a complete disagreement to the true cluster

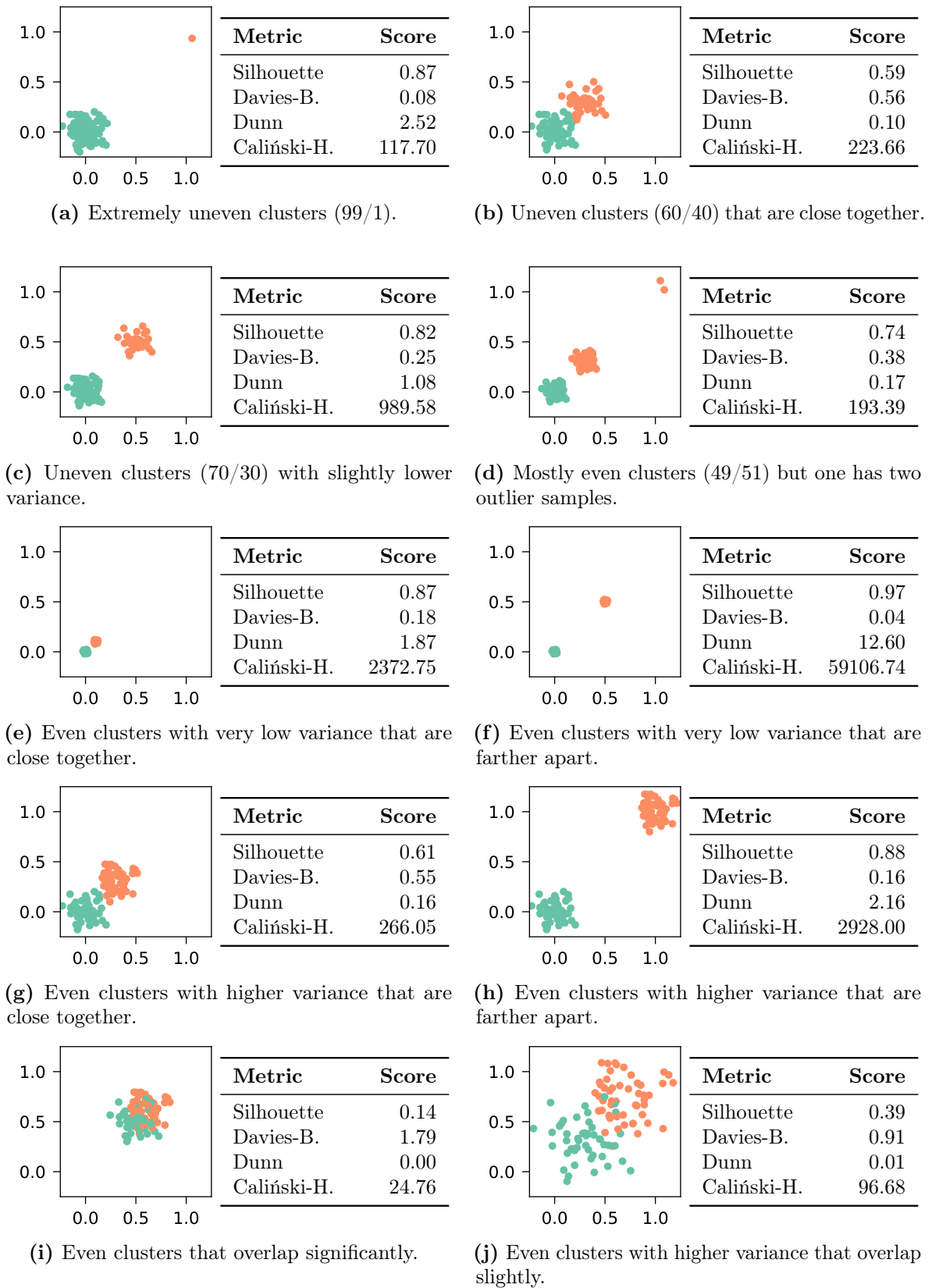


Figure 2.10: Internal evaluation metrics and their scores for different datasets, each of which has 100 samples that are distributed among two clusters.

labels, values around 0 represent random cluster assignments and +1 means a perfect clustering. The RI is defined in [Equation 2.17](#), followed by the definition of the ARI in [Equation 2.18](#), where TP is the number of pairs of samples that are part of the same clusters in \mathbf{y} and $\hat{\mathbf{y}}$ (true positives), TN is the number of pairs of samples that are both in different clusters in \mathbf{y} and $\hat{\mathbf{y}}$ (true negatives), n is the total number of samples and $E(\text{RI})$ is the expected RI of random labeling:

$$\text{RI} = \frac{\text{TP} + \text{TN}}{\binom{n}{2}} = \frac{\text{TP} + \text{TN}}{\frac{n(n-1)}{2}} \quad (2.17)$$

$$\text{ARI} = \frac{\text{RI} - E(\text{RI})}{\max(\text{RI}) - E(\text{RI})} \quad (2.18)$$

- *V-measure* [\[146\]](#): This metric is the harmonic mean of the homogeneity h (a cluster only contains samples of a single class) and the completeness c (all samples of a single class are contained in a single cluster). Depending on whether one of the two should be weighted more, parameter β can be adjusted to values larger than one (focus completeness) or lower than one (focus homogeneity). It yields values in the range $[0, 1]$ (higher is better), and it is formally defined in [Equation 2.19](#):

$$V_\beta = \frac{(1 + \beta) \cdot h \cdot c}{\beta \cdot h + c} \quad (2.19)$$

- *Fowlkes-Mallows index* [\[56\]](#) (FMI): This metric is the geometric mean of the precision (cf. [Equation 2.10](#)) and the recall (cf. [Equation 2.9](#)), which is similar to the F1 score that uses the harmonic mean instead (cf. [Equation 2.12](#)). It yields values in the range $[0, 1]$ (higher is better), and it is formally defined in [Equation 2.20](#), where TP is the number of pairs of samples that are part of the same clusters in \mathbf{y} and $\hat{\mathbf{y}}$ (true positives), FP is the number of pairs of samples that are part of the same cluster in \mathbf{y} but part of different clusters in $\hat{\mathbf{y}}$ (false positives) and FN is the number of pairs of samples that are part of different clusters in \mathbf{y} but in the same cluster in $\hat{\mathbf{y}}$ (false negatives):

$$\text{FMI} = \sqrt{\frac{\text{TP}}{\text{TP} + \text{FP}} \cdot \frac{\text{TP}}{\text{TP} + \text{FN}}} = \frac{\text{TP}}{\sqrt{(\text{TP} + \text{FP}) \cdot (\text{TP} + \text{FN})}} \quad (2.20)$$

Since both internal and external evaluations pose different problems, human evaluation is still an important part in clustering. The different metrics, appropriate visualizations and cluster statistics can be used as guidance to determine whether the results are satisfying.

2.4.4.2 t-distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE [\[111\]](#) is a projection method for reducing the dimensionality of high-dimensional data for the purpose of visualization and revealing structures. The original, high-dimensional samples are represented with a probability distribution, where similar samples have a higher probability than dissimilar samples, and afterwards, a similar probability distribution is constructed in the low-dimensional (commonly 2D) space. The Kullback-Leibler divergence [\[96\]](#) (metric for determining the difference between two probability distributions) is then minimized for these two distributions. The algorithm's performance can be controlled by the hyperparameter *perplexity* that “can be interpreted as a smooth measure of the effective number of neighbors” [\[111\]](#), “which says (loosely) how to balance attention between local and global aspects of your data” [\[188\]](#). Wattenberg et al. [\[188\]](#) state that the resulting visual clusters depend on the chosen perplexity

and multiple results with different values should be analyzed, where typical values range from five to 50 [111]. Moreover, cluster sizes and distances between clusters might not mean anything [188], so care must be taken to avoid false assumptions based on these visual cues.

Figure 2.11 shows the application of t-SNE on two datasets with varying perplexity values p . In Figure 2.11a, the first dataset is made up from two clusters C_1 and C_2 containing 2D points (X_i, Y_i) ⁹ each with 100 samples drawn from the normal distributions $X_1, Y_1 \sim \mathcal{N}(1, 0.1^2)$ and $X_2, Y_2 \sim \mathcal{N}(2, 0.1^2)$, respectively. The data points are color-encoded according to the distribution they were sampled from (“ground truth”), however, t-SNE has no knowledge of the true cluster labels and also does not produce/predict any output labels since it is a projection method and not a clustering model.¹⁰ We can see that for $p = 2$, $p = 10$ and $p = 100$, the resulting visualizations do not reveal any structure in the data, whereas the other perplexity values work just fine. In Figure 2.11b, we added a third cluster C_3 with samples drawn from $X_3 \sim \mathcal{N}(2, 0.1^2)$ and $Y_3 \sim \mathcal{N}(1.7, 0.1^2)$, i.e., a cluster whose y-coordinates are slightly below those from cluster C_2 . Again, we can see that more extreme perplexity values do not seem to work well. However, $p = 10$ not only yields a much better visualization than for the first dataset but arguably also the best among all other perplexity values since it can tell the rather close and thus partially overlapping clusters C_2 and C_3 apart.

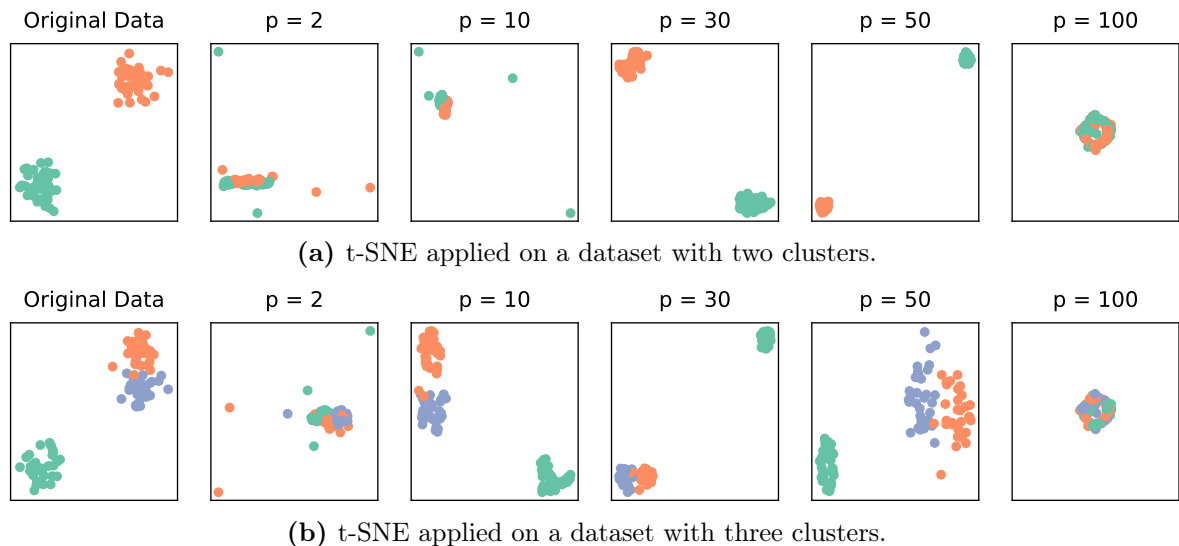


Figure 2.11: Varying perplexity values p when applying t-SNE on two example datasets.

2.4.4.3 k-Means

The k-means algorithm [112] tries to assign the n samples of \mathbf{X} to k clusters \mathcal{C} , where each sample \mathbf{x}_i should be in the cluster C_j (with $j \in [1, k]$) whose mean/centroid $\boldsymbol{\mu}_j$ is closest given the squared Euclidean distance (cf. Equation 2.14). In other words, k-means tries to minimize the within-cluster sum of squares (intra-cluster variance) by choosing appropriate centroids, which is formally defined in Equation 2.21:

$$\sum_{i=0}^n \min_{\boldsymbol{\mu}_j \in \mathcal{C}} (\|\mathbf{x}_i - \boldsymbol{\mu}_j\|^2) \quad (2.21)$$

⁹Note that for the purpose of this example, the data is already in a low-dimension, for which there would be no actual need of applying t-SNE as it can be visualized out of the box.

¹⁰However, we can use the visualizations to assign cluster labels afterwards, e.g., by manual inspection.

The cluster centroids are calculated based on the arithmetic mean, and they are typically not part of the original samples. At the start, k centroids are generated randomly, and then the samples are assigned according to their nearest distance, creating k clusters. Afterwards, the centroids are updated based on the samples in the corresponding clusters. This process is repeated until the algorithm converges or a predefined limit is reached. To speed up this convergence, the initial centroids can be chosen more carefully, e.g., using k-means++ [6]. The number of clusters k must be chosen at the beginning, which can be difficult if the number of expected clusters is unknown as it is often the case in real-world scenarios. This can be addressed, e.g., using the elbow method (different k are tested, their resulting explained variance is plotted against k , and then the k is chosen according to the “elbow” of the plot, i.e., where most variance is explained and higher k do no longer significantly contribute to an even higher explained variance) or using any of the internal evaluation metrics.

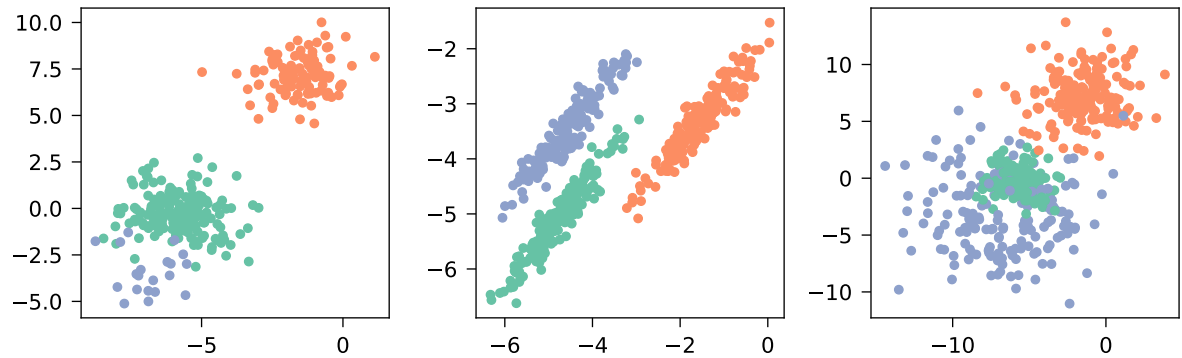
k-means is an often used algorithm due to its simplicity and fast performance which makes it applicable even for large datasets. However, there are some caveats that must be considered, one of which is the fact that the intra-cluster variance minimization as defined above assumes isotropic clusters (the variance is equal in all directions), so differently shaped clusters might not be identified correctly. Standardizing the input data can be helpful in this regard (cf. Section 2.5.1). Figure 2.12 presents various clustering examples of unevenly sized clusters, anisotropic clusters and clusters with different variances. The actual clusters are shown in Figure 2.12a, whereas the predicted clusters using k-means are presented in Figure 2.12b.

2.4.4.4 Hierarchical Clustering

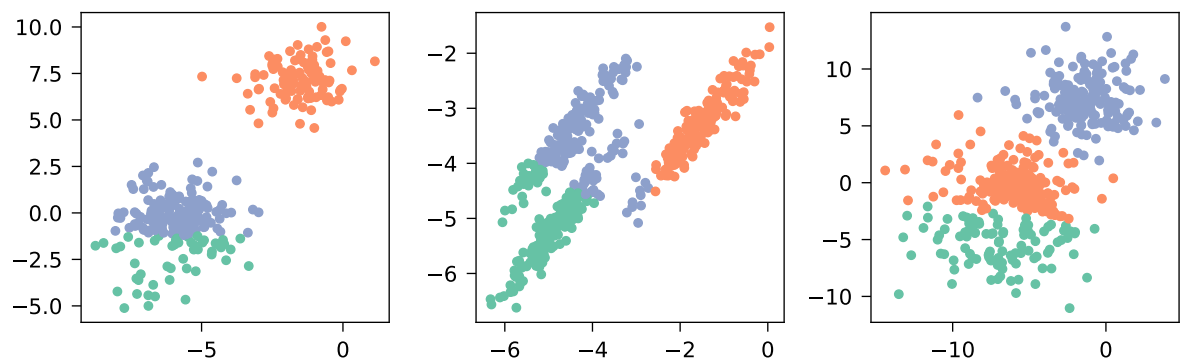
The main approach of hierarchical clustering is the step-by-step process of creating clusters, which can be divided into two categories: divisive and agglomerative clustering [145]. Divisive clustering is a top-down approach, where all samples of \mathbf{X} initially start in one single cluster, and this cluster is then repeatedly split into smaller subclusters until every sample is contained in its own cluster. Agglomerative clustering works in the exact opposite direction, i.e., it is a bottom-up approach where samples are continually merged into growing clusters until all samples are in one final cluster. The following concepts are described based on the latter.¹¹

Hierarchical agglomerative clustering starts with the n samples in n clusters \mathbf{C} , i.e., $C_i = \{\mathbf{x}_i\} \forall i \in [1, n]$. Then, the two closest clusters are merged based on some distance metric d (cf. the different distance metrics listed in Section 2.4.4.1) and on some *linkage* criterion D , which can be considered as a distance measure between clusters. The objective of the merging process is thus the minimization of distances. This results in $n - 1$ clusters, where the merged cluster C_{ij} now contains all samples from C_i and C_j , which are two samples at the very beginning, i.e., $C_{ij} = C_i \cup C_j = \{\mathbf{x}_i, \mathbf{x}_j\}$. The merging of the two closest clusters is then repeated until only one cluster remains which contains all samples $\mathbf{x}_i \in \mathbf{X}$ (stopping at an earlier merging step where still more clusters are available can be achieved by looking at the distances of the individual clusters (using a *dendrogram*) as explained further below with an example shown in Figure 2.13). There are several linkage criteria, some of them we list in the following:

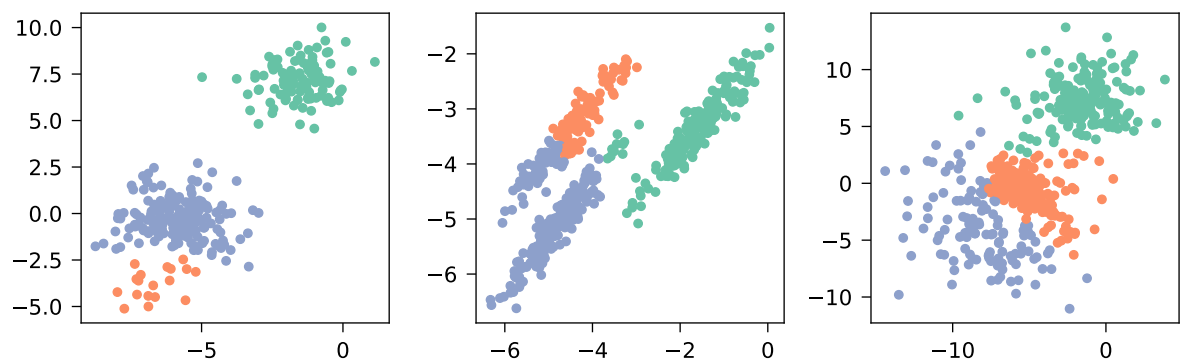
¹¹The concepts can be applied to divisive clustering as well but just in their opposite direction (e.g., splitting instead of merging or farthest distance instead of nearest distance).



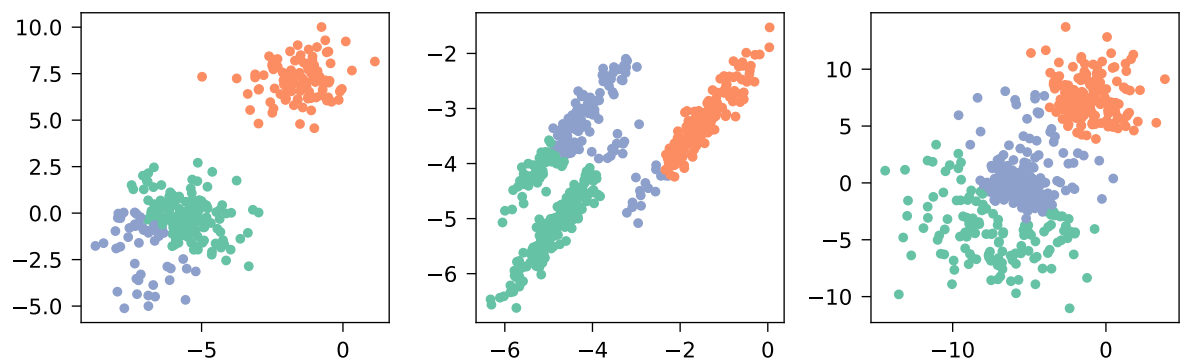
(a) The three datasets, each with their three actual clusters: different cluster sizes (left), anisotropic clusters (middle), clusters with different variances (right).



(b) The clusters predicted by k-means.



(c) The clusters predicted by agglomerative clustering with Euclidean distance and Ward's method.



(d) The clusters predicted by BIRCH.

Figure 2.12: Three example datasets with three clusters and their predicted clusters using different unsupervised machine learning models. The cluster colors vary between the models, which does not mean anything since the cluster labels are independent of any ordering.

- *Single linkage*: This criterion (also called minimum linkage) determines the minimum distance between all sample pairs of two clusters. It uses some specified vector distance d to calculate the cluster distance D between (potentially differently sized) clusters U and V and their samples \mathbf{u} and \mathbf{v} as defined in [Equation 2.22](#):

$$D(U, V) = \min_{\mathbf{u} \in U, \mathbf{v} \in V} d(\mathbf{u}, \mathbf{v}) \quad (2.22)$$

- *Complete linkage*: This criterion (also called maximum linkage) determines the maximum distance between all sample pairs of two clusters. It uses some specified vector distance d to calculate the cluster distance D between (potentially differently sized) clusters U and V and their samples \mathbf{u} and \mathbf{v} as defined in [Equation 2.23](#):

$$D(U, V) = \max_{\mathbf{u} \in U, \mathbf{v} \in V} d(\mathbf{u}, \mathbf{v}) \quad (2.23)$$

- *Weighted average linkage* [\[172\]](#): This criterion (also called WPGMA for Weighted Pair Group Method with Arithmetic Mean) determines the average distance between all sample pairs of two clusters without explicitly accounting for their sizes. It calculates the cluster distance D between (potentially differently sized) clusters U and V , where cluster U was previously constructed from subclusters S and T as defined in [Equation 2.24](#):

$$D(U, V) = D(S \cup T, V) = \frac{D(S, V) + D(T, V)}{2} \quad (2.24)$$

Before the above formula can be applied, the initial distances between the individual samples (single-sample clusters) have to be calculated based on some specified vector distance d , i.e., $D(U', V') = d(\mathbf{u}, \mathbf{v}) \forall U' = \{\mathbf{u}\}, V' = \{\mathbf{v}\}$.

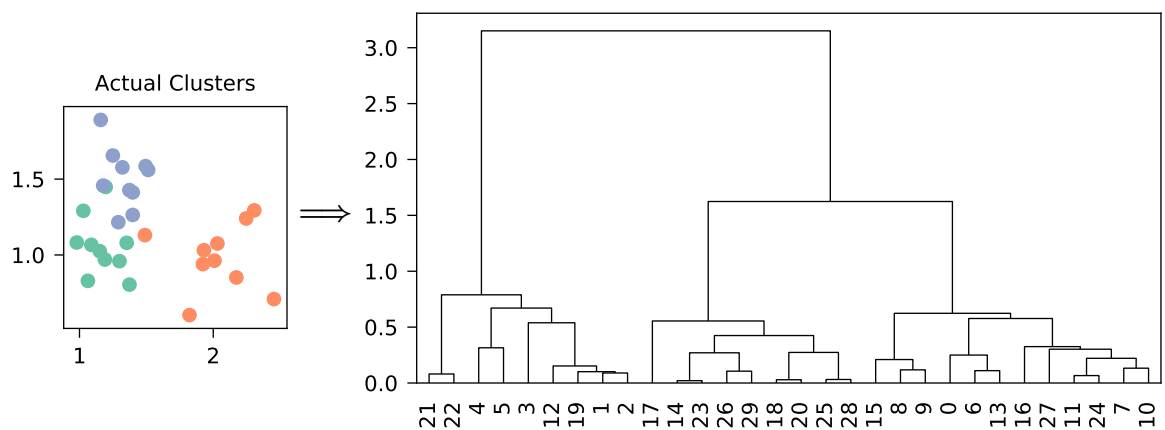
- *Ward's method* [\[89\]](#): This criterion determines the variance between all sample pairs of two clusters and is thus comparable to the objective of k-means clustering. It calculates the cluster distance D between (potentially differently sized) clusters U and V , where cluster U was previously constructed from subclusters S and T as defined in [Equation 2.25](#):

$$D(U, V) = D(S \cup T, V) = \sqrt{\frac{|V| + |S|}{N} D(V, S)^2 + \frac{|V| + |T|}{N} D(V, T)^2 - \frac{|V|}{N} D(S, T)^2} \quad (2.25)$$

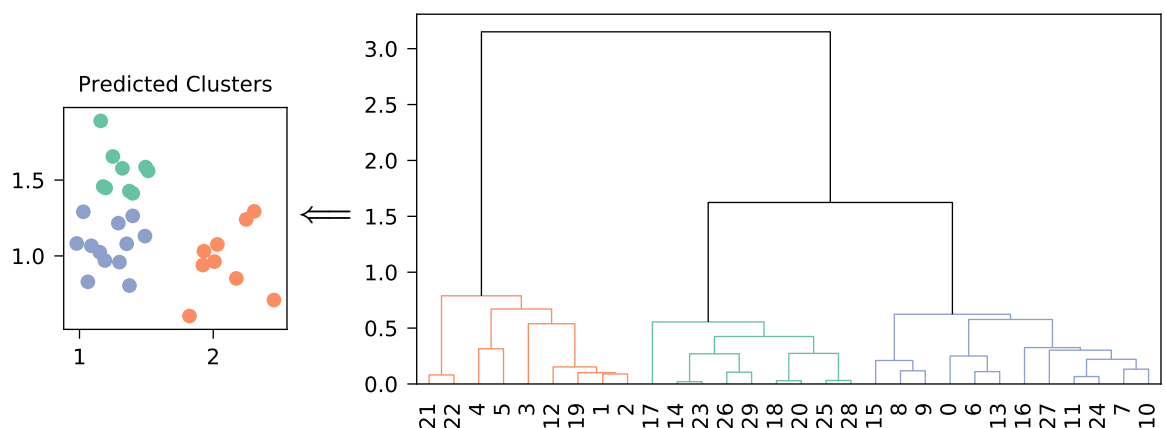
where $|*|$ represents the cluster's cardinality and $N = |V| + |S| + |T|$. Analogously to the weighted average linkage from above, the initial distances between the individual samples (single-sample clusters) must first be calculated based on some specified vector distance d , i.e., $D(U', V') = d(\mathbf{u}, \mathbf{v}) \forall U' = \{\mathbf{u}\}, V' = \{\mathbf{v}\}$.

After the entire merging process, the final result is a linkage matrix which stores all the calculated distances, i.e., the entire agglomerative clustering hierarchy. Such a matrix can then be visualized with a *dendrogram* [\[126\]](#), which uses a tree to display the distances between clusters and how they are linked to their subclusters (the cluster hierarchy). Each level in the tree represents exactly one merge step, which means that the tree has a depth of $n - 1$ since the n samples were merged $n - 1$ times in order to get the final root cluster with all samples. This has the advantage that the number of clusters k does not need to be specified in advance but can be determined based on the dendrogram. Choosing some k then simply means cutting the tree at depth $k - 1$, which yields k clusters. [Figure 2.13](#) shows an example of a dataset of 30 samples that form three clusters. Running the agglomerative clustering

yields the linkage matrix which is visualized in [Figure 2.13a](#). The x-axis shows the individual samples of the dataset, the y-axis represents the distance between clusters. In the dendrogram, we can see that there are three dense areas/branches: the left one from sample 21 to sample 2, the middle one from sample 17 to sample 28 and the right one from sample 15 to sample 10. This is a good indication where to cut the tree and thus extract the clusters. Therefore, we cut the tree at depth two, i.e., just at the level of our three main branches, which results in three clusters. This cut and the resulting clusters are visualized in the colored dendrogram in [Figure 2.13b](#), where we additionally show the final clustering result in the original data space (left figure). Especially for large datasets, it can be useful not to display every single sample in the dendrogram but to merge clusters based on the closest distance until only a specified number of tree branches remain. [Figure 2.14](#) shows a truncated dendrogram of the above example where the branch limit was set to ten. Plain numbers still represent single samples (e.g., the third branch from the left indicates sample 3), numbers within brackets (x), on the other hand, indicate merged clusters of size x (e.g., the last branch on the right (6) represents a cluster with six samples).



(a) The 30 samples forming three clusters (left) and the dendrogram (right) that was created based on the linkage matrix from agglomerative clustering with Euclidean distance and Ward's method.



(b) The same dendrogram (right) but with the tree cut at depth two, which results in clustering the samples into three groups that are shown for the original samples (left)

Figure 2.13: Example of agglomerative clustering and the dendrogram visualization based on a dataset of 30 samples and three clusters.

For hierarchical clustering, there are also some points to consider. First, for large datasets, the linkage matrix becomes rather large and may pose some memory problems. Second, new samples cannot simply be assigned to existing clusters since hierarchical clustering, unlike

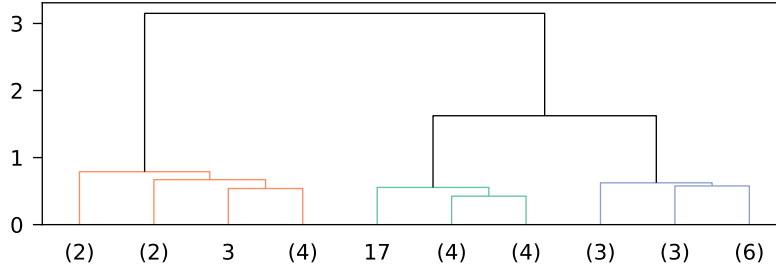


Figure 2.14: A truncated version of the dendrogram in [Figure 2.13b](#).

k-means, is not a generative approach (cf. [Section 2.4.3](#)), and thus, the entire hierarchy must be rebuilt. Moreover, the identified clusters largely depend on the chosen linkage criterion. For example, single linkage tends to create uneven cluster sizes, whereas Ward’s method leads to more evenly sized clusters. In [Figure 2.12](#), we again show how agglomerative clustering performs on example datasets. The actual clusters are shown in [Figure 2.12a](#), whereas the predicted clusters using agglomerative clustering with Euclidean distance for d and Ward’s method for D are presented in [Figure 2.12c](#).

2.4.4.5 Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH)

BIRCH [\[206\]](#) was specifically designed for large datasets. A clustering feature (CF) tree is built incrementally from the input samples, and it is height-balanced using a specifiable branching factor b and threshold t . The nodes of the tree do not contain the original samples but only aggregations thereof, so-called CF subclusters, which drastically reduces resource consumption (time and memory). A CF subcluster only stores its sample count m and both the linear sum (vector) $\mathbf{l} = \sum_{i=1}^m \mathbf{x}_i$ as well as the squared sum (scalar) $s = \sum_{i=1}^m \|\mathbf{x}_i\|^2$ of the subcluster samples. Given these aggregated measures, the subcluster centroid \mathbf{c} and its radius r can be calculated as defined in [Equation 2.26](#):

$$\mathbf{c} = \frac{\sum_{i=1}^m \mathbf{x}_i}{m} = \frac{\mathbf{l}}{m} \quad \text{and} \quad r = \sqrt{\frac{\sum_{i=1}^m (\mathbf{x}_i - \mathbf{c})^2}{m}} = \sqrt{\frac{s}{m} - \left(\frac{\mathbf{l}}{m}\right)^2} \quad (2.26)$$

Each node can hold multiple CF subclusters, which is bound by the branching factor b . In case it is not a leaf node, the subclusters can also contain a link to another node further down in the tree hierarchy. Starting with the root node, a new sample is tried to be fit into the subcluster that has the smallest radius r after adding the sample, or a new subcluster is created if the threshold t is exceeded. If the selected subcluster has any child nodes, then this process is repeated until a leaf is reached. Afterwards, the stored data (m, \mathbf{l}, s) is recursively updated. In case the branching factor b is reached, i.e., the current node already has the maximum number of subclusters, then this node is split into two child nodes, where all subclusters of the node must be distributed again. The current node’s parent subcluster (the aggregation of all the node’s subclusters) is then replaced with the two aggregated subclusters of the two new nodes. The final cluster of a sample corresponds to the CF subcluster leaf to which it is assigned. This means that BIRCH automatically identifies the number of clusters, which is the number of leaves in the tree. Alternatively, another clustering algorithm such as agglomerative clustering can be applied on the CF subcluster leaves to cluster them into k groups, where k must be less than or equal to the number of leaves. In [Figure 2.12d](#), we also present how BIRCH performs on the example datasets compared to the actual clusters shown in [Figure 2.12a](#).

2.5 Statistical Background

In this section, we establish a common ground for the terms standardization and normalization, and then we present statistical correlation and test methods used in this thesis.

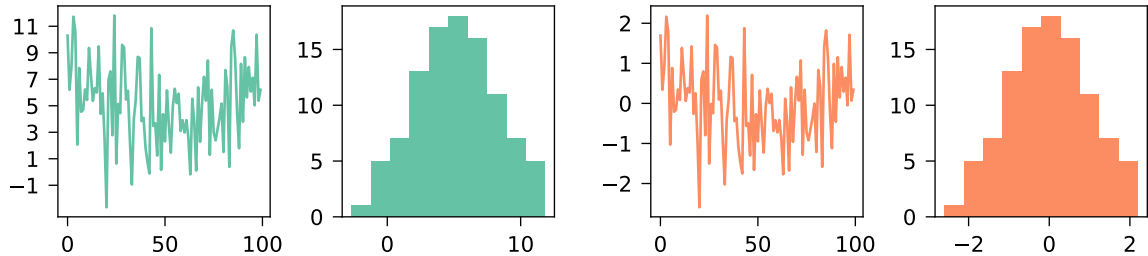
2.5.1 Standardization

Standardization means scaling values to have an arithmetic mean of zero and a standard deviation/variance of one (zero mean, unit variance). More formally, given a set of values $X = \{x_1, \dots, x_n \mid x_i \in \mathbb{R}\}$, its arithmetic mean $\mu = \frac{1}{n} \sum_{i=1}^n x_i$ and its standard deviation $\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2}$, the standardized set X_s is calculated as defined in [Equation 2.27](#):

$$X_s = \left\{ \frac{x_i - \mu}{\sigma} \mid \forall x_i \in X \right\} \quad (2.27)$$

If all values in X are equal, then $\sigma = 0$. In this case, only the mean μ is subtracted, which yields a standardized set of values X_s that only contains zeros (since $\mu = x_i \forall x_i \in X$).

[Figure 2.15](#) visualizes an example dataset of 100 values and the effect of standardization. All values were drawn from a normal distribution with a mean of five and a standard deviation of three, formally given by $\mathcal{N}(\mu, \sigma^2) = \mathcal{N}(5, 3^2)$. In [Figure 2.15a](#), the original, unaltered data and its histogram are shown. In [Figure 2.15b](#), the standardized data is shown with the new mean of zero and unit variance as if sampled from the normal distribution given by $\mathcal{N}(0, 1)$. This example demonstrates that the mean and standard deviation of the data change, but the proportions and relative distances of the individual data points are not affected, i.e., the original distribution is not changed, which is expected since standardization is a linear transformation.



(a) Data and histogram before standardization. (b) Data and histogram after standardization.

Figure 2.15: Example dataset and its histogram in the original scale (left) and after standardization (right).

2.5.2 Normalization

In this thesis, *normalization* is the procedure to scale values to the range $[0, 1]$. More formally, given a set of values $X = \{x_1, \dots, x_n \mid x_i \in \mathbb{R}\}$, its minimum value $\min(X)$ and its maximum value $\max(X)$, the normalized set X_n is calculated as defined in [Equation 2.28](#):

$$X_n = \left\{ \frac{x_i - \min(X)}{\max(X) - \min(X)} \mid \forall x_i \in X \right\} \quad (2.28)$$

If all values in X are equal, then $\max(X) - \min(X) = 0$. In this case, only the minimum $\min(X)$ is subtracted, which yields a normalized set of values X_n that only contains zeros (since $\min(X) = x_i \forall x_i \in X$).

[Figure 2.16](#) shows the same example as [Figure 2.15](#), i.e., 100 values randomly sampled from $\mathcal{N}(5, 3^2)$, but now, normalization is applied. Since normalization is also a linear transformation, the original distribution is not changed, as can be clearly seen in the histograms shown in [Figure 2.16a](#) and [Figure 2.16b](#).

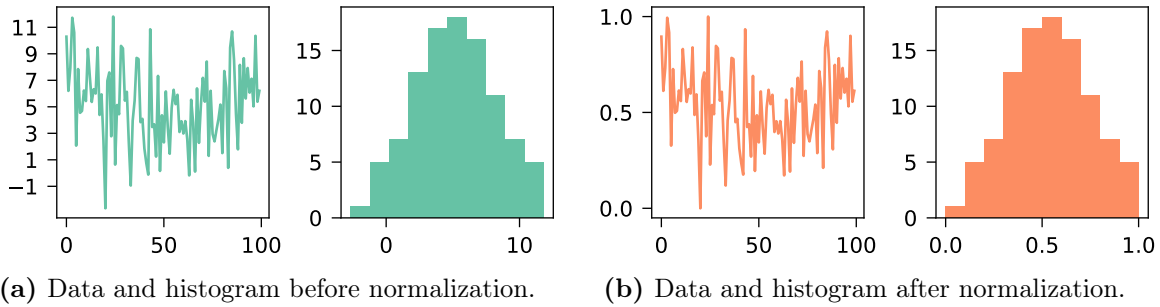


Figure 2.16: Example dataset and its histogram in the original scale (left) and after normalization (right).

2.5.3 Pearson Correlation

We can check whether two variables X and Y correlate with each other using different correlation measures that typically yield values close to $+1$ for a strong positive correlation, values close to -1 for a strong negative correlation and values around 0 for no correlation. One of the most widely used measures is the Pearson correlation coefficient $\rho_{X,Y}$, which calculates the linear correlation based on the covariance $\text{cov}(X, Y) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu_X) \cdot (y_i - \mu_Y)$ and standard deviations σ_X and σ_Y as shown in [Equation 2.29](#):

$$\rho_{X,Y} = \frac{\text{cov}(X, Y)}{\sigma_X \cdot \sigma_Y} \quad (2.29)$$

A visual representation of examples best demonstrates how various X and Y yield different correlation coefficients. [Figure 2.17](#) displays various datasets of two-dimensional points with coordinates represented by X and Y . In the first data row, we can see that the Pearson correlation coefficient takes on values close to ± 1 the stronger the data points resemble a straight line, either with X and Y in the same direction (positive correlation) or in the opposite direction (negative correlation). The second data row clearly shows that the correlation does not represent the slope of the direction (the middle dataset is undefined because $\sigma_Y = 0$ and thus [Equation 2.29](#) cannot be computed). Naturally, since the Pearson correlation is based on linear correlation, it cannot detect non-linear relationships, as can be seen in the third data row, where all datasets have a correlation of 0 (= no correlation).

2.5.4 Wilcoxon Signed-Rank Test

The Wilcoxon signed-rank test [\[189\]](#) is a statistical hypothesis test used for paired/matched samples (one sample a_i of the first dataset A must have a corresponding sample b_i in the second dataset B) to check whether two datasets come from the same distribution (null hypothesis) or not (alternative hypothesis). Internally, the differences $(a_i - b_i)$ of all data pairs

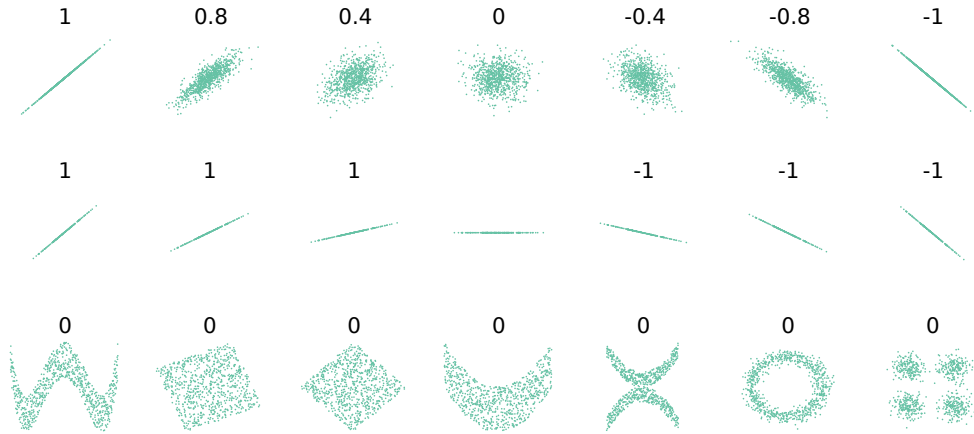


Figure 2.17: Example datasets and their Pearson correlation coefficients, adapted from [37].

are calculated and ranked according to their absolute value, which is ultimately used to check whether the median of all differences is around zero. Thereby, differences that are exactly zero are discarded in the ranking process per default. However, Pratt's [137] method can be used as an alternative that incorporates these zero-differences, which is useful when dealing with ordinal data where zero-differences are not unlikely. For checking if the null hypothesis must be rejected or, alternatively, cannot be rejected, the p-value is calculated (probability based on the calculated test statistic). The smaller the p-value is, the less confident we are that the null hypothesis is true. This confidence is expressed via the significance level α , with typical values of 0.1, 0.05 or 0.01 (lower values mean stricter significance thresholds, or, alternatively, higher confidence levels since the confidence level is $1 - \alpha$). If the p-value is equal to or smaller than the chosen α , we are confident enough and must reject the null hypothesis, which means that the median of the differences is not around zero, i.e., the differences of the two datasets are statistically significantly different. Conversely, if the p-value is larger than α , we are not confident enough and thus cannot reject the null hypothesis, meaning that the median of the differences appears to be around zero, i.e., the differences of the two datasets are not statistically significantly different. Since we check differences in both directions, this is referred to as the two-sided version.

There also exist two one-sided variations of the hypothesis test. One is checking whether the median of the differences is equal to zero or positive (null hypothesis) or negative (alternative hypothesis), and the other is checking whether the median of the differences is equal to zero or negative (null hypothesis) or positive (alternative hypothesis). This can be used to confirm that one dataset is statistically significantly smaller or larger than the other by rejecting the corresponding null hypothesis in favor of its alternative. Since we now only check differences in a single direction compared to the two-sided version above, we must adapt the significance level accordingly to $\frac{\alpha}{2}$.

2.5.5 Box Plots

Box plots [116] are used to visualize the distribution and important characteristics of a set of values. Since there are slight variations possible, we briefly introduce the box plots that we use throughout this thesis. As an example, we randomly sample 50 values from $\mathcal{N}(10, 1)$ and manually replace two of these values with 4 and 5. Figure 2.18 visualizes the resulting dataset with a box plot. The box ranges from the first quartile (25% percentile) to the third quartile (75% percentile), the so-called interquartile range (IQR). The line in the middle indicates the median (50% percentile). Left and right to the box are the whiskers, which represent

values outside the IQR. They extend to the lowest and highest data point but are limited to a maximum of $1.5 \cdot \text{IQR}$. All values that surpass this threshold are considered outliers and are visualized with diamond-shaped symbols as shown in the example for values 4 and 5. For more details, [Table 2.5](#) provides an overview of various statistics.

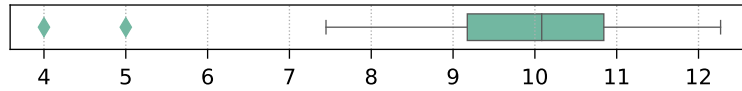


Figure 2.18: Example of a box plot with lower outlier values (cf. [Table 2.5](#) for details).

μ	σ	min	p_{10}	p_{25}	p_{50}	p_{75}	p_{90}	max
9.88	1.57	4.00	8.38	9.17	10.08	10.84	11.50	12.27

Table 2.5: Various statistics of the example data. μ = average, σ = standard deviation, p_i = $i\%$ percentile, min = minimum, max = maximum.

Chapter 3

Topology-driven Crash Analysis

In this chapter, we present our first approach on tackling problems within the multi-system environment. The goal of our process crash analysis, which is based on the topologies and crash events of the various systems, is to identify potentially problematic software technologies that may lead to crashes across multiple systems. The following sections cover all details of this approach, an evaluation on real-world data, and problems and limitations we faced throughout our work. Major parts of this chapter were published in [157].

3.1 Motivation

Faults leading to crashes are common in software systems, especially in large-scale systems due to their high complexity [32, 127]. This can result in further failures, performance degradations or even entire outages, which can have an economical impact as well as damage the reputation of the service provider [32]. As systems tend to grow even further and more and more data about their behavior is collected, manually inspecting all the crashes is unfeasible and automated approaches are necessary, which has caught the interest of many researchers [202, 205, 181]. Since the number of crashes can become large, prioritizing them is essential to help developers find the most important or urgent ones. Typical techniques for prioritization are bucketing and grouping similar crashes, which can then be used, for example, to focus on the groups with the highest number of crash occurrences. All this is an active field of research, and many approaches have already been developed, which include the inspection of crashes within single software systems or applications (e.g., [124]), specific products or product families (e.g., Microsoft Windows and Office suite [68], or products from Mozilla [67]), or certain software ecosystems such as Android [65]. However, investigating crashes that occur across different, independent software systems has not yet been a focus in related work. If we identified common cross-system crash causes, we could develop a fix and potentially apply it to all affected systems, rather than having to find the same root cause multiple times and fixing every system individually.

Analyzing crashes in a multi-system environment comes with additional challenges. First, the technological landscape is significantly larger and much more diverse than in a single system because various technologies are deployed in different versions and interact with a wide range of components. Second, the multi-system aspect must be taken into consideration when filtering and prioritizing the individual crashes, since we are especially interested in those crashes that not only occur frequently but also across multiple systems. To this end, we developed an approach that utilizes the topological information of systems and creates so-called *software technology tuples* based on the technologies of the systems' processes. A tuple can either store

a single (1-tuple) or multiple technologies (n -tuple) indicating the communication between n processes. For instance, the 2-tuple (Java, .NET) represents two connected processes, one executing a Java-based application and the other running within a .NET environment. We check for every tuple how often the associated process(es) crashed and merge them across all systems. Using a specifiable ranking metric, the tuples are then sorted and the top-ranked ones are selected for further inspection, where groups of common crash properties (e.g., the error or exception message) are created. Finally, these resulting crash groups can be analyzed by an engineer as a starting point for identifying the root cause of the problem, where a fix can potentially be applied to all systems where these crashes happened.

3.2 Data Requirements and Assumptions

Our evaluation is based on the data we collected from our industry partner, however, the general approach is independent of this particular infrastructure. In the following, we list all required data and all our assumptions that must hold in order to apply our approach in other settings as well.

3.2.1 Topology

The topology of a system must at least contain data of all processes, including lifetime properties (the start and end timestamps) and, for n -tuples with $n > 1$, information on the process communication. For the latter, it suffices to know whether two processes communicate with each other or not, the tuple extraction is then simply based on their (overlapping) lifetimes. Additionally, every process must have a list of software technologies attached that were active throughout its lifetime. Every software technology entry must store the type (e.g., Java, Tomcat, etc.) and the timestamp until it was active (the end timestamp¹), and it can have optional properties such as the version.

[Figure 3.1](#) shows a small, abstract example system with three connected processes ($P1$ is linked with $P2$, which is linked to $P3$, but $P1$ and $P3$ are not connected), each with data on their lifetimes as well as the list of software technologies. From the timeline, we can see one additional assumption of our approach. Since we only have a single timestamp for the technologies, which are the end timestamps, we cannot know when they started. We thus assume that every attached technology is already active when the corresponding process starts. For instance, A and B both start together with process $P1$ at timestamp 100. Naturally, this might not actually always be the case, the effects of which we discuss in [Section 3.6.3](#).

3.2.2 Events

Events are process crashes² that happened in a system throughout the monitored period of time. Every event must include the time of the occurrence, the corresponding process entity and at least one crash property (e.g., exception, stack trace, crash location).

¹Ideally, there would also be a start timestamp available, which indicates when a technology became active. However, since the data that is provided by our industry partner and that we are going to use in the evaluation unfortunately does not contain this information, our approach must be capable of handling this limited data as well, which is the reason we only list the end timestamp of a technology as a requirement.

²They do not necessarily need to be process crashes. In fact, our approach can handle arbitrary events with the single restriction that they must occur on process entities. However, it must make sense to investigate such events in combination with the software technologies, which is certainly the case for crashes.

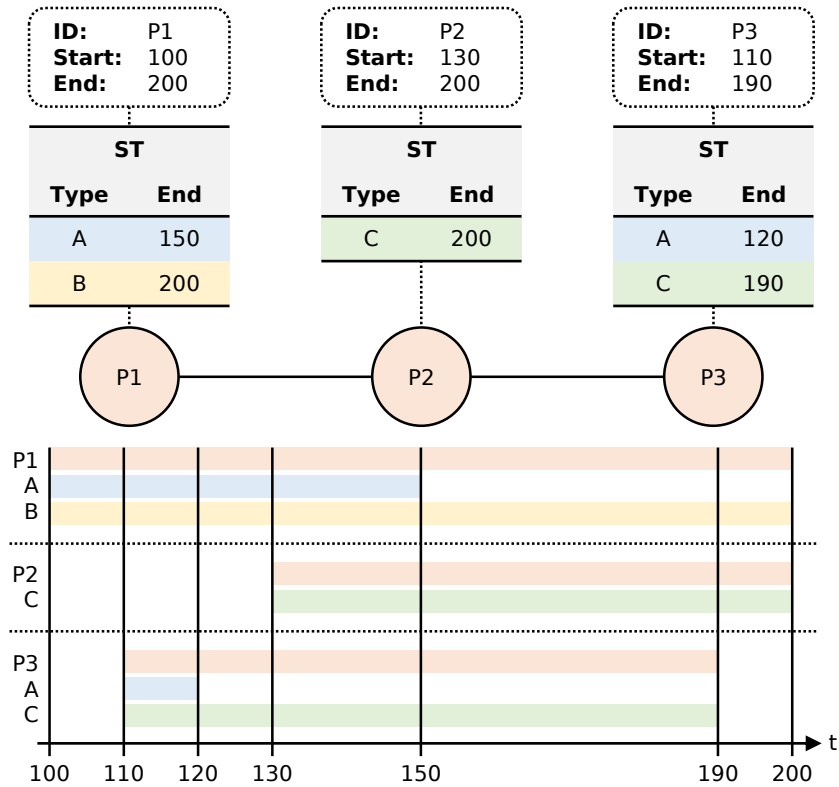


Figure 3.1: Example system with three processes and their software technologies (ST). The timeline shows the start and end timestamps between which the various components were active.

In [Table 3.1](#), we show a few crash event examples of the processes in [Figure 3.1](#) and with the exception as the selected crash property. This could be typical crash exceptions if process $P1$ was Java-based and $P2$ a process within the Microsoft .NET environment. In both cases, the processes were restarted, which explains why their end timestamps are independent of the crashes. The event timestamps can be used to see which software technologies were active when the crash occurred. For instance, at the time of event $E2$, only technology B was running on process $P1$, whereas for event $E1$, both A and B were active.³

Event	Process	Property: Exception	Timestamp
$E1$	$P1$	java.lang.NullPointerException	105
$E2$	$P1$	java.lang.IllegalArgumentException	167
$E3$	$P2$	System.TypeInitializationException	192

Table 3.1: Example crashes with the exception property.

3.3 Approach

Now that we established the data requirements, we continue with the detailed description of our approach. The main idea is to create so-called software technology tuples and link them with process crashes to see which technologies seem more error-prone. After having identified

³According to our assumption that all technologies are active when the corresponding process starts.

problematic tuples, we use any of the crash properties to investigate them more closely, which can help us in identifying the root cause. Before the tuple creation, we have to appropriately process the topologies of all the systems we want to analyze. We do this by creating snapshots that capture the changes and states within a system, i.e., we know exactly which processes and which software technologies were active at a certain point in time, and which processes communicate with each other.

3.3.1 Tuple Creation

Tuples are the core concept in our approach, everything that follows builds on them. A tuple stores the software technologies, and then we count how often a process actively using these technologies crashed and how often it did not crash. For simplicity reasons and since we do not know the data flow within the processes anyway, we decided to drop the exact time information⁴ and reduce it to these two single metrics (crashed, not crashed). In addition to these crash counts, we also record the systems in which the crashes occurred. The goal is to extract information on whether a tuple can be considered more error-prone than others, especially if multiple systems are involved, and is thus of interest to engineers who investigate the crashes. Our approach supports the following types of tuples:

- *1-tuples*: These tuples only capture a software technology of a single process, which means that the process communication information in the analyzed data is optional in this case. They are primarily designed to investigate bugs within a particular software technology. For every system, process and each technology X , a tuple is created in the form (X) . For example, for the system in [Figure 3.1](#) we would create five 1-tuples: (A) and (B) of process $P1$, (C) from process $P2$, and (A) and (C) from process $P3$ (two tuples we have already seen in the other processes).
- *2-tuples*: These tuples store the software technologies of two connected processes. They are primarily designed to investigate if there are software incompatibilities leading to failures, perhaps due to a wrong system configuration [\[198\]](#). For instance, if there are mismatching technologies (e.g., different, incompatible versions) running on two communicating processes, we can capture such crash incidents with 2-tuples, whereas 1-tuples could not express this particular issue. For every system, we extract all process pairs and then create active technology pairs in the form (X, Y) , where X is the technology of the first process, Y the one of the second process and active means that the execution time of both X and Y must be overlapping (the duration is irrelevant). For example, for the system in [Figure 3.1](#), we would create three 2-tuples: (A, C) and (B, C) of processes $P1$ and $P2$, and (C, C) from processes $P2$ and $P3$. Although $P1$ and $P3$ have overlapping lifetimes and thus overlapping, active software technologies, no tuples are created because the two processes do not communicate with each other. We also do not create the tuple (C, A) for processes $P2$ and $P3$ since technology A from process $P3$ is not active any more when $P2$ (more specifically, technology C from $P2$) starts, i.e., the technologies do not overlap.
- *n -tuples*: These tuples store the software technologies of multiple ($n > 2$) connected processes. Since the direct process communication is already covered by 2-tuples, we regard n -tuples as a more theoretical application, but we will discuss this topic later in [Section 3.6](#). They are analogous to the above case where $n = 2$, i.e., the same conditions must apply: The technologies must be active and the processes must be linked (no full

⁴We only know the type of software technology, the rest is a black box, as we do not know which inner parts are active at what time, so incorporating the exact timestamps of the crashes would not make sense.

connection required, a “chain” of communicating processes suffices). For the system in [Figure 3.1](#) and $n = 3$, we would create two 3-tuples: (A, C, C) and (B, C, C) of processes $P1$, $P2$ and $P3$.

For every created tuple, we then check whether it crashed somewhere during its lifetime (the time period where all software technologies are active) or “survived” the entire time. A tuple crashed if one or more crash events occurred on one or more processes that run the tuple’s software technologies. For every system we parse, we get a set of annotated tuples T , where each tuple $t \in T$ contains the information as specified in [Table 3.2](#).

Data	Description	Example
$t.techs$	The software technology tuple.	(A, C) of processes $P1$ and $P2$.
$t.crashed$	1 if at least one of the processes running the $techs$ crashed and this crash occurred during the time the $techs$ were active, 0 otherwise.	1 because process $P1$ (running technology A) crashed at time 105 (event $E1$).
$t.events$	The set of all crash events that occurred on the processes or an empty set if no crash happened.	{ $E1$ }
$t.system$	The identifier of the monitored system where the processes are running.	<i>DemoSystem</i>

Table 3.2: Data for an annotated tuple $t \in T$ after the tuple creation process. The examples are based on the system and events in [Figure 3.1](#) and [Table 3.1](#).

3.3.2 Tuple Merging

The next step in our approach is to merge the set of annotated tuples T within each system. Thus, for each system, we create a set of groups \mathbf{G}_T , where each group contains the tuples t whose software technologies $t.techs$ are equal. More formally, $\mathbf{G}_T = \{G_1, \dots, G_k \mid G_i \subseteq T \wedge \forall i \neq j : G_i \cap G_j = \emptyset \wedge \forall t_x, t_y \in G_i : t_x.techs \equiv t_y.techs\}$, where k is the number of groups, i.e., the number of unique technologies within the current system, and $\bigcup_{G \in \mathbf{G}_T} G = T$. We consider two technologies equal if their properties are equal, which means that the mandatory types (e.g., Java) but also all optional data must be the same. If the version information is available, we treat this as a special case because we can optionally apply a version compaction: Often, version numbers can be rather long and users might want to ignore irrelevant version tokens. Our approach enables users to do so by specifying how many version tokens should at least be kept and how many should be discarded, starting from the back. Our default setting is to drop the last token but keeping at least two in the front. For instance, if the version is 1.8 (two tokens), it remains unchanged. If the version is something like 1.8.15.127 (four tokens), it is compacted to 1.8.15 (last token is removed). Of course, this is an entirely optional step, so users may choose to always consider the entire version number. For n -tuples with $n \geq 2$, equality is determined based on whether the tuple is exactly identical or “reverse-equal”, more formally, $(x_1, \dots, x_n) \equiv (x_n, \dots, x_1)$, where x_i is a single software technology with all its (optional) properties as described above. As an example from [Figure 3.1](#), tuples (A, C, C) and (C, C, A) would be considered equal but not (C, A, C). Ultimately, we end up with our k equality groups \mathbf{G}_T , where all equal annotated tuples of T are assigned to a separate group $G \in \mathbf{G}_T$. For each system s , we now merge these annotated tuples in every group, which results in a set of merged tuples M_s , where each tuple $m \in M_s$ contains the information as specified in [Table 3.3](#). Every tuple $m \in M_s$ contains information on how often it crashed

($m.crashed$) and how often it survived ($m.\neg crashed$) throughout the observation period of the corresponding system.

Data	Description
$m.techs$	$t.techs$, arbitrary $t \in G$
$m.crashed$	$\sum_{t \in G} t.crashed$
$m.\neg crashed$	$\sum_{t \in G} (1 - t.crashed)$
$m.events$	$\bigcup_{t \in G} t.events$
$m.system$	$t.system$, arbitrary $t \in G$

Table 3.3: Data for a merged tuple $m \in M_s$ and group $G \in \mathbf{G}_T$ after the tuple merging process. G contains annotated tuples t that are considered equal. For $t.techs$ and $t.system$, we can use an arbitrary t since all software technologies are identical within G and we are still only inspecting a single system s .

The final step is to combine the data of all systems. Assume that we have a set of systems S , and each system $s \in S$ is assigned its corresponding set of merged tuples M_s . We first create the superset $M = \bigcup_{s \in S} M_s$. Similarly to before, we then create a set of groups \mathbf{G}_M , where each group contains equal merged tuples m , but now across all different systems. More formally, $\mathbf{G}_M = \{G_1, \dots, G_l \mid G_i \subseteq M \wedge \forall i \neq j : G_i \cap G_j = \emptyset \wedge \forall m_x, m_y \in G_i : m_x.techs \equiv m_y.techs\}$, where l is the number of groups, i.e., the number of unique technologies across all systems, and $\bigcup_{G \in \mathbf{G}_M} G = M$. Equality is defined precisely as above, which results in a set of final tuples F that covers all systems, where each tuple $f \in F$ contains the information as specified in [Table 3.4](#). The new value $f.crashedSystems$ represents the set of systems in which the tuple crashed, whereas $f.\neg crashedSystems$ indicates the set of systems where the tuple did not crash. Note that both these sets may overlap. This happens, for example, if an annotated tuple t_1 crashed on some process of system s but another annotated tuple t_2 with equal software technologies ($t_1.techs \equiv t_2.techs$) did not crash on some other process of the same system. In the merging process, $m.crashed$ would then be 1 and $m.\neg crashed$ would be 1 as well, which would ultimately result in the final tuple f with $f.crashedSystems = \{s\}$ and $f.\neg crashedSystems = \{s\}$.

3.3.3 Ranking

After having computed the set of final tuples F , we want to extract tuples that are ‘‘interesting’’. We accomplish this by a user-specifiable ranking metric (higher is better/more interesting) that sorts the tuples based on the data features stored in every f (cf. [Table 3.4](#)). By default, we characterize interesting tuples as those that crashed often and in many different systems, since we are especially focusing on investigating cross-system crashes, where identifying a common bug and finding a fix can potentially bring the most benefits. The default ranking metric r for a final tuple f is formally defined in [Equation 3.1](#):

$$r(f) = \frac{|f.cS|}{|f.cS| + |f.\neg cS|} \cdot |f.cS| \cdot f.crashed \quad (3.1)$$

where $|*|$ represents the set’s cardinality and cS is short for $crashedSystems$. This ranking metric consists of three main factors. The first factor rewards a tuple that crashed in many different systems compared to the number of systems where it did not crash. It yields values

Data	Description
$f.techs$	$m.techs$, arbitrary $m \in G$
$f.crashed$	$\sum_{m \in G} m.crashed$
$f.\neg crashed$	$\sum_{m \in G} m.\neg crashed$
$f.events$	$\bigcup_{m \in G} m.events$
$f.crashedSystems$	$\bigcup_{m \in G \wedge m.crashed > 0} \{m.system\}$
$f.\neg crashedSystems$	$\bigcup_{m \in G \wedge m.\neg crashed > 0} \{m.system\}$

Table 3.4: Data for a final tuple $f \in F$ and group $G \in \mathbf{G}_M$ after the final cross-system merging process. G contains merged tuples m that are considered equal across all systems. For $m.techs$, we can use an arbitrary m since all software technologies are identical within G .

in the range $[0, 1]$, where 0 means that the tuple did not crash at all (no systems exist where the tuple crashed) and 1 means that the tuple crashed in every possible scenario, i.e., every time the tuple was observed, a crash occurred (no systems exist where the tuple survived). The second factor represents the number of crashed systems, and it is necessary to balance the previous one, since otherwise, tuples with $|f.\neg crashedSystems| = 0$ would always have a maximum score of 1 despite having a potentially very low absolute number of crashed systems.⁵ Lastly, the third factor is simply a scaling factor that rewards tuples with a high absolute number of crashes, so reoccurring crashes are ranked higher than those that happen rarely.

3.3.4 Crash Property Analysis

After ranking the final tuples in F , we sort them in descending order and select the top-ranked ones (how many is up to the user). The final step of our approach is then to analyze a specified crash property for the crashes within such a top-ranked tuple. This is done by first grouping the crash events according to the selected property (e.g., the exception), i.e., we form groups of equal property values. Within in each group, we then apply yet another grouping based on the systems where the (grouped) crashes occurred. Finally, the results are sorted and visualized in a bar plot, where the y-axis indicates the crash property values in descending order of occurred crashes, and the x-axis shows the absolute number of crashes. If a property is missing (e.g., the data is incomplete or erroneous), then the grouping algorithm simply treats these cases as “missing”. This way, we do not know the actual value, but we can still see the entries in the plot, so we do not accidentally overlook them. The individual bars of the plot can be multi-colored, where each color represents a different system in which the crashes happened. For us, the most promising visual cues are heavily multi-colored bars with a high number of crashes, which is exactly what we defined as interesting above. Naturally, the plots depend on the selected crash property of the top-ranked tuples, which, in turn, depend on the chosen ranking metric.

Figure 3.2 shows a small example of such a bar plot. The title indicates which tuple was analyzed and which rank it was assigned given our ranking metric. Note that this rank is *not* equal to value obtained via the ranking metric r . We only use that value for sorting to get

⁵The extreme case would be $f.crashedSystems = 1$, i.e., the tuple occurred only in a single system, which would lead to $\frac{|f.cS|}{|f.cS|+|f.\neg cS|} = \frac{1}{1+0} = 1$. Of course, this is not what we consider to be interesting.

the most interesting tuples. After this sorting, we replace the actual ranking metric value with the tuple’s position, which we call its rank. For instance, if there are three tuples f_1 , f_2 and f_3 and calculating their ranking metric value (cf. [Equation 3.1](#)) yielded $r(f_1) = 0.71$, $r(f_2) = 0.02$ and $r(f_3) = 0.34$, we first sort these values in descending order (highest ranking metric value is the best), which results in the list $(f_1 : 0.71, f_3 : 0.34, f_2 : 0.02)$. Afterwards, the ranking metric values are replaced with their corresponding position in this list (the rank), i.e., $(f_1 : 1, f_3 : 2, f_2 : 3)$, which means that tuple f_1 is the most important one and f_2 the least important one. The rank is thus a lower-is-better value in the range $[1, n]$, where n is the number of tuples and 1 means the highest/best possible rank. In this case, the results for the 1-tuple $f_1 = (|Tomcat, 8.0.44|)$ ⁶ are shown, which happened to be the 1st-ranked tuple. We opted for investigating the exception crash property, and the plot reveals that three exceptions were thrown, where the `java.lang.NullPointerException` occurred most often (10 crashes). The colors represent the different systems in which the crashes happened. In the example, three systems were affected by `java.lang.NullPointerException` and `java.lang.IllegalArgumentException` crashes, and in two systems, a `java.lang.OutOfMemoryError` led to a crash.

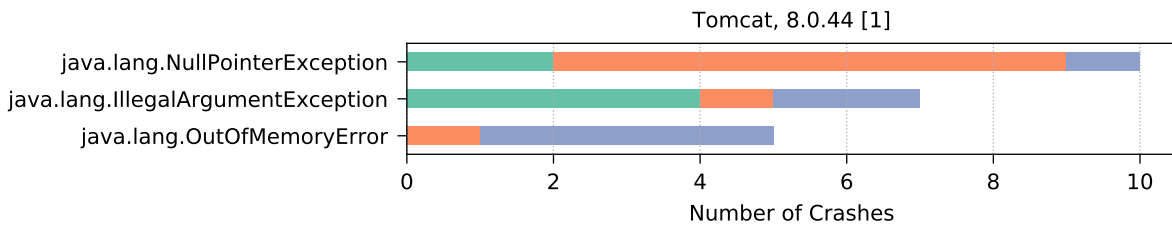


Figure 3.2: Example crash property bar plot.

Of course, the plots alone do not automatically provide an explanation for the various root causes of the problems, which was not the goal of our approach in the first place. However, they serve as a starting point for further manual investigation by showing prioritized tuples or tuples of interest. An engineer can then actively search for causes and perhaps identify a common issue and a corresponding fix, which can potentially be applied in all affected systems.

3.4 Data for Evaluation

We already presented how the data must look like in order for our approach to be applicable. Now, we introduce the dataset we use for the evaluation. The data originates from software systems of customers of Dynatrace, and each system is completely independent of each other and may operate in arbitrary domains. The structure is exactly as described in [Section 2.3 on p. 7](#). However, we only need a few parts of the collected data, which we briefly summarize in the following:

- **Topology:** Only the process components are required. They store every information needed, which includes all relevant properties such as lifetimes, connections (for process communication) and their software technologies. The technologies also fulfill all data requirements, and we additionally have the optional edition and version information at our disposal (cf. [Table 2.1 on p. 9](#)).

⁶The enclosing `|` characters are only there to avoid confusion with n -tuples, i.e., to make it clear that everything within `|...|` is a single tuple element but with potentially multiple internal entries, such as the type and version in this example case.

- Events: Only the process crash events are relevant, which also cover every information that our approach requires. The following crash properties are available: the fault location (location within the executable or the linked library), the fault module path (of the executable or the linked library), the fault module version (of the executable or the linked library), the executable path (same as the module path in case the crash occurred in the executable and not in a linked library), the class name of the exception/error (.NET-based processes) or the class where the exception occurred (Java-based processes), the assembly name (.NET-specific units), the exception message and the process signal (e.g., “aborted”, “segmentation fault”).

For the evaluation, we gathered monitoring data spanning over 15 months, so changes in the topology such as different versions of software technologies are more likely to be included than when inspecting a shorter observation period. To avoid an increased load on the monitored systems and to keep the data size manageable, we decided to export the first week of every month. More specifically, as listed in [Table 3.5](#) (cf. [Figure 3.3](#) for a visual representation), we exported data starting from January 2017 until March 2018, i.e., a total of 15 exports, each covering the first seven days of the corresponding month.⁷ The table also lists the number of systems, processes and crashes that occurred during the various time periods. Ultimately, we included only those systems where at least one software technology was recorded on a process in order to be applicable to our approach.⁸ Additionally, all processes without software technology information must be discarded as well (compare columns *#Proc.* and *#ST-Procs.*), leaving us with fewer crashes, especially in the first couple of exports (compare columns *#Crashes* and *#Crashes for ST-Procs.*). It must be noted that the properties of the remaining crashes are not fully complete as well. This is because of two reasons: First, the aforementioned list clearly indicates that not every property can be collected for every crash (technology specifics). Second, properties cannot be resolved sometimes, for example, due to a data extraction error or if the property was (temporarily) not available. [Figure 3.4](#) shows the crash property availability for all our crashes that occurred on processes with at least one software technology (column *#Crashes for ST-Procs.*), grouped by each monthly export.

Ultimately, the dataset covers roughly 500 systems and an average of 14000 crashes per month (11000 after dropping processes without any software technologies), however, there is a strong tendency of an increasing number of systems⁹ as well as crashes, which is an indication for a growing customer base. Furthermore, the data reveals that the differences between the number of processes (*#Procs.*) and the number of processes with software technologies (*#ST-Procs.*) shrink over time, meaning that more detailed data is available, which, in turn, results in a higher number of usable crashes (compare columns *#Crashes* and *#Crashes for ST-Procs.*). These trends can also be seen in the visual representation of the data summary in [Figure 3.3](#), where we can also inspect the export differences of the various metrics more easily. Further details on the data exploration can be found in the appendix (cf. [Section B.1 on p. 199](#)).

⁷Due to a configuration error, the export of February 2018 actually starts from 29th of January rather than February 1st. However, we still have a complete week at our disposal.

⁸This also means that we dropped some systems where process crashes occurred, which is fine since we cannot use such systems in our approach anyway.

⁹In the majority of the cases, decreases in system counts can be attributed to trial customers that only appear in some of the observed months.

Export	Start 00:00	End 23:59	#Systems	#Procs.	#Crashed Procs.	#ST-Procs.	#Crashed ST-Procs.	#Crashes	#Crashes for ST-Procs.
January 2017	01.01.	07.01.	149	46985	92	16984	42	788	192
February 2017	01.02.	07.02.	203	71463	132	24970	67	3201	387
March 2017	01.03.	07.03.	185	61984	158	22826	78	4027	2253
April 2017	01.04.	07.04.	216	80117	256	29708	162	6103	1967
May 2017	01.05.	07.05.	233	81917	254	34086	154	2216	1154
June 2017	01.06.	07.06.	319	140953	333	51077	214	13300	11390
July 2017	01.07.	07.07.	318	119624	347	51877	257	9159	2476
August 2017	01.08.	07.08.	430	185056	404	84436	268	10466	5550
September 2017	01.09.	07.09.	514	210529	1548	132720	495	12353	7653
October 2017	01.10.	07.10.	591	278463	717	219526	606	24266	16682
November 2017	01.11.	07.11.	675	348508	980	281154	864	31662	25872
December 2017	01.12.	07.12.	734	373395	1763	302742	1159	46648	42056
January 2018	01.01.	07.01.	819	384146	380	307806	207	1160	1015
February 2018	29.01.	04.02.	931	437400	360	345396	259	1482	1314
March 2018	01.03.	07.03.	879	522124	1668	427692	1499	48421	42882
Average			480	222844	626	155533	422	14350	10856

Table 3.5: Summary of our real-world monitoring dataset. Start and end entries are in the format *day.month*, and every export begins at 00:00 and ends at 23:59 UTC (Coordinated Universal Time). The # character represents the number of systems, processes and crashes. *Procs.* is the abbreviation for processes, *ST-Procs.* are processes with at least one software technology (ST).

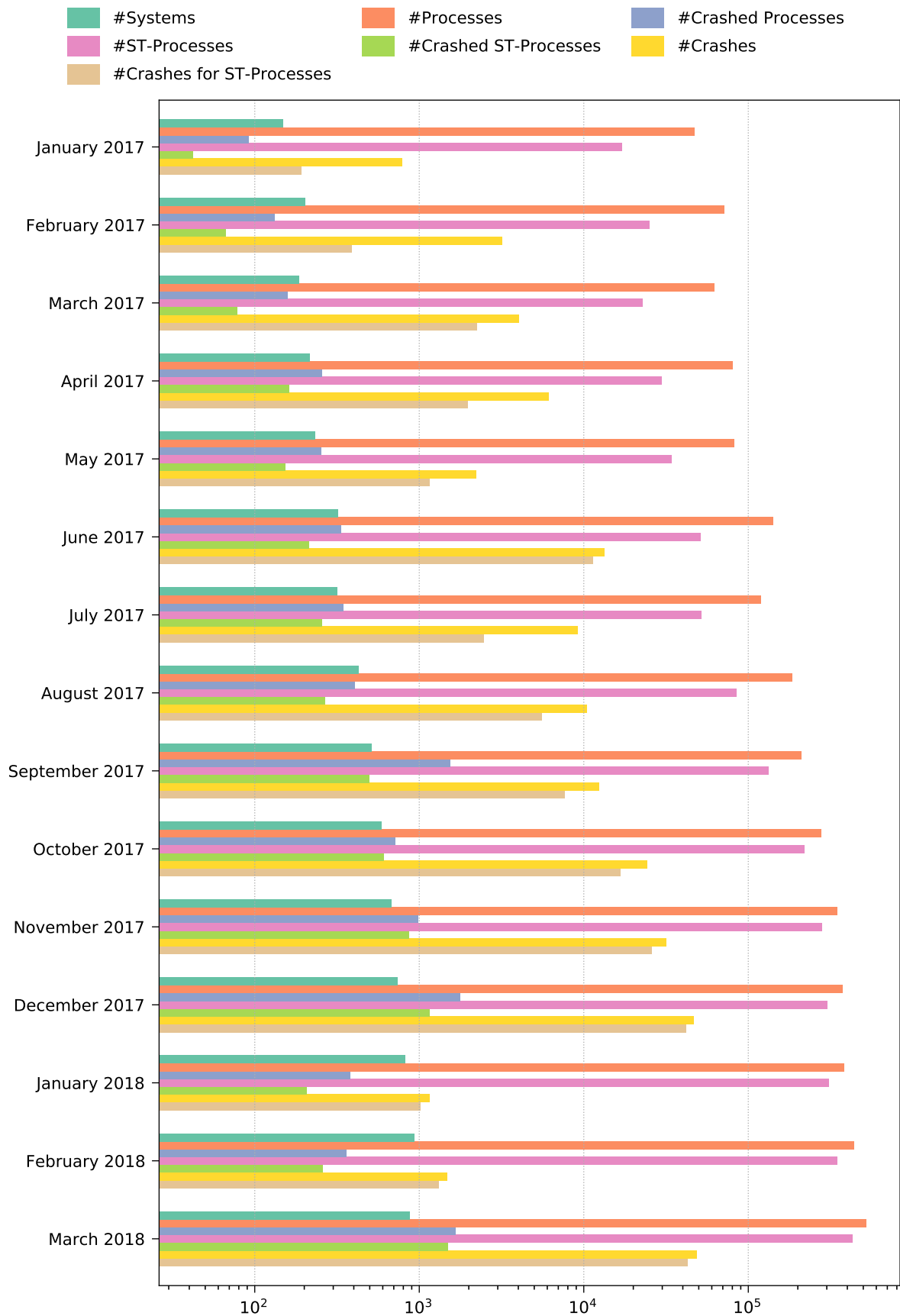


Figure 3.3: Visualization of the data summary presented in [Table 3.5](#) (logarithmic scale). The # character represents the number of systems, processes and crashes.

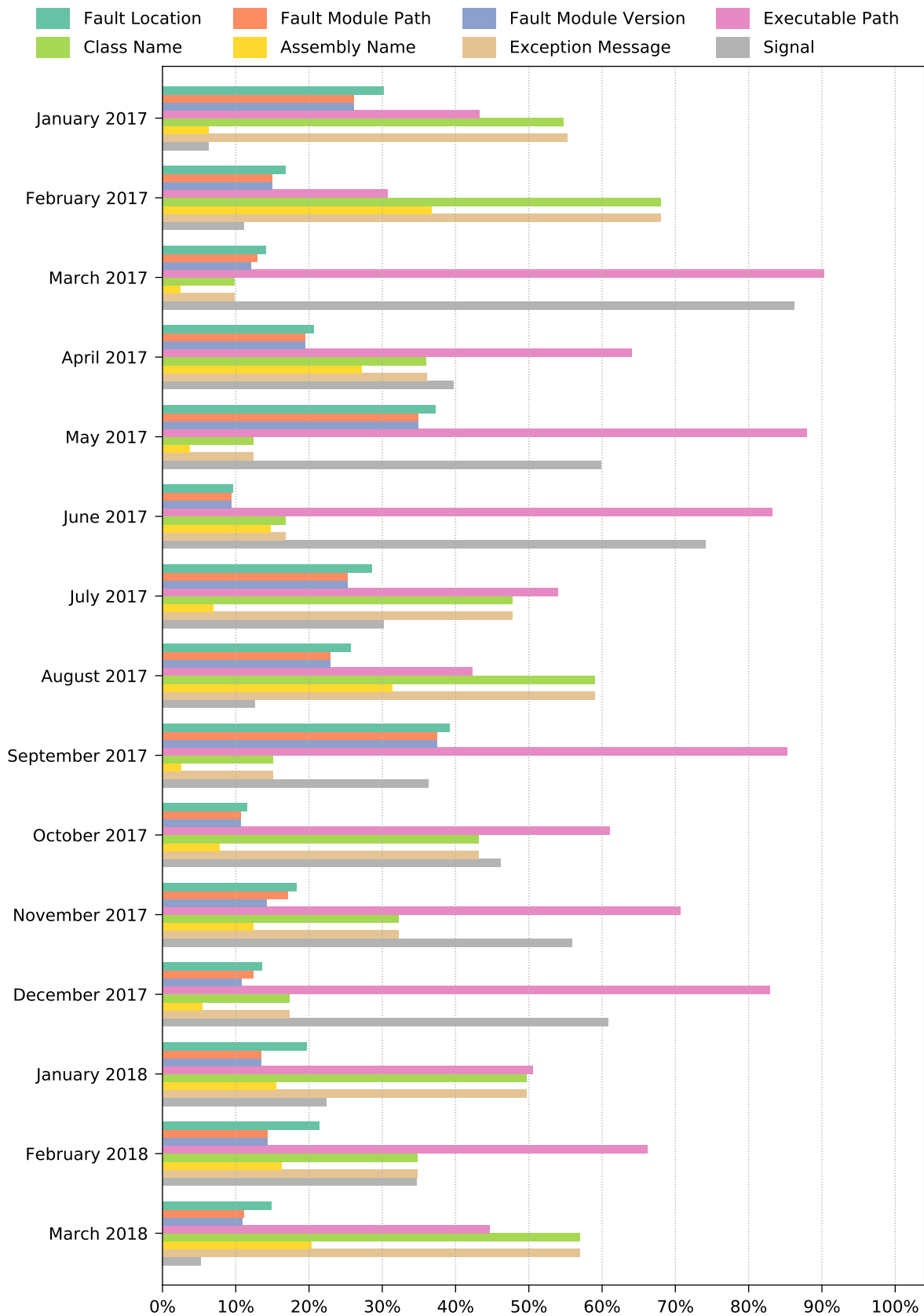


Figure 3.4: Availability (in percent) of the different crash properties for all crashes of processes with at least one software technology (cf. column *#Crashes for ST-Procs.* in [Table 3.5](#)).

3.5 Evaluation

Given our two datasets, we can now begin with the evaluation of our approach. Specifically, we would like to investigate whether we can identify error-prone software technologies and incompatibilities between them across multiple software systems based on common crash properties. We formulate three research questions (RQ) that we originally defined in [157]:

- RQ 1: Can our *automated analysis* find error-prone software technologies in a multi-system environment? To answer this question, we investigate 1-tuples within the real-world dataset from our industry partner.
- RQ 2: Do the results of the crash property analysis reveal insights to identify cross-system root causes? We want to detect the common root cause in the data with an appropriate selection of crash properties and a *manual investigation* of the results. Given the real-world dataset, we perform such a manual analysis based on selected results.
- RQ 3: Can *process communication* provide further insights to find connected, error-prone software technologies? In addition to the 1-tuple analysis, we extract 2-tuples for the real-world dataset and make a comparison to see whether they yield additional information or if the 1-tuples are already sufficient.

In the following, we cover the results of our approach when applied to the real-world, industrial dataset as described in Section 3.4. We first present the automatically detected top-ranked 1-tuples, where we perform a manual inspection for a selection afterwards. Finally, we also run a 2-tuple analysis to investigate the impact of process communication.

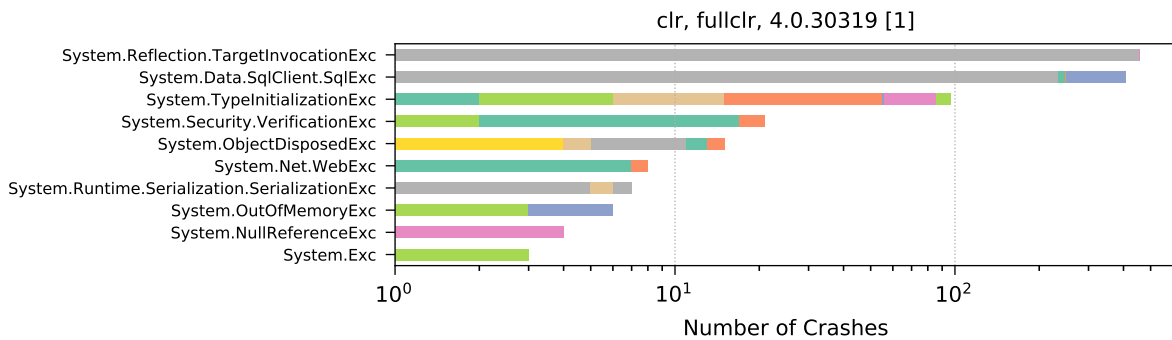
3.5.1 Automated Analysis

For investigating our first research question (RQ 1), we performed a 1-tuple analysis for each of the 15 months and decided to use the class name as main crash property to see whether the crashes occurred in the same set of classes.¹⁰ Moreover, the number of crashes with this property is sufficiently high (cf. Figure 3.4), and it should lead to more helpful results compared to using the process signal or the executable path, where also many crashes have these properties attached, but the information is simply too coarse for a meaningful manual investigation. Each monthly export yields top-ranked 1-tuples that we can visualize with the bar plots showing the corresponding crash property groups. Since our main objective is still finding multi-system crashes, we excluded groups which only consist of crashes that occurred in a single system (visually speaking, these are single-colored bars). Moreover, we only list the top ten groups at maximum, i.e., the ten groups with the highest number of crashes, to avoid an overloaded output. Finally, we deliberately do not show missing property entries which would otherwise be displayed as “missing”. The reason is that we expect some processes not to have this information attached, for instance, processes executing code that does not support the concept of classes (e.g., code written in C). We thus only inspect crashes that occurred within certain runtime environments (e.g., Java or .NET), which is perfectly fine as long as we keep in mind that we did not analyze all crashes.

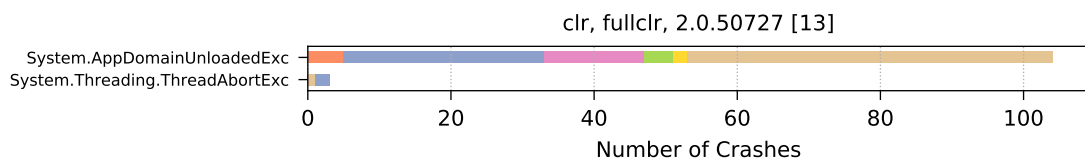
Figure 3.5 provides three representative examples of top-ranked 1-tuples. In the bar plot of Figure 3.5a, we can see the groups for our selected class name property of the 1st-ranked tuple (`|clr, fullclr, 4.0.30319|`) of the October 2017 export, where *clr* is the type, *fullclr* the

¹⁰Of course, the class name alone does not mean that the exception was caused by the same reason, but it does serve as a common starting point for further investigations.

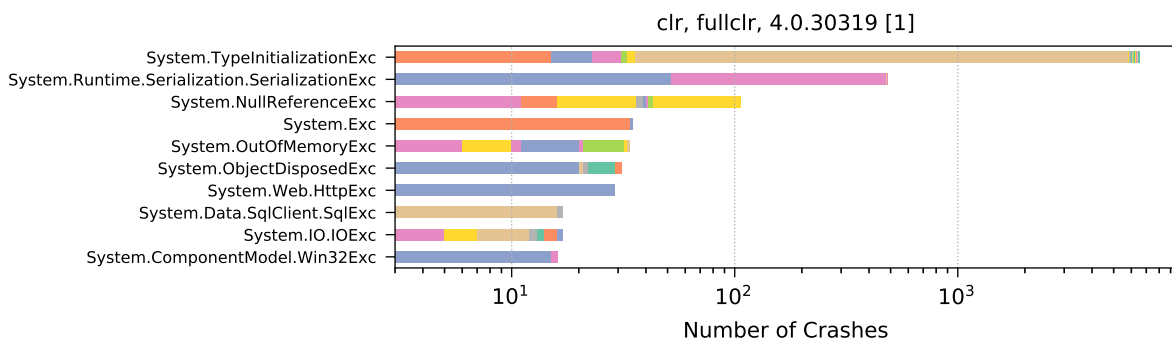
edition and 4.0.30319 the version (already the compacted and merged number as defined in our approach). This tuple is part of the .NET framework in version 4.7, where *clr* stands for Common Language Runtime. The `System.Reflection.TargetInvocationException` occurred most often (456 times) but only in two systems, whereas, for instance, the `System.TypeInitializationException` occurred in 97 crashes but in eight different systems. [Figure 3.5b](#) shows another crash property groups example of the 13th-ranked tuple (`|clr, fullclr, 2.0.50727|`). Evidently, the 104 crashes with the `System.AppDomainUnloadedException` across six systems appears to be interesting and worthwhile of a manual investigation (cf. [Section 3.5.2](#)). Lastly, the 1st ranked tuple in [Figure 3.5c](#) happened to be the same as the 1st-ranked tuple in the October export but with different crash distributions. Here, the `System.TypeInitializationException` occurred significantly more often (6523 times) and in 42 different systems, although a direct comparison between exports should be treated with care as the number of systems and crashes changes as well. Another interesting property group could be the `System.OutOfMemoryException` with a not necessarily high number of occurrences (only 34 times) but with nine affected systems. Perhaps this could indicate some sort of memory leak. A full overview of the five top-ranked tuples for all monthly exports can be found in the appendix (cf. [Section B.2 on p. 204](#)).



(a) Export October 2017: 1st-ranked 1-tuple with a total of 1025 crashes.



(b) Export October 2017: 13th-ranked 1-tuple with a total of 107 crashes.



(c) Export March 2018: 1st-ranked 1-tuple with a total of 7293 crashes.

Figure 3.5: Property groups of top tuple examples of the class name crash property. The number in the square brackets represents the rank (1 = highest possible rank). *Exc* is an abbreviation for *Exception* to shorten unnecessary long labels.

All these cases are prioritized suggestions by our approach that ultimately an engineer has to inspect and decide which of them to study more closely. For demonstrating purposes and to answer our second research question whether these results can actually be meaningful, we took on the role of such an engineer and initiated a manual investigation.

3.5.2 Manual Investigation

As an example of what an engineer could do with the output and results provided by our automated approach, we decided to look at the crashes of the software technology (`|clr, fullclr, 2.0.50727|`) in [Figure 3.5b](#), and specifically, at the `System.AppDomainUnloadedException` property group in more detail. As a starting point for our investigation, we list all the other properties of the crashes with this exception in [Figure 3.6](#). Unfortunately, we cannot gain any additional insights here, as most of the properties are either missing, could not be collected (assembly name) or are simply not useful enough (exception message; however, we do now know that the message is the same for all crashes).



Figure 3.6: All other crash properties of the crashes with the `System.AppDomainUnloadedException` for the tuple (`|clr, fullclr, 2.0.50727|`) in [Figure 3.5b](#).

We thus continued our search for a possible common cause on the Internet and found several hints that this can happen due to bugs in SQLite and NUnit, using the Visual Studio Test Runner instead of the MSTest Runner or switching between different versions of the .NET framework. For a closer inspection, we then looked at the individual crashes and the corresponding processes including the time where the events occurred, as well as additional log files (if available) to determine whether there are correlations or similarities. Utilizing the topology metadata of the affected processes, we found that the majority of crashes occurred in the `ReportingServicesService.exe` and happened every twelve hours. Given this information, we searched for possible explanations and finally discovered that by default, Microsoft’s Reporting Services recycles application domains every twelve hours as well [\[119\]](#), which is most likely the common root cause for these process crashes. We were thus able to identify an actual,

common problem across multiple systems based on the results from our approach (RQ 2), where fixing it could potentially benefit all the affected systems.¹¹

Note that this manual investigation is not part of our automated tuple analysis approach but simply serves as a proof of concept to show that we can indeed make use of the provided results. Naturally, it is not guaranteed that we can find a common cause in every case (such as for the `System.AppDomainUnloadedException` example from above). However, we do not claim this in the first place, as our approach is merely a multi-system crash prioritization and not an automatic root cause identification tool, which would surely be an interesting field of research in future work. However, we note that detailed crash properties (e.g. stack traces or library versions) would greatly enhance the outcome of our approach and thus aid such a manual investigation, which we discuss in [Section 3.6.2](#).

3.5.3 Process Communication

To investigate process communication (RQ 3), we also performed a 2-tuple analysis to check if we can gain additional information compared to the 1-tuple analysis from before. While the 1-tuple results may indicate errors in single software technologies, we expect from the 2-tuple results that we can potentially detect communication-based problems, for instance, if two neighboring processes and their technologies are (in parts) incompatible. The idea is that here, the software technology pair, i.e., the combination of two technologies, crashes more often than they would crash individually and independently from each other. In other words, if we see crashes of a process with technology A which communicates with another process with technology B , and otherwise, technology A seems to be running without any issues (e.g., no other process communication or communication with different technologies than B), then we can assume that the 2-tuple (A, B) could be suspicious, more so than the 1-tuple (A) . Our approach helps us in identifying such cases, which would then be ranked higher than their corresponding 1-tuple counterparts. For example, assume that technologies A and B sometimes crashed and sometimes survived across multiple systems, but the combination (A, B) often crashed. In the 1-tuple analysis, this would lead to average ranks, maybe somewhere in the middle. In the 2-tuple analysis, on the other hand, (A, B) would be ranked very high (using our default ranking metric). Conversely, if, for instance, technology A crashed all the time on its own and regardless of any neighboring software technologies, then we would obtain a high rank for its 1-tuple (A) but not necessarily for its 2-tuples.¹²

We can visualize this 1-tuple to 2-tuple rank comparison by creating a so-called *rank-diff* plot, for which we show an example in [Figure 3.7](#). The x-axis represents the rank of a 2-tuple (A, B) and the y-axis is the difference to its highest 1-tuple rank, which is either from tuple (A) or (B) , depending on which of the two has the better rank. The difference is then calculated as the 2-tuple's rank minus the better 1-tuple's rank, meaning that a positive difference indicates that the 1-tuple was more important, whereas a negative difference indicates that the 2-tuple was more important. In the example, there are two 2-tuples. Assume that these tuples are (A, B) and (B, C) with ranks 1 and 2, respectively (directly obtained from the x-axis labels in the figure). Further assume that the ranks of the corresponding 1-tuples are $(A) = 4$, $(B) = 5$

¹¹We do not have access to the actual systems and the source code, so our manual investigation inevitably comes to a stop at this point. A solution would be to inform a corresponding engineer of every affected system provider and instigate an in-depth cross-system investigation. However, this is outside the scope of this thesis.

¹²Of course, it can happen that a 2-tuple (A, B) crashed often as well (100% lifetime overlap of technologies A and B), which, in turn, can lead to a high 2-tuple rank, but in this case, this high rank would already be explained by the always crashing 1-tuple (A) .

and (C) = 1. For each 2-tuple, we first determine which of its 1-tuples has the better rank.¹³ For (A, B), we choose (A) because its rank 4 is better than (B)'s rank 5. Analogously, we select (C) for (B, C). Now, we calculate the differences: (A, B) - (A) = 1 - 4 = -3 and (B, C) - (C) = 2 - 1 = +1. -3 means that the 2-tuple was more important (the corresponding 1-tuple rank was worse), whereas +1 means that the 1-tuple was (slightly) more important. From a visual perspective, we can say that the more bars in the plot are ≥ 0 , the less the process communication seems to affect the tuple results.

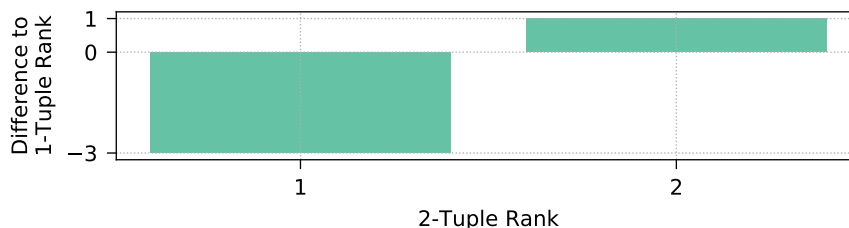
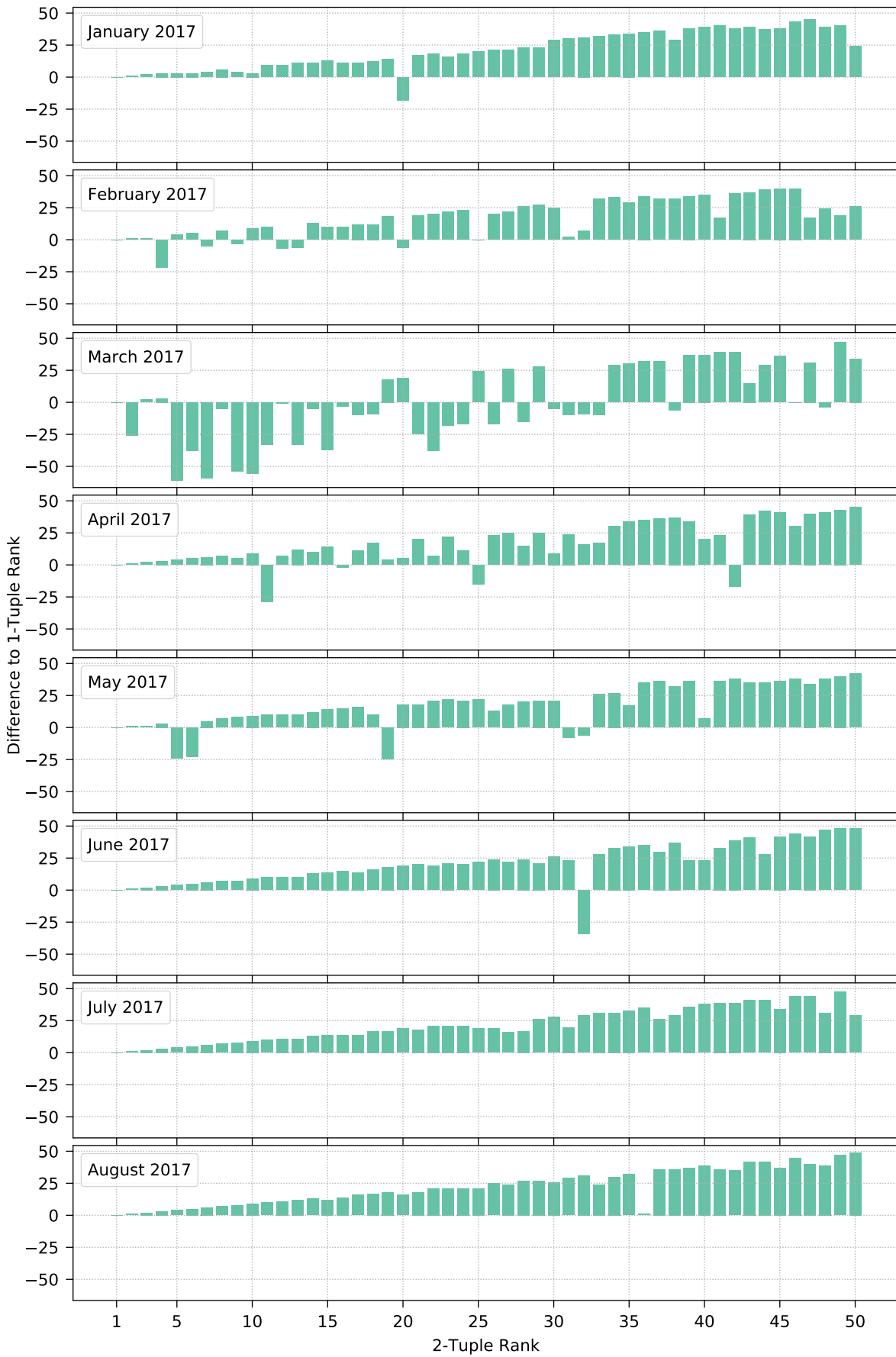


Figure 3.7: Example rank-diff plot for comparing the ranks of two 2-tuples with the ranks of their corresponding 1-tuples.

In [Figure 3.8](#), we show the rank-diff plots for all our monthly exports, covering the top 50 2-tuples. In most of the exports, the majority of the differences are positive, which means that the 2-tuples were generally considered less important than their 1-tuple counterparts, or in other words, the 2-tuples simply contained a 1-tuple that was also top-ranked. In these cases, analyzing 2-tuples does not yield additional information, as no new/unseen technologies were marked as suspicious. We can also observe a growing trend, i.e., the differences to the 1-tuple ranks are increasing. Such a trend occurs if there is a highly ranked 1-tuple (A) that happens to be part of multiple top-ranked 2-tuples. For instance, if there are some 2-tuples with the same technology A and some other, low-ranked technology X_i , and if these 2-tuples have adjacent ranks, i.e., $\text{rank}(A, X_{i+1}) = \text{rank}(A, X_i) + 1$, then the differences would be monotonically increasing, since the rank of the corresponding 1-tuple (A) remains constant. There are some exceptions with negative differences, although in most cases, such 2-tuples only occurred in one or two systems, which are not of particular interest to us given our main objective of identifying cross-system crashes. In fact, there are only five tuples where the number of crashed systems is at least three, which are listed in [Table 3.6](#). Three of them have a low negative difference of -2, -3 and -6, respectively, meaning that the corresponding 1-tuples have similar ranks, so we do not gain sufficiently more information here. The only remaining 2-tuples of potential interest are (`|websphere, null, 8.5.5|`), (`|java, ibm, 1.7|`) and (`|java, ibm, 1.7|`, `|java, ibm, 1.7|`) of the March 2018 export, where an engineer could again analyze their crash properties and start another manual investigation to check whether a common root cause exists that can be attributed to some form of (partial) software incompatibility between the technologies of these tuples.

Taking these results into account, running a 2-tuple analysis to inspect the process communication does not seem to yield much additional, useful information, making the 1-tuple analysis sufficient enough. However, this insight only holds for the data we analyzed, and other datasets or data from other export periods might result in a different 2-tuple to 1-tuple comparison.

¹³Remember that the rank is the position of a tuple after sorting all tuples according to their values obtained from the ranking metric (cf. [Equation 3.1](#)) in descending order. In contrast to the ranking metric, which is a higher-is-better metric, the rank is a lower-is-better measure, where a value of 1 means the highest/best possible rank.



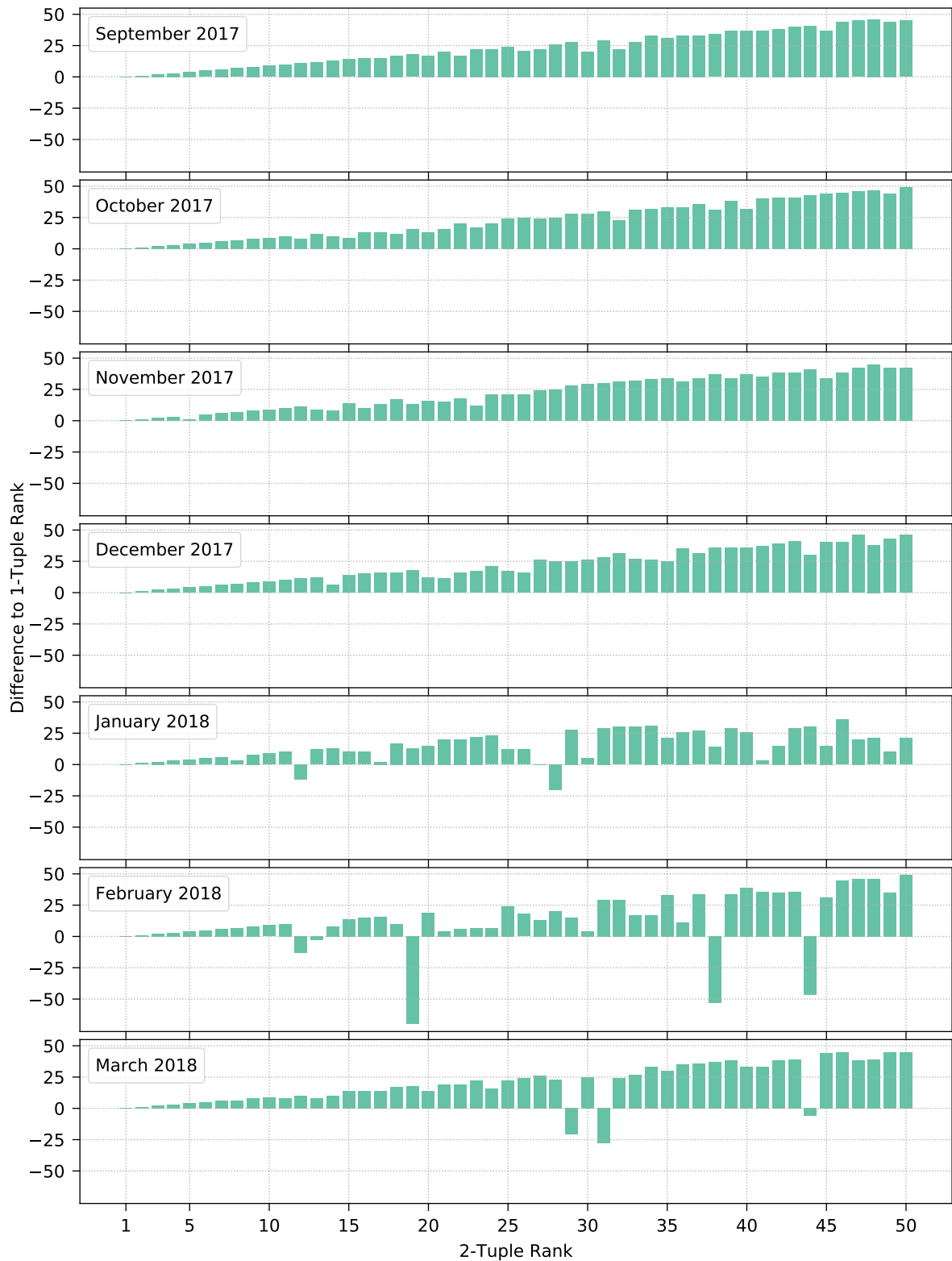


Figure 3.8: The rank-diff plots for each of the 15 months of our export data. Each plot shows the 50 top-ranked 2-tuples and the difference to their corresponding (best) 1-tuple rank.

Export	2-Tuple	Rank (Diff.)	#c	#nc	#cs	#ncs
Apr 2017	(asp.net, null, 4.5.2 , wcf, null, 4.5.2)	16 (-2)	194	565	3	11
Feb 2018	(dotnet, .net framework, 4.7.2114 , clr, fullclr, 4.7.2114)	13 (-3)	110	6758	7	52
Mar 2018	(websphere, null, 8.5.5 , java, ibm, 1.7)	29 (-21)	1489	178688	5	22
Mar 2018	(java, ibm, 1.7 , java, ibm, 1.7)	31 (-28)	1333	178620	6	30
Mar 2018	(websphere, null, 8.5.5 , websphere, null, 8.5.5)	44 (-6)	569	47526	6	24

Table 3.6: The five 2-tuples with a negative rank difference (cf. [Figure 3.8](#)) and at least three crashed systems. *null* indicates a missing edition in the software technology. Abbreviations: *Diff.* = difference to (best) 1-tuple rank, *#c* = number of crashes, *#nc* = number of times the tuple did not crash, *#cs* = number of crashed systems, *#ncs* = number of systems where the tuple did not crash.

3.6 Discussion

We now discuss the evaluation results for our dataset as well as the approach itself. We start with general insights and lessons learned and then continue with problems and limitations that we encountered, finishing with threats to validity.

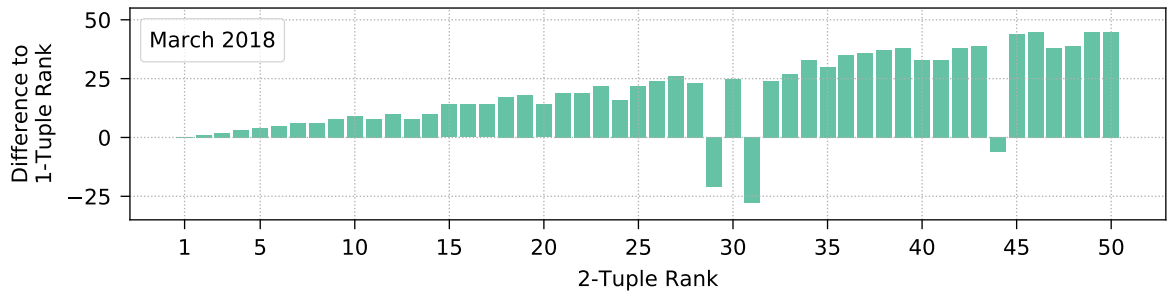
3.6.1 Lessons Learned

In the following, we present various lessons that we learned and general insights we gained when applying our approach to the industrial dataset:

For our data, the 2-tuple analysis for investigating process communication does not contribute much additional, valuable information. We already mentioned this part briefly in the previous section, where we discussed that this observation is only true for the data we analyzed. However, it is also important to note the impact of the chosen ranking metric (cf. [Equation 3.1](#)). Our focus was on finding high crash occurrences across multiple systems, but if we decided to follow a different goal, we would have to change the ranking metric, which, in turn, would result in different ranks and thus in a different 2-tuple to 1-tuple comparison as well. [Figure 3.9](#) displays rank-diff plots of the March 2018 export for three different ranking metrics. We can see the differences between the original ranking metric r ([Equation 3.2](#)), a ranking metric r_a that highlights the multi-system aspect even more by dropping the crash-count factor ([Equation 3.3](#)), and a ranking metric r_b that ignores the systems entirely and just calculates a tuple-crash ratio ([Equation 3.4](#)). Clearly, the focus of r_b completely deviates from that of our original goal as well as that of r_a , and [Figure 3.9c](#) thus shows a drastically different picture of the 2-tuple to 1-tuple comparison, which would then require another manual investigation to gain further insights.

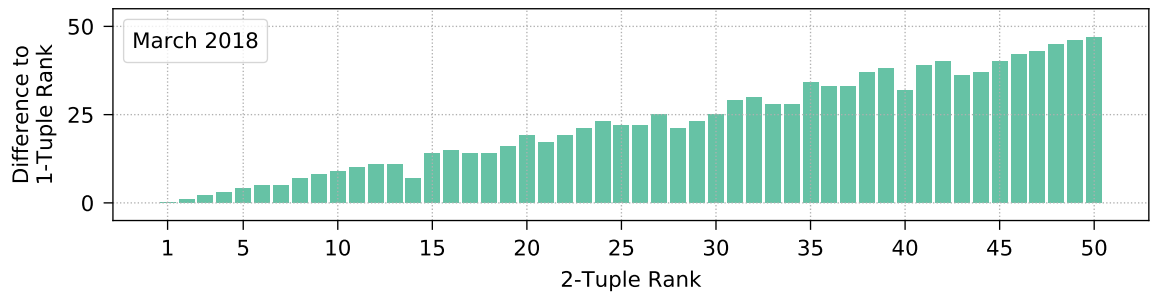
Performing an n -tuple analysis does not pay off. This follows directly from above. If already 1-tuples cover nearly all the necessary information compared to 2-tuples, then this is even more the case for n -tuples with $n > 2$. We also argue that crashes due to software incompatibilities should already be detectable with 2-tuples, as chances are much higher that an error occurs when processes communicate directly with each other rather than over multiple

$$r(f) = \frac{|f.cS|}{|f.cS| + |f.\neg cS|} \cdot |f.cS| \cdot f.crashed \quad (3.2)$$



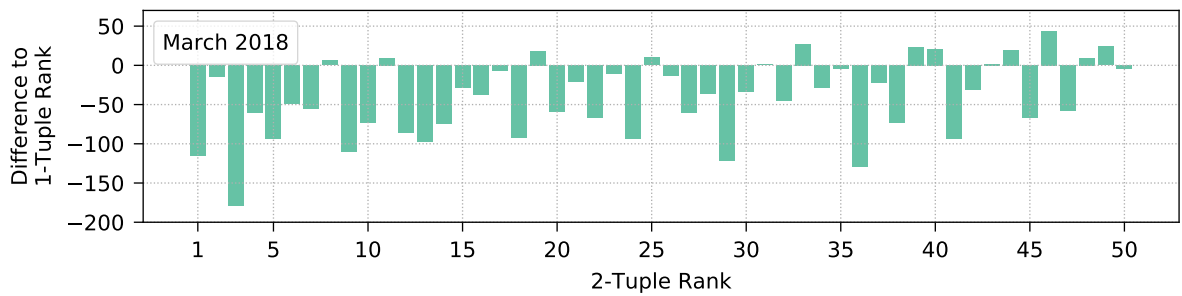
(a) The original ranking metric r (identical to Equation 3.1).

$$r_a(f) = \frac{|f.cS|}{|f.cS| + |\neg f.cS|} \cdot |f.cS| \quad (3.3)$$



(b) A new ranking metric r_a which drops the $f.crashed$ factor of the original metric.

$$r_b(f) = \frac{|f.c|}{|f.c| + |\neg f.c|} \quad (3.4)$$



(c) A new ranking metric r_b which solely focuses on the number of crashes and non-crashes.

Figure 3.9: Different ranking metrics and their effect on the 2-tuple to 1-tuple comparison of the March 2018 export, visualized by rank-diff plots.

(process) hops. Naturally, if the 2-tuples do yield significantly different results, an n -tuple analysis (starting with $n = 3$) can indeed be make sense.

Given appropriate crash properties, the approach yields useful results. As demonstrated in the manual investigation in [Section 3.5.2](#) the prioritized results provided by our automated approach can be used to investigate the causes of crashes for actual cross-system problems. Of course, not every top-ranked tuple necessarily indicates a common issue, but it does serve as a valid starting point for an engineer to decide whether further steps should be taken. Moreover, we can utilize different crash properties and thus gather more knowledge about the crashes. Unfortunately, the data we had at our disposal did not have detailed properties such as stack traces and library versions, from which we would have greatly benefited in both the automated crash property analysis as well as the manual investigation.

The manual crash investigation can be tedious. Our approach cannot automatically reveal a common root cause for a given set of crashes but only prioritize those that seem to be worthwhile to take a closer look. It is then still up to the engineer to perform a manual investigation, which can be a hard and tedious task, especially if the available data for the crashes is limited. The fewer crash properties are available, the more difficult it is to actually find the underlying reasons. In our dataset, we also have this problem to some extent, which we will discuss in the next section.

3.6.2 Problems and Limitations

A keen reader might already have seen that our dataset unfortunately suffers from missing detailed crash properties, i.e., there are some limitations imposed on us by the data (not the approach). With missing, we not only mean that not every crash has all properties available (cf. [Figure 3.4](#)) or that some entries carry mediocre up to entirely useless information (cf. [Figure 3.6](#)), but also that there could be much more valuable properties, most notably stack traces and attached library versions. If we had such details at our disposal, this would immensely aid the manual investigation for finding a possibly common root cause. Especially stack traces would help to pinpoint the exact error location. Since stack traces are complex and provide much more information than, for example, our plain class names, we would need a more fitting property equality measure for the crash property grouping step, since simply comparing two traces and checking if they are identical is most certainly not what the users want.¹⁴ Partial stack trace matching or some sort of stack trace similarity measure [\[26\]](#) could be a noteworthy extension of our approach. Besides stack traces, detailed log data could also prove valuable, which has been demonstrated in related work [\[200, 201\]](#).

3.6.3 Threats to Validity

Our dataset only covers the first week of each month. However, given that software technologies typically do not change that frequently, especially in production environments, we argue that one week suffices to get an overall insight into the different software systems. Moreover, if crashes occur due to erroneous implementations or incompatibilities, they should most likely occur within one week, so observing the entire month would just increase the total number of crashes but not change their distribution. Even if there are different distributions, we should at least be able to detect them on a monthly basis since we did not skip anything in between our exports, i.e., we collected the first week of every consecutive month. If they should actually

¹⁴It is highly unlikely that two programs of two different, independent software systems have exactly the same source code, even if the tasks are the same. If they are not identical, our approach currently would produce crash property groups that only contained a single stack trace each, which is not useful.

occur on a weekly basis, then we miss the analysis of these, but this does not invalidate the findings on all the other exports.

A different problem we have with our data is the timestamp of the software technologies of the processes. As already briefly mentioned in [Section 3.2.1](#), the only available timestamp is the end timestamp, i.e., the time the technology was last seen. Unfortunately, we do not have the exact information when a software technology became active. In our approach, we handle this problem by simply assuming that every technology starts together with its corresponding process as indicated in [Figure 3.1](#), which can be wrong (e.g., the technologies in the example of [Table 2.1 on p. 9](#) are most certainly updated to newer versions rather than both versions running from the start on). For 1-tuples, there is no impact on the number of created tuples, but we might classify some as crashed (depending on the time of the crash events), for instance, if a crash occurred right after the process start and one of the technologies had actually not yet been active at that point in time. For n -tuples, the assumption additionally leads to an increased number of tuples because we also generate tuples which do not actually overlap. [Figure 3.10](#) shows an example of these two cases. Process $P1$ has a technology $A1$ that is later updated to a newer version $A2$. Analogously, process $P2$ updates its technology $A1$ to $A3$. Since we are unaware of the actual start timestamps of $A2$ and $A3$, we incorrectly assume that both already started with their respective processes. We thus have the following 1-tuples: (A1) for both processes, (A2) and (A3). In the example, both processes crash at timestamp 120. With our simplified assumption, we thus wrongly mark the tuples (A2) and (A3) as crashed, although they only become active after the crash (timestamps 150 and 130), and therefore did not actually crash. For the 2-tuple case, we have the same problem with (A1, A3), where at the time of the crash, technology $A3$ is not yet active, so the tuple does not actually exist yet. Additionally, we create the non-existent tuple (A2, A1) because we assume that the technology $A2$ of process $P1$ and $A1$ of process $P2$ overlap. Of course, this assumption is wrong since $A2$ actually starts (timestamp 150) after $A1$ of process $P2$ already ended (timestamp 130). All this might sound worrying, however, we must bear in mind that this only occurs in cases where software technologies are actually updated within our one-week exports, which is not something that happens all too often, as already mentioned in the previous threat. Chances for a wrong classification further decrease by the fact that our assumption is not 100% incorrect. For instance, the tuple (A2) lives in the time range [150, 200] compared to our assumed range [100, 200], meaning that we are only incorrect when crashes happen in [100, 150), whereas everything is in order should they occur in [150, 200]. Even if we do incorrectly classify some tuples, this merely results in a slight increase in false positives, where an engineer would then have to invest more time in the manual investigation if such a tuple happened to be among the top-ranked ones. Of course, we could easily fix all these problems if the actual start timestamps of the software technologies were available, so it is not a problem of our approach but rather a problem of the inaccurate data we got.

Regarding external validity, the results can partially be generalized to other software systems since the process crashes and causes thereof are similar or can even be identical (cf. the manual investigation in [Section 3.5.2](#)). The technologies, the crashes and their properties are not specific to our industry partner because they originate from the hundreds of independent systems that we have at our disposal, supporting our claim of generalizability. Of course, as time progresses, our findings and results become less important due to the evolution of the technological landscape. Furthermore, our conclusion that 2-tuples do not provide significantly more information than 1-tuples cannot be generalized since different data (different systems, other data ranges, other crash properties) as well as different goals (achieved with appropriate ranking metrics) can yield significantly different results and must thus always be evaluated individually.

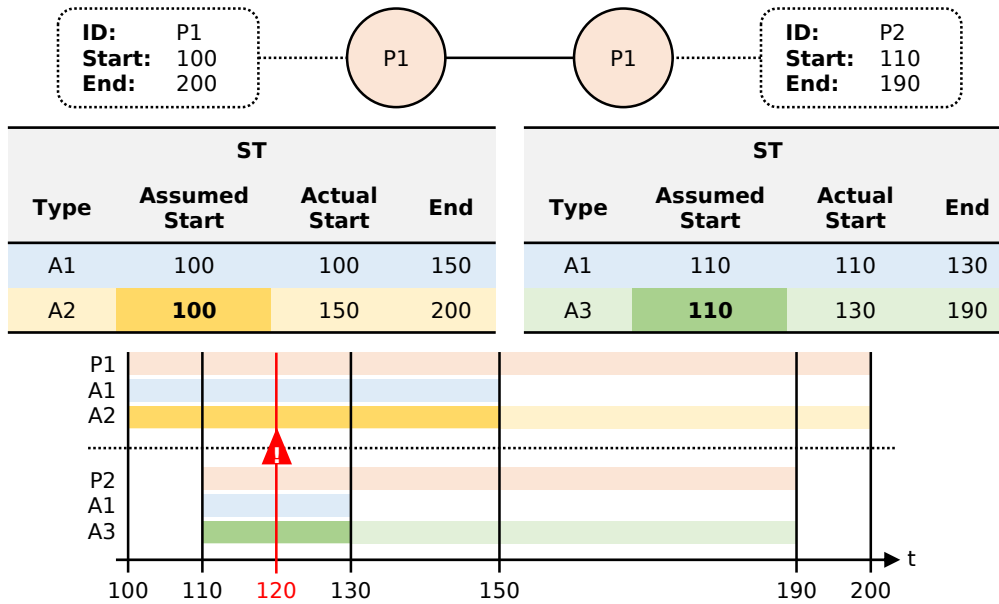


Figure 3.10: Example of the incorrect assumption of software technology start timestamps (highlighted with dark yellow and dark green) of two processes. At timestamp 120, crash events occurred on both processes.

3.7 Related Work

Software bug and crash management is a large, active field of research and can coarsely be divided into three categories that include optimization (of data related to bugs and crashes, e.g., the quality of bug reports), triage (prioritization, detection of duplicates, assignment to engineers) and fixing (localization of faults and their resolution) [202, 205, 181, 69]. Our approach mainly falls into the triage, whereas the manual investigation can be mapped to the fixing category, which, however, is not part of our actual approach any more. Managing bugs and crashes on individual systems, products or individual applications has been thoroughly investigated, and the ideas of the following related work is often closely related to our own goal. However, we do not focus on a single program or product family but take the crash grouping and prioritization to an even more general level, namely for entire, independent software systems that provide a wide range of different technologies and applications.

Ghafoor and Siddiqui [67] tried to find common bugs in different programs from the Mozilla product family (e.g., Firefox, Thunderbird, MailNews). They extracted stack traces via the Bugzilla API and grouped them according to a threshold-based stack trace similarity measure using the Levenshtein distance [104] for comparing the strings of the stack frames. In their evaluation, the authors analyzed 132 products with a total of 877000 bug reports and 17219 stack traces. They conducted a manual inspection and concluded that the resulting groups of similar bugs are indeed correlated. Fixing a common issue could thus benefit possibly all affected programs. In [48], the authors improved the automatic grouping of crash reports by yet another stack trace similarity. They applied a two-stage grouping where they first compared the top stack frame, i.e., the location of the occurred error, and then continued with the similarity of the entire trace. 82156 crashes across ten different releases of Mozilla Firefox were investigated, with the conclusion that 80% of the bugs can be identified using stack traces. Glerum et al. [68] presented Windows Error Reporting (WER) for processing the automatic collection of debugging data (e.g., stack traces and memory dumps) for reports from various Microsoft products such as Office, Windows or Windows Mobile. WER employs

a two-sided bucketing algorithm: The client-side heuristic for grouping similar crashes is based on information such as the program name as well as the module name, version and timestamp, whereas the server-side heuristic processes stack frames. Afterwards, the results are aggregated, sorted and grouped for prioritization purposes. With over ten years' worth of data and billions of reports, the authors highlighted the great success and results of their approach. Kim et al. [94] improved WER by using weighted crash graphs to combine the stack traces of all crashes of a bucket. Given a graph similarity measure, truly similar crashes are separated from those that the original bucketing heuristics incorrectly assigned to the same bucket. The tool ReBucket [42] aims to improve WER as well by also including a stack trace similarity measure. The authors implemented a measure called Position Dependent Model, which is based on the number of functions in a stack trace, their distance to the top stack frame as well as the offsets between matching functions. The evaluation of five Microsoft products with 1198 crash reports showed better results than the original bucketing. Another modification of WER was developed in [41], where the authors extracted additional information from memory dumps. Compared to syntactic stack trace similarities, this approach considers program semantics that are reconstructed from these dumps via a binary-level backward taint analysis. They evaluated 140 bugs in Windows 8 and Office 2013, where they could significantly outperform WER's triaging capabilities. Wang et al. [186] presented a different approach by investigating the correlation between different crashes. The correlation is based on structural information (crash type signature and stack trace), temporal information (similar times of occurrences) and semantic information (textual similarity of user comments). They defined several rules for identifying correlation and evaluated their approach on Mozilla Firefox and Eclipse bug reports with auspicious results. As we can see, most of these approaches utilize stack traces, since they have been found essential in the debugging process [161]. This highlights once more the unfortunate circumstances regarding our datasets, where such a crash information is not available, thus limiting the triaging process.

Shifting the focus on how to specifically handle recurring crashes and potentially providing fixes, Modani et al. [121] proposed automatic stack trace matching to retrieve fixes for problems that have previously already been solved. A similar idea was presented even earlier in 2005 by Brodie et al. [26]. Gao et al. [65] automatically parsed well-known Q&A sites (e.g., Stack Overflow), given the stack traces of the crashes, to find fixes for common bugs and root causes. In the evaluation, the authors studied 90 different Android projects with 161 issues with promising results, although they limited the crashes to be specific to the Android framework. We could include such an idea in our own manual investigation to support engineers in finding the root causes more easily. In [124], the goal was to identify recurring faults for which fixes and solutions already exist, where such errors occur most prominently due do not updating to newer software versions. The authors used function call traces to detect such recurrences and evaluated their approach on a commercial software application with different releases. It would be an interesting addition to our approach if we stored old/previous problems and their fixes, and then notified the engineers when crashes occurred due to the same issue, thereby providing the corresponding fix. In future work, this could theoretically even be extended to automatically fixing and repairing the failed software [122].

One step further is then to not only prioritize collected crashes but also to predict whether new ones are worth investigating in more detail. Kim et al. [93] presented an approach to automatically classify crash events as relevant or irrelevant (e.g., just an isolated event) by learning from previous crash occurrences, where higher crash counts were marked as more important. They collected crash event dumps from Mozilla Firefox and Thunderbird, and then used two machine learning algorithms for the relevancy prediction. Such an idea could also be a powerful extension of our approach if we were capable to pre-filter the thousands of crashes that occur in our multi-system environment, which could greatly improve our

prioritization procedure. In the area of prediction, there is also a branch called cross-project defect prediction [209, 138, 77, 136], where the goal is to leverage the data of multiple projects to be able to predict defects (process crashes could be an example of such defects) for systems that do not (yet) have enough data to build a within-project defect prediction model. Albeit challenging, it sounds quite interesting, especially if we investigated whether we could transfer the idea of cross-project prediction (or parts thereof) to our cross-system scenario and perhaps create a multi-system crash/event prediction approach.

3.8 Outlook

As it is the case with most research, there still remain many possible improvements, extensions and open challenges. First, we could collect more data, both in the sense that we could analyze the entire month instead of only its first week, but also that we could continuously create exports to see how the multi-system environment with the various software technologies evolves over time. Moreover, given their availability, it would be immensely helpful to also incorporate the start timestamps of technologies as well as detailed crash properties, such as stack traces and information about library versions, into our approach, which would improve our automated crash property analysis and further aid engineers in investigating the possible root causes. The manual investigation itself could also be greatly improved as already indicated when discussing the related work. For instance, we could store already identified common issues together with their possible solutions and fixes to provide a knowledgebase, where recurrences of crashes caused by the same problem could easily be detected, for which a corresponding remedy could then be provided. Another interesting part of future work could be the automatic detection of crashes, i.e., a crash prediction approach, where we could utilize data from multiple systems (cf. cross-project defect prediction).

Since the data exports from our industry partner did not provide enough detailed crash properties and the idea of a crash prediction sounded both promising and interesting, we decided to not pursue further improvements and extensions of our topology-driven crash analysis for now but rather continue with such a prediction approach. Given the same topological information and event structure, we included the remaining part of our industrial multi-system data, namely the time series, which form the basis of our event prediction approach described in the following chapter.

Chapter 4

Time-Series-based Event Prediction

In this chapter, we present our second approach that works within the multi-system environment. Now additionally utilizing the available time series data, our main objective is the prediction of performance-related events to notify system providers in a timely matter, which we try to accomplish with a combination of a sophisticated data preprocessing framework and supervised machine learning models. In the following, we describe the details of our approach, an evaluation on real-world as well as synthetic data, and problems and limitations that we encountered. The majority of this chapter was published in various conferences. More specifically, the data preprocessing framework was proposed in [153], initial results of the event prediction were published in [155, 156], and extended insights were presented in [154].

4.1 Motivation

As we already have seen in the previous chapter, faults that can lead to crashes are ubiquitous in software systems. However, crashes are not the only thing service providers have to worry about. Other performance-related events, such as slowdowns and performance degradations, are omnipresent as well, which can heavily impact the operation of a system. Researchers have thus put a strong focus on the areas of proactive fault management [149] and anomaly prediction [175], so system administrators can be notified before the actual occurrence of performance degradations, which enables them to take preventative measures and possibly avoid incoming crashes, downtimes or slowdowns altogether. There are two main directions in related work, often in conjunction with various machine learning methods: Approaches that use log files as a prediction basis [150, 199, 58, 134, 43] and those that utilize monitoring data such as memory, disk or CPU metrics [20, 175, 4, 164, 207, 135]. Many of the proposed solutions yielded promising results, but the majority of them only focused on the data of a single application or system, whereas a prediction within a multi-system environment is still lacking in current research, despite the possible benefits. One such benefit is the possibility to combine the data of multiple systems in order to alleviate the problem of insufficient data of an individual system, where we could otherwise not create an appropriate machine learning model. Another advantage would be a cross-system event prediction, where we could utilize the multi-system data to make predictions in a new, unseen system without any historic data. This idea is similar to cross-project defect prediction [209, 138, 77, 136], where the goal is also to use data from already seen projects and to try to predict defects in other projects.

In this chapter, we thus propose an approach for predicting performance-related events in a multi-system environment. The main motivation is that certain events might be explained due to effects and patterns within infrastructure monitoring metrics, e.g., increasing CPU

load, growing memory or suspicious disk behavior, which are common phenomena that can be observed across multiple systems, independent of the actual domain and tasks of those systems. With this idea in mind and after a discussion with experts from our industry partner, our approach is therefore based on 34 infrastructure monitoring time series, which include CPU, memory, disk and network metrics, with the goal to predict service slowdown performance events. Since our multi-system environment introduces a high complexity, we first present a highly customizable data preprocessing framework that aids us in creating appropriate output which can then be used for training our prediction machine learning models. There are many important points to consider, ranging from the sizes of the different systems to the data sampling and balancing strategies, as well as training and testing decisions and post-processing options. Overall, we create a variety of datasets and train numerous random forest classifiers to see how our approach copes with the multi-system data from our industry partner.

4.2 Data Requirements and Assumptions

Similarly to our topology-driven crash analysis from [Chapter 3](#), we primarily designed our new event prediction approach for the data from our industry partner. Nevertheless, it does work for other datasets as well as long as they fulfill certain requirements and assumptions, which we present in the following.

4.2.1 Topology

Each system must have an associated topology that stores all relevant components/entities (can be physical or abstract entities, such as sensors, hosts, servers or services) together with their connections (forming a graph), lifetime information and other arbitrary, optional properties. Every component has exactly one component type, and this type defines to which other types it may be linked. An individual component can then be connected to zero or more entities of these types. The lifetime of a component is given by a start timestamp, indicating when it first became active, and by an end timestamp, indicating when it was last seen, i.e., became inactive. The lifetime information is only required for components that potentially have time series data attached (see further below in [Section 4.2.3](#)). It is not necessary that the topology's graph is fully connected, which means that there might be some disconnected subgraphs or even solitary components. Another requirement is to select the main component type, which is going to be the main entry point when querying the data later on.

[Figure 4.1](#) shows a small example system with its topology and lifetime information. There are four component types, A , B , C and D , with the following connectivity rules: Entities of type A can be connected to B and B can be interlinked with both C and D . There are two disconnected subgraphs, one originating from component $A1$ and the other from $A2$. Assume that we started to monitor this system in the time interval $[t_1, t_6]$ and that component type A is not mapped to any time series. Components $B1$, $B2$, $C1$ and $D1$ all were active right from the start, whereas $B3$, $C2$ and $C3$ were only seen later at timestamps t_2 and t_4 , respectively. We can also see that some entities became inactive throughout the system's observation period. $B1$ only lived until t_3 , $C1$ until t_4 , and $B3$ and $C3$ until t_5 . There is no lifetime information available or, more specifically, necessary for the components of type A , since there is no time series data attached. It suffices to know that these components exist and are linked as displayed in the figure.

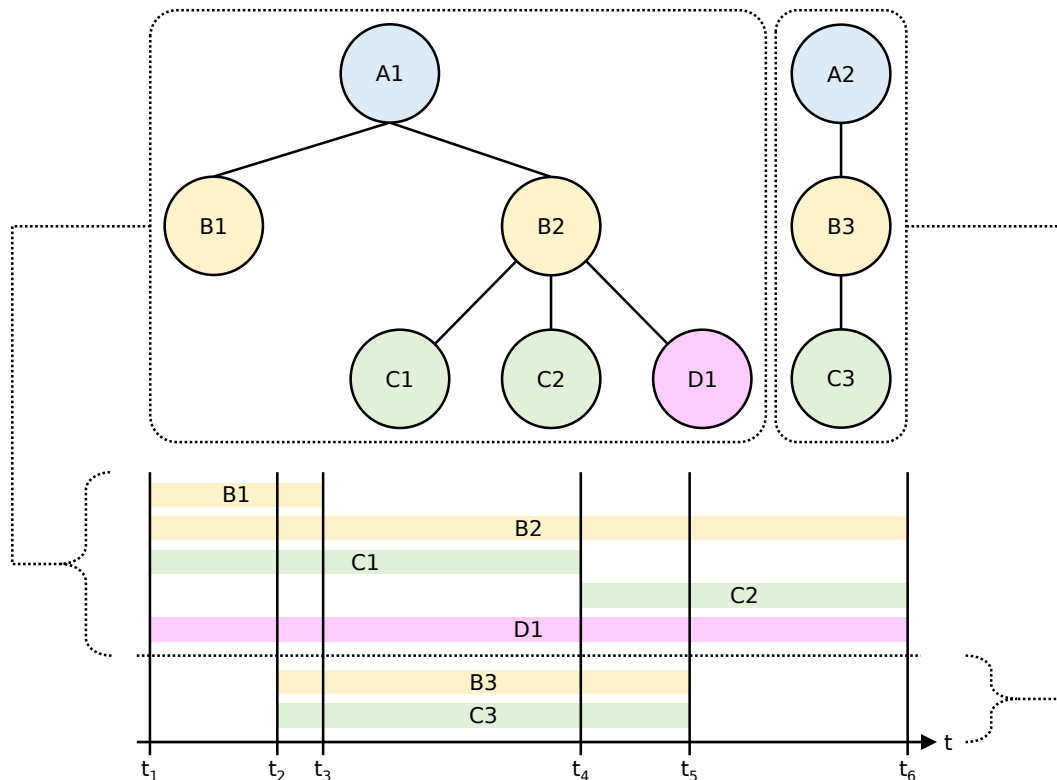


Figure 4.1: Example topology with lifetime information of a small system.

4.2.2 Events

Events can be of any event type of interest (e.g., crash, slowdown, etc.) that occur at one of the above components. For the prediction, only events that occur on entities belonging to the main component type are considered, and furthermore, only a single type of event may be selected. Each event must store the type, the time of its occurrence and the affected component.

[Figure 4.2](#) shows an example of an event occurrence using the same system as above. At timestamp t_e , some event occurred at component $A1$, which means that type A is the main component type in this example. At this time, $A1$ was linked with $B1$ and $B2$, and $B2$ was connected to $C1$ and $D1$ but not $C2$, since $C2$ had not yet been active at the time of the event. The other topology subgraph is completely independent of this event occurrence because no connecting links between the subgraphs of $A1$ and $A2$ exist.

4.2.3 Time Series

Time series are the primary data source for the event prediction. Each component type defines which kinds of time series can be collected at components of that type. It is not necessary that every component type must be mapped to time series, nor is it required that each individual entity must have data for every time series, i.e., data can be missing. The resolution of the time series must be the same across all systems, and they must be evenly spaced, i.e., $t_{i+1} - t_i = \Delta t \forall i \in [1, n - 1]$, where t_i is a timestamp and n is the length of the series. However, the global granularity can be arbitrary (seconds, minutes, hours, etc.).

Again, we use the system specified in [Figure 4.1](#). [Table 4.1](#) lists the component-type-to-time-series-kind mapping and all the components with their actually attached time series data.

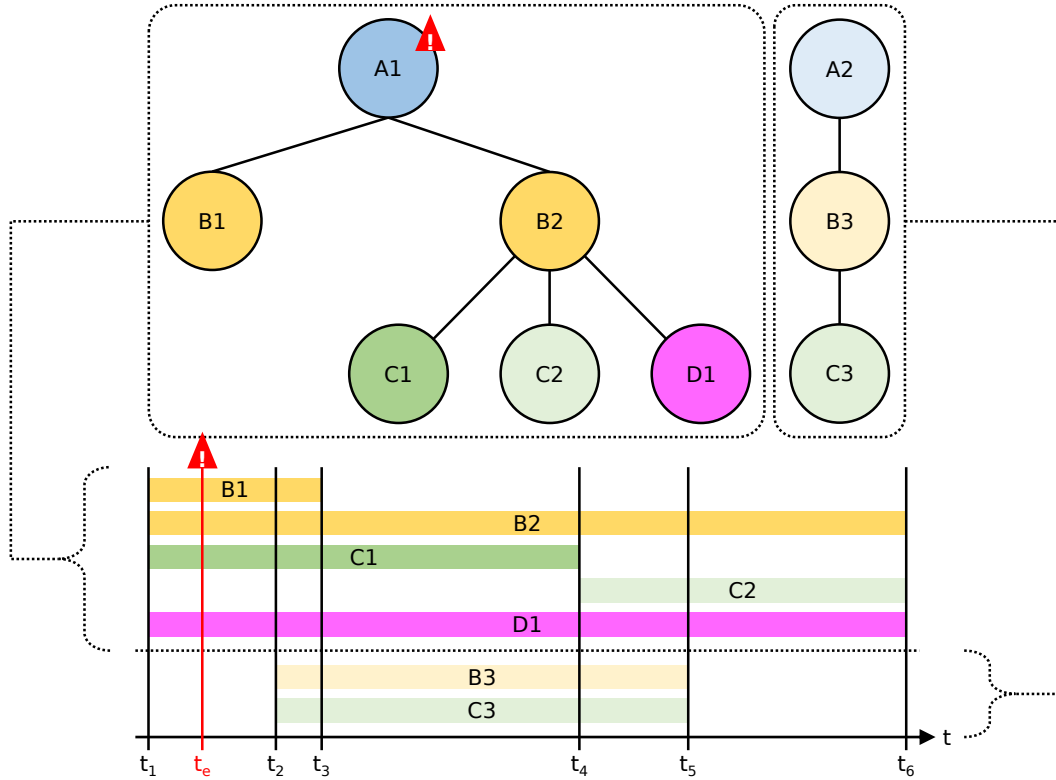


Figure 4.2: Example event occurrence at component $A1$ and timestamp t_e for the same system as shown in [Figure 4.1](#). The affected components are highlighted with darker colors.

As described above, type A has no time series mapping, so the entities $A1$ and $A2$ naturally do not have any attached data. The components of type B can have up to two time series kinds: B-01 and B-02, but only entity $B2$ actually has data for both ($B-01_{B1}$, $B-02_{B1}$). For $B1$, there is only B-01 available ($B-01_{B1}$), whereas $B3$ does not have any time series data attached. Analogously, components $C1$ and $C3$ support time series C-01 ($C-01_{C1}$, $C-01_{C3}$), and again, for $C2$, all data is missing. Lastly, $D1$ has data for all three time series kinds ($D-01_{D1}$, $D-02_{D1}$, $D-03_{D1}$). As specified before, all time series must be evenly spaced and have the same resolution, i.e., Δt of B-01, B-02, C-01, D-01, D-02 and D-03 is equal across all components (and all systems), and Δt could be one minute, for example. A visualization of example time series data for all the relevant components in consideration of their lifetimes is shown in [Figure 4.3](#).

4.3 Data Preprocessing Framework

Appropriately processing data is an essential and challenging part of machine learning applications, especially if the amount of data is large and its complexity high [\[95, 171, 27\]](#), which is often the case when dealing with time series. For instance, in the area of performance monitoring and predictive maintenance [\[149, 199, 164, 21, 86, 135\]](#), huge amounts of raw and unprocessed time series are continuously gathered, stored and analyzed for event prediction. Therefore, many researchers have proposed tools, frameworks and approaches [\[125, 16, 71, 168, 15, 57, 86, 135\]](#) for automatically processing the data and preparing it for further use, such as visualization or machine learning. While there exists work that considers the topology of a system [\[135\]](#), how to design a preprocessing framework for a multi-system environment is still an open challenge. We thus propose a sophisticated, highly configurable preprocessing

Component Type → Time Series Kind	Components → Attached Time Series
$A \rightarrow \emptyset$	$A1 \rightarrow \emptyset$ $A2 \rightarrow \emptyset$
$B \rightarrow B-01, B-02$	$B1 \rightarrow B-01_{B1}$ $B2 \rightarrow B-01_{B2}, B-02_{B2}$ $B3 \rightarrow \emptyset$
$C \rightarrow C-01$	$C1 \rightarrow C-01_{C1}$ $C2 \rightarrow \emptyset$ $C3 \rightarrow C-01_{C3}$
$D \rightarrow D-01, D-02, D-03$	$D1 \rightarrow D-01_{D1}, D-02_{D1}, D-03_{D1}$

Table 4.1: Example time series mappings for the same system as shown in [Figure 4.1](#).

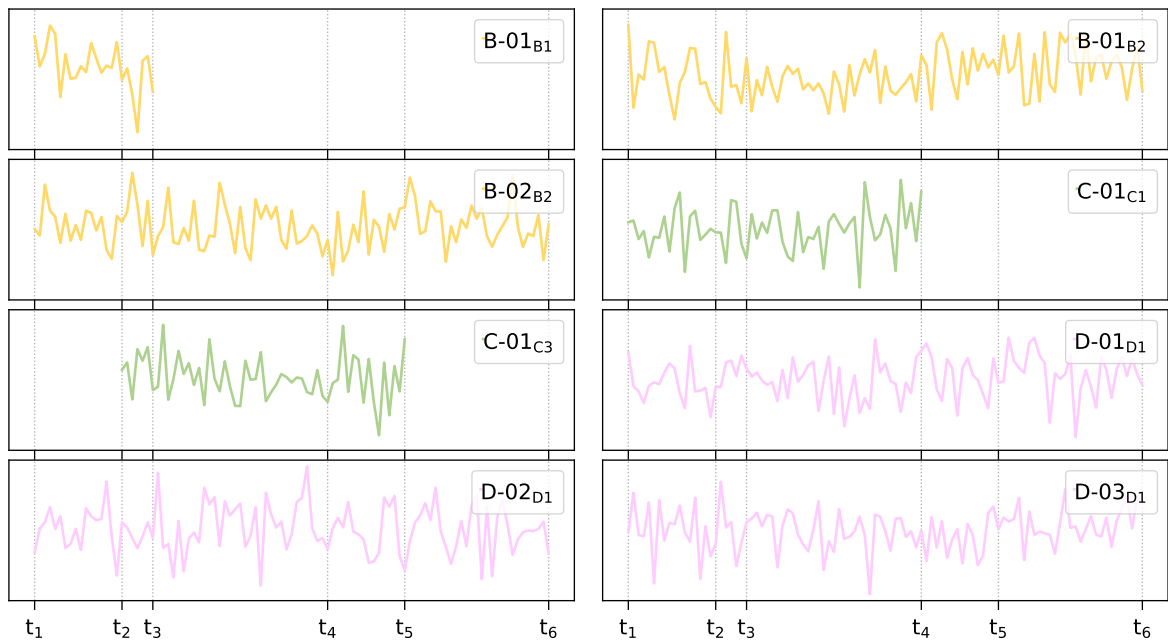


Figure 4.3: Example data for the *Attached Time Series* in [Table 4.1](#) with the corresponding component lifetimes as specified in [Figure 4.1](#).

framework that can handle event and time series data from topologies of multiple systems, and create ready-to-use CSV files that can directly be plugged into existing machine learning models.

4.3.1 Requirements

Originally defined in our contribution in [153], we first list five requirements (RQM) that a data preprocessing framework should fulfill, following the data assumptions introduced in Section 4.2 and recommendations from related work:

- RQM 1: *Multi-system analysis support* [209, 77, 157]: Handling data from multiple systems comes with new challenges that must be addressed by the framework. Specifically, the different sizes of the systems in terms of components must be taken into account, since these have a direct impact on the data balance.
- RQM 2: *System topology support* [135]: The topology of a system, i.e., the components and their lifetimes as well as connections, must be supported by the framework. An important task is also to map events and timeseries to the corresponding entities and to resolve all connections accordingly.
- RQM 3: *Data selection and sampling techniques* [125, 193]: Given potentially huge amounts of data, selecting and sampling play an essential role. The framework should enable users to exactly specify which systems, topological components, events and time series (for each component type) should be processed. Configuration settings for the time series must allow precise control over which parts and how they should be processed.
- RQM 4: *Missing data handling techniques* [71, 168]: Incomplete data is a common issue due to recording errors, data loss, bugs and various other reasons, especially with real-world data. A framework must thus be capable of appropriately handling missing system components as well as incomplete time series data.
- RQM 5: *Big data performance scalability* [66]: Continuously collecting time series data from hundreds of components in multiple systems results in large amounts of data. The performance of a preprocessing framework needs to scale well with increasing input sizes.

4.3.2 Preprocessing Pipeline

Our proposed framework is primarily designed for assisting users in creating time-series-based feature vectors that can be used for training machine learning models with the goal of event prediction. Repetitive and error-prone tasks are completely handled by the framework, allowing users to concentrate on the machine learning part rather than the equally important but often tedious data selection and preprocessing. We expect only three inputs: The raw data together with an appropriate interface for querying, and so-called *configuration* files (config as an abbreviation). After having established a data connection (typically only done once in the beginning), the only thing users have to provide are configs, which store every relevant preprocessing information and are thus the most important driver of our framework. Due to this reason, we opted for the YAML file format to aid users in reading, creating and adapting configs. Once a config is passed to the framework, the framework automatically processes the data according to this config and creates feature vectors stored in CSV files as output, which can directly be used for training and testing machine learning models.

A general overview of our configuration-based framework is shown in [Figure 4.4](#). It consists of four main steps, each of which takes the data from the previous step as input and produces an output for the next one, starting from the raw data and configuration files, and ending with the final CSV files. The first part of this preprocessing pipeline is establishing the *data access* to the raw data. Afterwards, a config is used to specify which parts of all the raw data should be selected for further processing (*data selection*). In the third step, the config defines all locations where actual data should be extracted (*sampling*), which results in (still empty/unfilled) samples. The fourth and last part is the *data extraction*, where the time series are actually queried and processed according to the provided config. This is done for all samples of the previous step, thereby filling them and creating feature vectors, which are ultimately stored in the final CSV output files. We describe the individual steps and their config settings [in detail](#) in the following sections.

4.3.2.1 Data Access

In the first step, we establish the data access to the raw data of the different systems, which includes the topologies, events and time series. We accomplish this by providing an application programming interface (API) from which we can query structured data.² i.e., the users have to provide corresponding implementations for their raw data sources, for example, an implementation to query the time series in an InfluxDB, one to provide the events that are stored in CSV files and another to access the topology from JSON files. If the different data sources do not change, this part only needs to be done once at the very start, the remaining tasks for users are then to simply write configs. The data access step *da* can be described with the function defined in [Equation 4.1](#):³

$$da(rawData) \rightarrow structuredData \quad (4.1)$$

where *rawData* describes the original data (topologies, events, time series) and *structuredData* the unified, structured view on that data, which the framework can access to extract and process (selected) parts in the next step.

4.3.2.2 Data Selection

The second part of the preprocessing pipeline is selecting the data that should be extracted by our framework, which addresses the requirements RQM 1 and RQM 3. As annotated in [Figure 4.4](#), the data selection can be specified in the config file, which contains the settings `systems` for selecting the systems, `eventType` for specifying the type of events for the `mainComponentType` and `timeSeriesKinds` for selecting the metrics we want to analyze. `from` and `to` specify the observation period, and `timeUnit` defines the resolution of the time series (e.g., one minute). The example in [Listing 4.1](#) shows how these config settings could look like in the YAML format. Here, three systems and four time series kinds are selected for processing. The events of interest are *ServiceSlowdowns* and the main component type is thus *Service*.

¹There are actually more configuration settings than mentioned here, but many are irrelevant for the points demonstrated in this thesis (e.g., the location of databases, the file output names or settings for randomization).

²The exact format of this structured data is implementation-specific and thus not explained in more detail in the context of this thesis. Roughly summarized, the structured data represents easy-to-use objects for topologies, events and time series, which are much more convenient to use in a programming context than to repeatedly having to access the raw data (e.g., parsing JSON files or querying an InfluxDB).

³The actual function should be $ds(rawData, c_{da})$, where c_{da} are the config settings for the data access part. However, as mentioned in the previous footnote, these settings are not important in the context of this thesis, which is why they are omitted in [Equation 4.1](#).

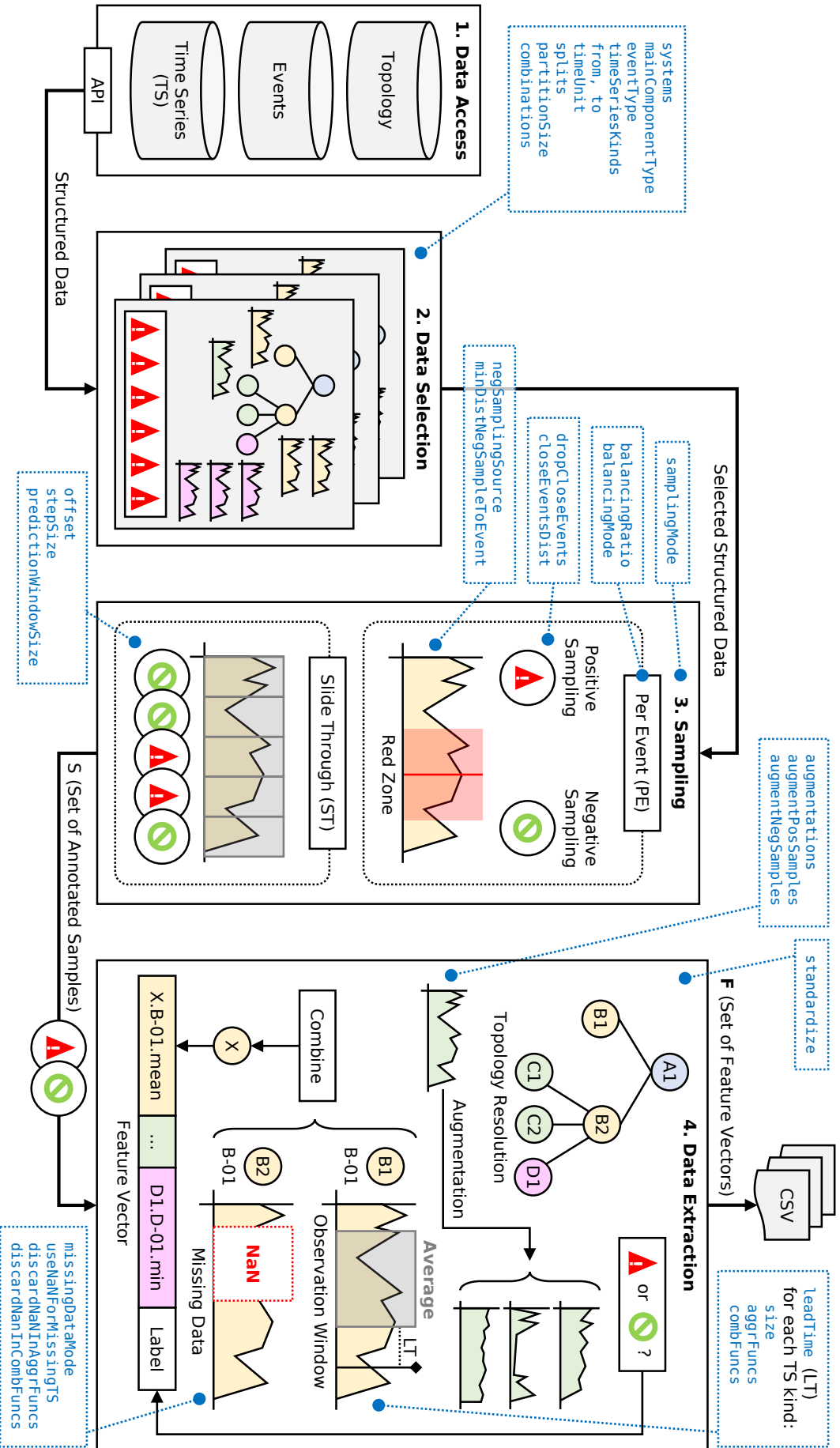


Figure 4.4: Overview of the configuration-based framework that consists of a four-step preprocessing pipeline: data access, data selection, sampling and data extraction. The blue annotations represent configuration settings that can be set for the corresponding parts.

The observation period is a single day in the resolution of one minute, i.e., the resolution of the individual time series of the four selected kinds is one minute. There are also a couple of settings on how to partition the inspected data. `splits` specifies how the above observation period should be split into multiple smaller parts (e.g., a 60% train split and a 40% test split). This splitting is done based on the observation period partitions, which are extracted with the given `partitionSize` (e.g., the period is 20 days and the partition size is one day, then 20 partitions are created which are then distributed according to the splits). Finally, the setting `combinations` can be used to repeat the splitting multiple times in a randomized fashion. With the input `structuredData` from the data access part, the data selection step ds can be described with the function defined in [Equation 4.2](#):

$$ds(\text{structuredData}, c_{ds}) \rightarrow \text{structuredData}_{ds} \quad (4.2)$$

where $\text{structuredData}_{ds}$ represents the subset of the structured data as specified via the corresponding configuration settings c_{ds} .

```

...
systems:
- "SystemA"
- "SystemB"
- "SystemC"
eventType: "ServiceSlowdown"
mainComponentType: "Service"
timeSeriesKinds:
- "H-01"
- "H-02"
- "D-04"
- "N-01"
from: "2020-07-21_00:00"
to: "2020-07-22_00:00"
timeUnit: "1_min"
...

```

Listing 4.1: Partial YAML config file with some data selection settings.

4.3.2.3 Sampling

The sampling step parses the selected data subset from the previous step to create so-called *annotated samples* $s \in S$, which store a timestamp t_s , the entity (must be of the main component type), the corresponding system and whether it is a positive (an event occurred) or a negative sample (no event occurred). Given the $\text{structuredData}_{ds}$ from above, the sampling step sp can be described with the function defined in [Equation 4.3](#):

$$sp(\text{structuredData}_{ds}, c_{sp}) \rightarrow S \quad (4.3)$$

where S is the set of annotated samples that are obtained when running the preprocessing framework with the sampling configuration settings c_{sp} . Two different kinds of sampling are supported, which can be selected with the setting `samplingMode`.

The first method is *Per-Event* (PE) sampling, where positive samples are based on event occurrences. For each event, an annotated sample is created at the time of the event. However, there are two main configuration settings that influence this behavior. `dropCloseEvents` specifies whether events that happened on the same entity should be dropped, given a distance threshold `closeEventsDist`. These settings can help, for instance, if the users want to ignore data from multiple consecutive events that perhaps occurred only due to the first event. After having created all positive samples, negative samples must be created, which is done by

randomly selecting timestamps for random entities within the observation period. There are several settings allowing more control over this second sampling procedure. `negSamplingSource` defines which random entities are eligible for creating negative samples. Three values are available: non-event entities (no events occurred during the entity’s lifetime), event entities (at least one event occurred during the entity’s lifetime) or all entities (no event-based restriction). The latter two can be specified in more detail if `minDistNegSampleToEvent` is given, which represents the minimum distance to any positive sample (cf. *Red Zone* in [Figure 4.4](#)). While the amount of positive samples is limited by the number of events, the negative sample count can be controlled with the ratio of positive to negative samples as `1 : balancingRatio` (e.g., `1 : 2` would mean that twice as many negative samples are created). Another important setting is the `balancingMode` that can either be system balancing or overall balancing, which addresses our first requirement RQM 1. The former simply treats every system individually, thus applying the balancing ratio (to create the negative samples) within each system on its own. Overall balancing, on the other hand, considers the system sizes. The balancing ratio is applied to the sum of all positive samples across all systems, and then the resulting negative samples are distributed according to the system sizes that are defined by the number of entities of the main component type. More formally, the number of negative samples (before addressing the balancing ratio) for a system sys is equal to $\frac{sys.\#mc}{\sum_{sys' \in Sys} sys'.\#mc} \cdot \sum_{sys' \in Sys} sys'.\#ps$, where Sys is the set of available systems, and $sys.\#mc$ and $sys.\#ps$ are the number of main components and number of positive samples of system sys . For example, if the balancing ratio is `1 : 1` and if there are two systems with 10 and 90 events (i.e., 100 negative samples must be created) and 20 and 60 main components each, the overall balancing mode will result in 25 negative samples ($\frac{20}{20+60} \cdot (10 + 90) = 25$) for the first system and 75 for the second ($\frac{60}{20+60} \cdot (10 + 90) = 75$), whereas the system balancing mode will yield 10 and 90 negative samples, respectively.

The second method is *Slide-Through* (ST) sampling, where samples are created using a sliding sampling point that advances through the observation period. With an optional `offset`, this sampling point is successively moved forward until the end of the inspected data period is reached, and at each step, a sample is created. The setting `stepSize` can be used to specify by how much the point advances in each step. ST requires no balancing settings since the ratio of positive to negative samples is automatically determined based on how many of the created sliding samples were classified as positive or negative. Whether a sample is labeled as positive or negative is defined by the `predictionWindowSize` pws . Assume that ST returned a sample for some entity at timestamp t_s . If at least one event occurred at this entity in the interval $[t_s, t_s + pws)$, i.e., in the prediction window, then the sample is labeled as positive, and otherwise, it is labeled as negative. In case $pws = 0$, the sample is only classified as positive if at least one event happened exactly at timestamp t_s . A positive sample is duplicated for each individual event that occurred within the prediction window in order not to discard any event data. [Figure 4.5](#) shows an example time series and the process of ST sampling. Starting at timestamp 5 (offset), we slide through the series in steps of five (step size), which results in a total of five samples for the observation period $[0, 30)$. Given a prediction window size of two, we create one positive sample at timestamp 5 because an event occurred at timestamp 6, which is inside the corresponding prediction window $[5, 5 + 2)$, and four negative samples at the remaining timestamps. Note that prediction window sizes can influence the *lead time*, which is the time between the sample timestamp and the data extraction window (the observation window; cf. [Section 4.3.2.4](#) for details). In the example, the positive sample at timestamp 5 actually has an additional lead time of one (in addition to the global lead time specified in the data extraction part; cf. [Section 4.3.2.4](#)), since the event occurred at timestamp $6 = 5 + 1$. The larger the prediction windows are, the more the lead time might be affected.

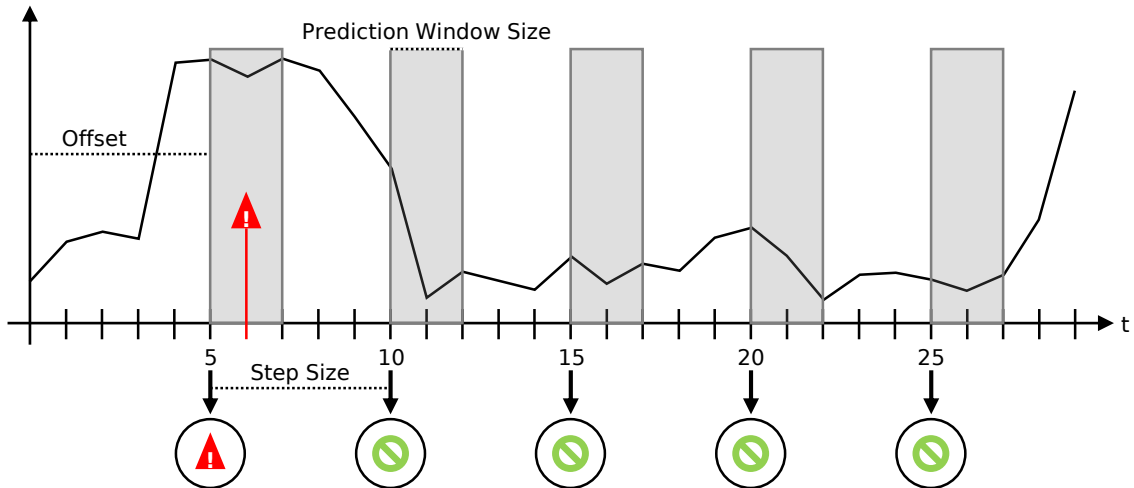


Figure 4.5: Example of the slide-through sampling mode on a time series and the resulting samples. Config settings: `offset = 5`, `stepSize = 5`, `predictionWindowSize = 2`.

4.3.2.4 Data Extraction

Until this point, no actual time series data has been extracted, which is the task of the final preprocessing step. Given the set of annotated samples S from the sampling part, the data extraction step de can be described with the function defined in [Equation 4.4](#):

$$de(S, c_{de}) \rightarrow \mathbf{F} \quad (4.4)$$

where \mathbf{F} is the set of feature vectors which are created for the annotated samples in S , i.e., $\mathbf{F} = \{\mathbf{f}_s \mid s \in S\}$, and c_{de} represents the corresponding data extraction configuration settings that can be categorized into three groups: settings controlling the *observation windows*, settings for *handling missing data* and *data augmentation* settings. There is also one setting that controls the raw data before the following data extraction, which is called `standardize`. If enabled, each time series of each individual entity is standardized (cf. [Section 2.5.1 on p. 33](#)) beforehand. This way, we ensure that the value ranges of the different entities are about the same, which is important when considering absolute time series metrics (e.g., the available memory in bytes), as these can yield drastically different values compared to relative metrics (e.g., one service could run on a host with 64GB of main memory, whereas another service could be connected to a host with only 4GB of memory).

Observation Windows The observation window settings contain all relevant information needed to extract data from a set of time series, which ultimately results in a feature vector \mathbf{f}_s for an annotated sample s . To better understand the following technical description, we provide a small example config in [Listing 4.2](#), which is visualized in [Figure 4.6](#) (details on the example are provided further below). A global setting is the `leadTime lt`, which indicates the time between where data should be extracted and the timestamp of the annotated sample, i.e., it can be used to collect data for a sample that lies lt in the future. For each time series kind that was chosen in the data selection step, an observation window⁴ can be specified with three config settings to extract the actual data from the time series of that kind. The final feature vector is then a conjunction of blocks (cf. the three blocks B-01, C-01 and D-01 of the example feature vector show in [Figure 4.6](#)), where each block contains the data of the corresponding

⁴In fact, multiple windows can be specified, however, this is a rare use case which does not occur in this thesis. Since it would needlessly complicate the following description, we decided to omit unnecessary details.

observation window, i.e., the feature vector will contain as many blocks as there are time series kinds, since one kind equates to one observation window. For a given annotated sample, the framework first automatically resolves all entities to which the sample's main component is connected (cf. *Topology Resolution* in [Figure 4.4](#) and requirement RQM 2). Afterwards, the time series are prepared for each entity, and the observation windows for the corresponding time series kinds are applied. Three config settings control the data extraction as follows:

- **size** indicates the observation window size ows for time series of the specified type, i.e., the inspected time range of the corresponding series \mathbf{x} , meaning that ows time units of \mathbf{x} should be collected. The window's start timestamp t_w is determined by its size, the timestamp t_s of the annotated sample and the lead time lt , and it is calculated as $t_w = t_s - lt - ows$. In the end, a vector \mathbf{r}_x containing ows raw values of time series \mathbf{x} is extracted using the timestamps of the window, where $\mathbf{r}_x = (x_t \mid t \in [t_w, t_w + ows]) = (r_{1_x}, \dots, r_{ows_x})$, i.e., \mathbf{r}_x is a subsequence of \mathbf{x} .
- These raw values can optionally be aggregated with the setting **aggrFuncs**, which specifies a list of aggregation functions $AF = (\text{aggr}_i \mid i \in [1, n])$, where n is the number of functions, and aggr_i can be any valid function that takes the raw time series values as input and returns a single value, such as the minimum, maximum, average, standard deviation, median, skewness, kurtosis, slope of the fitted regression line or Pearson correlation. The raw values \mathbf{r}_x are then transformed into aggregated values $\mathbf{a}_x = (\text{aggr}_i(\mathbf{r}_x) \mid \forall i \in [1, n] \wedge \text{aggr}_i \in AF) = (a_{1_x}, \dots, a_{n_x})$, which form one block of the feature vector \mathbf{f}_s for the current sample s .

This procedure is repeated for all time series kinds $K = \{k_j \mid j \in [1, m]\}$, with m being the number of kinds, until the entire feature vector is finished, i.e., all blocks have been assembled. This can be formulated as $\mathbf{f}_s = (\mathbf{a}_x^{k_1}, \dots, \mathbf{a}_x^{k_m})$, where $\mathbf{a}_x^{k_j}$ represents the aggregated values of time series \mathbf{x} of kind $k_j \in K$.

- The topological assumptions of our data state that there can be multiple entities of the same component type. If there are $e \geq 2$ such entities, this can result in e time series $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_e\}$ of the same kind k . Here, we cannot create the feature vector because \mathbf{a}_X^k is not defined. We can only compute the individual aggregated values, i.e., $\mathbf{a}_x^k \forall \mathbf{x} \in \mathbf{X}$. One idea would be to simply append these individual results to the feature vector (resulting in multiple boxes), but then, the feature vectors across all annotated samples S would no longer be equal in length (the number of entities might vary between the samples), violating our definition of a feature vector (cf. [Section 2.4.1 on p. 13](#)).

This is where the third config setting **combFuncs** comes into play, which allows us to specify a list of combination functions $CF = (\text{comb}_i \mid i \in [1, n])$, where n is the same number of functions as there are aggregation functions, and comb_i can be any valid function that takes a list of values as input and returns a single value (again, minimum, maximum, average, etc.). The number of aggregation and combination functions must be equal because there must be exactly one matching combination function for each aggregation function, i.e., $\text{aggr}_i \leftrightarrow \text{comb}_i \forall i \in [1, n]$. We can now solve our multi-time-series predicament. We create a list of aggregations \mathbf{a}_i^k by combining the i -th entries of all the individual aggregated values \mathbf{a}_x^k . Let $\mathbf{a}_x^k = (a_{1_x}^k, \dots, a_{n_x}^k)$, then $\mathbf{a}_i^k = (a_{i_x}^k \mid \forall \mathbf{x} \in \mathbf{X})$. Naturally, we obtain n such aggregation lists \mathbf{a}_i^k since we have n aggregation functions. Finally, we merge the values of every \mathbf{a}_i^k by applying the matching combination function, which results in the combined values $\mathbf{c}_X^k = (\text{comb}_i(\mathbf{a}_i^k) \mid \forall i \in [1, n] \wedge \text{comb}_i \in CF) = (c_{1_X}, \dots, c_{n_X})$ that replace our undefined \mathbf{a}_X^k . We now can combine arbitrary many time series \mathbf{X} of a certain kind k while still producing a single box (of size n) in the feature vector.

If there are multiple time series but we did not decide to specify any aggregation functions, i.e., we want to collect the raw values $\mathbf{r}_x \forall x \in \mathbf{X}$, the procedure is very similar. Analogously as before, we create a list of raw data \mathbf{r}_i^k by combining the i -th entries of all the individual raw values \mathbf{r}_x^k . Let $\mathbf{r}_x^k = (r_{1_x}^k, \dots, r_{ows_x}^k)$, then $\mathbf{r}_i^k = (r_{i_x}^k \mid \forall x \in \mathbf{X})$. Naturally, we obtain ows such raw data lists \mathbf{r}_i^k since we have ows raw values. In contrast to above, we now only have a single combination function with $CF = (\text{comb})$ that is applied to all these raw data lists. Finally, we merge the values of every \mathbf{r}_i^k by applying this combination function, which results in the combined (raw) values $\mathbf{c}_X^k = (\text{comb}(\mathbf{r}_i^k) \mid \forall i \in [1, ows]) = (c_{1_X}, \dots, c_{ows_X})$ that replace our undefined \mathbf{a}_X^k . We now can combine arbitrary many time series \mathbf{X} of a certain kind k while still producing a single box (of size ows) in the feature vector.

Example Since the above mathematical description of the data extraction with observation windows might seem complicated at first glance, we provide a small example in [Figure 4.6](#), which is based on (parts of) the topology of the system in [Figure 4.1](#). Assume that we selected A as our main component type and that we have an annotated sample s with timestamp t_s , entity $A1$ and a negative label (no event occurred at the timestamp). We now want to create the corresponding feature vector \mathbf{f}_s . The label can directly be copied from the annotated sample, whereas the actual time series data values are extracted using the config settings specified in [Listing 4.2](#).

```

...
leadTime: 5
observationWindows:
- B-01:
  size: 10
  aggrFuncs:
  - "Min"
  - "Max"
  - "Avg"
  combFuncs:
  - "Min"
  - "Max"
  - "Avg"
- C-01:
  size: 5
  aggrFuncs: []
  combFuncs:
  - "Avg"
- D-03:
  size: 15
  aggrFuncs:
  - "Slope"
  - "Correlation"
  combFuncs:
  - "Avg"
  - "Avg"
...

```

Listing 4.2: Partial YAML config file with the data extraction settings used in the example in [Figure 4.6](#).

We chose a global lead time (LT) of 5, which is thus the same for all observation windows. We further decided to inspect three time series kinds: B-01 of component type B , C-01 of component type C and D-03 of component type D . For B-01, we chose an observation window of size 10 and three aggregation functions that calculate the minimum, maximum and average of the data within this window. Since we have multiple time series of this kind, we also have

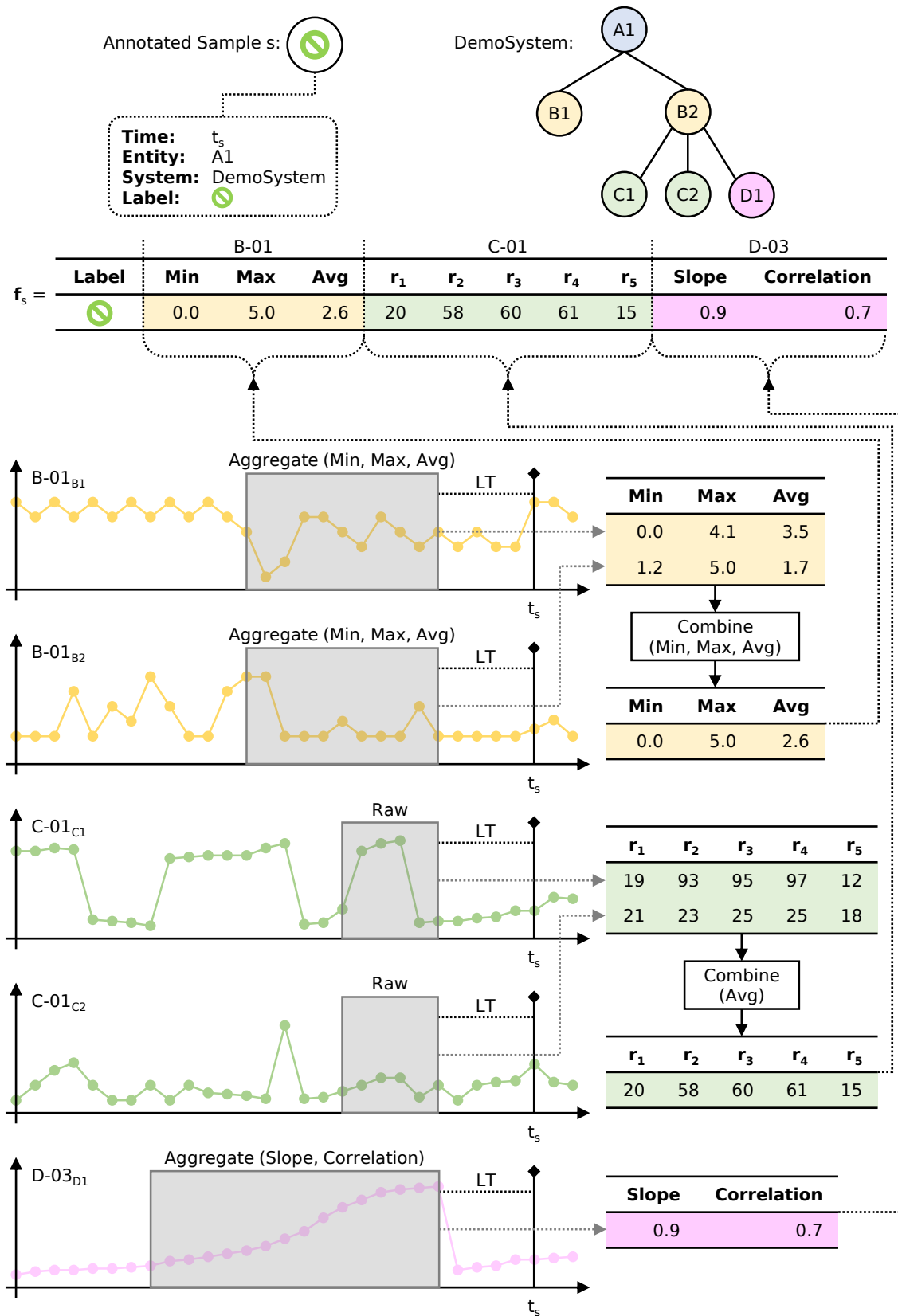


Figure 4.6: Example of how a feature vector f_s is created for an annotated sample s using different observation windows as specified in [Listing 4.2](#).

to specify equally many combinations functions. Here, we simply used the same functions, which results in the minimum of the minima, the maximum of the maxima and the average of the averages. For C-01, we selected an observation window of size 5 but no aggregation functions in order to extract the raw values. Again, there are multiple time series of kind C-01 in our example, but since we have no aggregation functions, we only need to provide a single combination function, where we opted for the average. Lastly, the observation windows of D-03 yield 15 values, which are aggregated using the slope of the regression line and the Pearson correlation. For the combination functions, we chose the average two times, meaning that we calculate the average of the slopes as well as the averages of the correlations.

First, the topology is resolved. Given the main component $A1$, we know that we have to extract data for the connected entities $B1$ and $B2$ of type B , $C1$ and $C2$ of type C and $D1$ of type D . We start with $B1$ and $B2$, for which we need to collect data for time series kind B-01, according to our config. $B-01_{B1}$ and $B-01_{B2}$ show the corresponding time series. Starting from our sample timestamp t_s , we first go back LT (lead time) steps and then extract the previous ten data values. Afterwards, we aggregate these ten values using the functions Min (minimum), Max (maximum) and Avg (average), and temporarily store the aggregated values. Since we have two components of the same type, we also have two time series of the same kind, which we need to combine using our specified combination functions (Min, Max, Avg). In the example, we thus calculate the minimum of the two minima, the maximum of the two maxima and the average of the two averages, which results in the first box of the feature vector \mathbf{f}_s (cf. $B-01$ in [Figure 4.6](#)). Analogously, we proceed for the components $C1$ and $C2$ and their corresponding time series $C-01_{C1}$ and $C-01_{C2}$. However, we can skip the aggregation part and directly carry over the raw values. Again, we have two components of the same type and therefore two time series of the same kind. In this case, we need to use our single combination function (Avg), which is applied to all raw values. In the example, we thus calculate the average of the first two raw values (r_1 of $C-01_{C1}$ and r_1 of $C-01_{C2}$), then the average of the second two raw value (r_2 of $C-01_{C1}$ and r_2 of $C-01_{C2}$), and so forth. This results in the second box of the feature vector \mathbf{f}_s (cf. $C-01$ in [Figure 4.6](#)). Finally, we need to extract data for entity $D1$, which is again analogous to above, although we do not need to use our combination functions because we only have a single component and therefore only a single time series as well. Hence, we can directly use the results of the aggregation (the slope and the correlation) and store them in the last box of the feature vector \mathbf{f}_s (cf. $D-03$ in [Figure 4.6](#)). This concludes the data extraction for the annotated sample s .

Handling Missing Data Incomplete data (cf. *NaN* (Not a Number) in [Figure 4.4](#)) is a common issue. Therefore, the framework provides several config settings that can be used to tackle such problems, thereby addressing our fourth requirement RQM 4. The first one is `missingDataMode`, which is used to handle missing data within an observation window. The following actions are all performed on the raw data $\mathbf{r} = (r_1, \dots, r_{ows})$, i.e., before the aggregation and combination steps. It can be set to either perform no action and keep the original NaN data, to drop the window entirely, to fill up missing values with the most recent non-NaN value, to fill up missing values with the nearest neighbor that is not a NaN value (in case of a tie, the previous/left neighbor is used), or to use linear interpolation for missing data. A small example of an observation window of size 7 and the effect of these different modes is shown in [Table 4.2](#).

The next setting `useNaNForMissingTS` controls how to deal with missing time series. A time series can be missing if there are no entities of the corresponding time series kind (the topology is missing components) or if all the observation windows were discarded in the previous step using the drop mode. If we disable this setting, then we simply discard the entire

	r_1	r_2	r_3	r_4	r_5	r_6	r_7
Original Data	NaN	2.0	3.0	NaN	NaN	NaN	7.0
Missing Data Mode							
No Action	NaN	2.0	3.0	NaN	NaN	NaN	7.0
Drop				\emptyset			
Most Recent	NaN	2.0	3.0	3.0	3.0	3.0	7.0
Nearest Neighbor	2.0	2.0	3.0	3.0	3.0	7.0	7.0
Linear Interpolation	NaN	2.0	3.0	4.0	5.0	6.0	7.0

Table 4.2: Example of the effect of different missing data modes.

feature vector. On the other hand, if we enable it, the corresponding box in the feature vector is filled with NaN values. In [Figure 4.7](#), we show such a case for the same config and topology introduced in [Listing 4.2](#) and [Figure 4.6](#). Assume that entity $D1$ is not available, which means that there are no components of type D anymore and, in turn, there no longer exist any time series of kind D-03 that we originally specified in our config. Enabling `useNaNForMissingTS` solves this problem by replacing the corresponding feature vector values of the box $D-03$ (slope and correlation) with NaN.

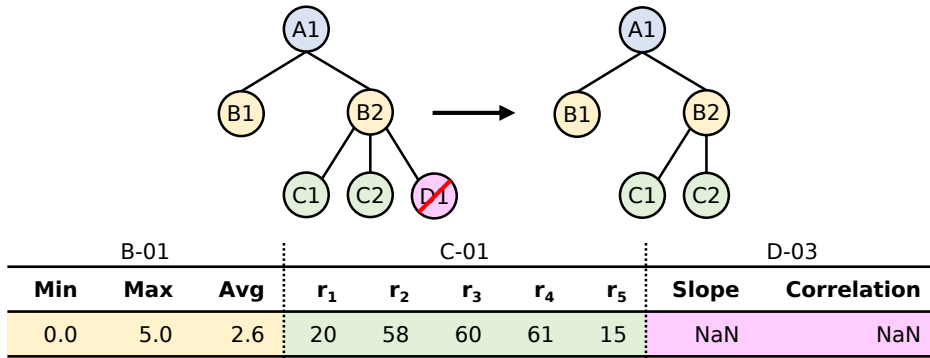


Figure 4.7: Example of missing entities and therefore missing time series, using the same example as shown in [Listing 4.2](#) and [Figure 4.6](#) but with entity $D1$ being dropped, which results in no time series of kind D-03. With the enabled configuration setting `useNaNForMissingTS`, the corresponding feature vector box $D-03$ is filled with NaN values.

Depending on the missing data mode, it might also sometimes happen that we obtain observation windows with NaN values among the raw values (cf. [Table 4.2](#)), which are then passed to the aggregation and combination functions. We can control how these NaN values should be treated with the settings `discardNaNInAggrFuncs` and `discardNaNInCombFuncs`. Both these values specify a threshold ratio below which NaN values are discarded. For example, a value of 0.3 means that NaN values are ignored as long as the amount of NaN values does not exceed 30% of the total number of values of an observation window. If there are more than 30% NaN values, no action is performed and all values (including NaN) are passed to the aggregation/combination functions. Assume that we have a window containing 100 values and ten of them are NaN (= 10%), then the NaN values are discarded (because 10% < 30%) and the remaining 90 non-NaN values are be passed to the aggregation/combination function. This can be helpful to still get meaningful aggregation or combination results rather than an all-NaN output, especially if the NaN values are scarce (which is the case when using an

appropriate missing data mode). For instance, calculating the average of 90 out of 100 values is still useful, where otherwise, we could not compute the average due to the NaN values.⁵

Data Augmentation The last part of the data extraction step shown in [Figure 4.4](#) is the optional augmentation, which can be used to create multiple feature vectors for a single annotated sample to address data imbalance by means of oversampling the minority class (cf. [Section 2.4.2 on p. 15](#)). The augmentation is performed after having processed the raw data with the selected missing data mode. The most important config setting is called `augmentations`, which allows us to specify arbitrary many augmentation functions $\text{aug} \in \text{AUGF}$ together with a number n_{aug} of how many times this function should be applied to the data of an observation window, where `aug` can be any valid function that takes the raw time series values $\mathbf{r} = (r_1, \dots, r_{\text{ows}})$ as input and returns equally many augmented values $\mathbf{r}_{\text{aug}} = (r_{1_{\text{aug}}}, \dots, r_{\text{ows}_{\text{aug}}})$. Following the same data extraction procedure as described above, each of the n_{aug} augmented values \mathbf{r}_{aug} is ultimately converted to a corresponding feature vector \mathbf{f}_{aug} , which is repeated for all selected augmentation functions $\text{aug} \in \text{AUGF}$, i.e., the augmentation of a single annotated sample results in $\sum_{\text{aug} \in \text{AUGF}} n_{\text{aug}}$ individual feature vectors, in contrast to the single feature vector when no augmentation is performed ($1 : \sum_{\text{aug} \in \text{AUGF}} n_{\text{aug}}$ vs. $1 : 1$ mapping).

Repeatedly calling an augmentation function on the same raw values might at first sound pointless, however, all the functions provided by our framework incorporate randomness to generate different augmented values every time they are invoked. We provide five functions, which we adapted from [\[182\]](#). The first one is called *jitter*, which adds random numbers drawn from a normal distribution $\mathcal{N}(0, \sigma^2)$ to the raw data values (σ can be defined by the user). The next function is *scaling*, where a single random scaling factor is drawn from a normal distribution $\mathcal{N}(1, \sigma^2)$ and each raw value is multiplied with this factor (σ can be defined by the user). The third function is called *magnitude warp*. Here, a random curve with k knots is generated with the same size as the observation window, where a knot represents a local minimum/maximum of the curve (k can be defined by the user). The knot values are randomly drawn from a normal distribution $\mathcal{N}(0, \sigma^2)$ and are evenly placed within the observation window (σ can be defined by the user). Each point of the generated curve is then added to the corresponding raw value of the observation window. The fourth function named *time warp* also utilizes a random curve (the k knot values are randomly drawn from a normal distribution $\mathcal{N}(1, \sigma^2)$, where k and σ can again be defined by the user), but in this case, the timestamps (and not the values) of the observation window are shifted according to the points of the curve. Afterwards, the new values corresponding to the original timestamps are reconstructed using linear interpolation. A small example (adapted from [\[154\]](#)) should convey the idea more clearly: Given an observation window of size 5 with raw values \mathbf{r} and timestamps \mathbf{t} as $\mathbf{r} \equiv [5 \ 7 \ 5 \ 8 \ 7]$, the timestamps are first randomly shifted based on the generated curve, resulting, for instance, in $\mathbf{t}' = [1.00 \ 2.01 \ 2.97 \ 4.02 \ 5.00]$, and then the new values \mathbf{r}_{aug} are computed for the original timestamps \mathbf{t} using linear interpolation, which yields $\mathbf{r}_{\text{aug}} \equiv [5 \ 6.98 \ 5.09 \ 7.94 \ 7]$. The last augmentation function is called *permutation*, where segments of length z , i.e., continuous parts of the observation window, are randomly permuted p times (z and p can be defined by the user). [Figure 4.8](#) shows a visual example of how these augmentation functions transform an original observation window of size 100 with $\mathbf{r} = (0, \dots, 99)$. Depending on the parameters, the augmented values \mathbf{r}_{aug} can be significantly different. The example only contains a single augmentation of every function, i.e., $n_{\text{aug}} = 1 \ \forall \text{aug} \in \text{AUGF}$. Of course, a larger n_{aug} would yield different augmented values due to the randomization.

⁵Strictly speaking, this depends on the concrete implementation of the function and whether it can handle NaN values out of the box. The Python-based scientific computation package NumPy [\[73\]](#), for example, provides both `mean` and `nanmean` functions. However, since this is implementation-specific and thus of limited interest in the context of this thesis, we will not go into further details regarding this particular subject.

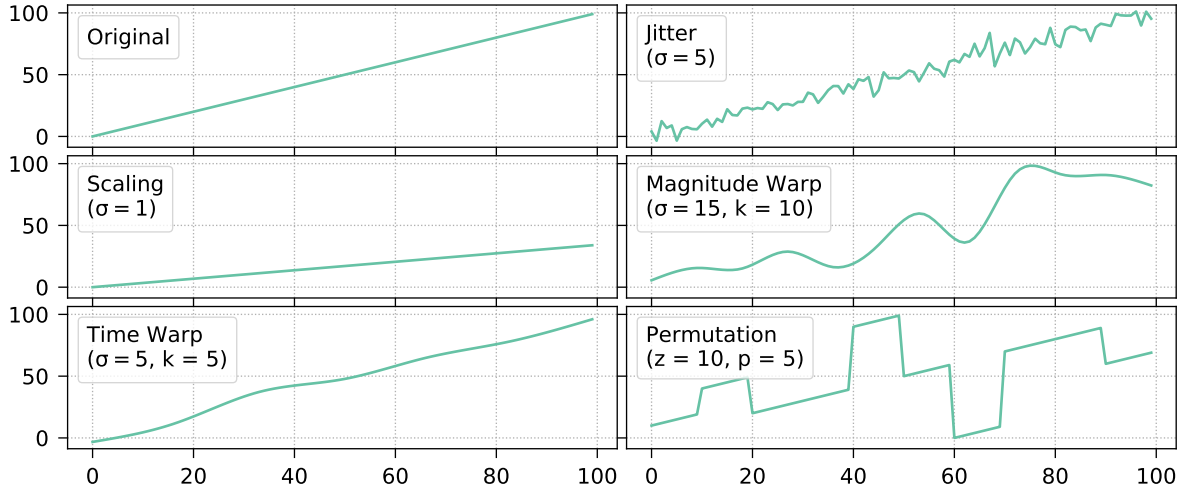


Figure 4.8: Example of the different augmentation functions applied to 100 raw data values.

Finally, two last config settings remain: `augmentPosSamples` and `augmentNegSamples`. These settings control whether we want to augment positive samples and/or negative samples, respectively. If only one of the settings is enabled, then this has a direct impact on the balancing ratio, i.e., the ratio of the number of positive samples to the number of negative samples. This can be useful if we have an unbalanced dataset and we want to apply oversampling on the minority class (cf. [Section 2.4.2 on p. 15](#)).

4.3.3 Scalability

To check whether our framework also fulfills the last requirement RQM 5, we performed an evaluation where we checked how the framework scales with an increasing number of systems and time series. To this end, we selected a system with 25 main components of type *A* that are connected to one or two entities of type *B*, which, in turn, are interlinked with one or two components of either type *C* or *D*. During the observation period of this system, 100 events occurred. We explicitly disabled any parallelization for a better comparability of the results when conducting the following two experiments:

- In the first evaluation experiment, we created up to 1000 duplicates of the systems and measured the execution time of the framework when creating feature vectors according to some given configuration settings. As main configuration settings, we decided to use per-event sampling with a balanced ratio, and we specified to extract 30 values of a single time series kind D-01, which we then aggregated using the average (single aggregation function). Due to the balanced sampling ratio, each system thus yields 100 positive + 100 negative = 200 features vectors, i.e., up to $200 \cdot 1000 = 200000$ feature vectors for the configuration with the maximum number of systems.
- The second experiment is analogous to the first, with the only difference that the number of systems remains the same (only the single system is used), but instead, the number of time series increases. Here, we created up to 1000 duplicates of the single time series kind D-01 and treated them as if they were different kinds, i.e., D-0001 up to D-1000. All other config settings are the same as before (balanced per-event sampling, average of the observation windows of size 30). Naturally, the number of feature vectors now remains the same at 200 (there is only one system), however, the length of the vectors

increases proportional to the number of time series (one additional feature vector box for every additional time series), with a maximum length of 1000.

The results of the two experiments are shown in [Figure 3.4](#), where we can see that increasing the number of systems ($\#Systems$) as well as time series ($\#Time\ Series$) both scale linearly proportional, which is a desirable property, especially for large datasets (RQM 5). We can also observe that increasing the number of systems has an overall lower effect on the run-time performance of our framework.

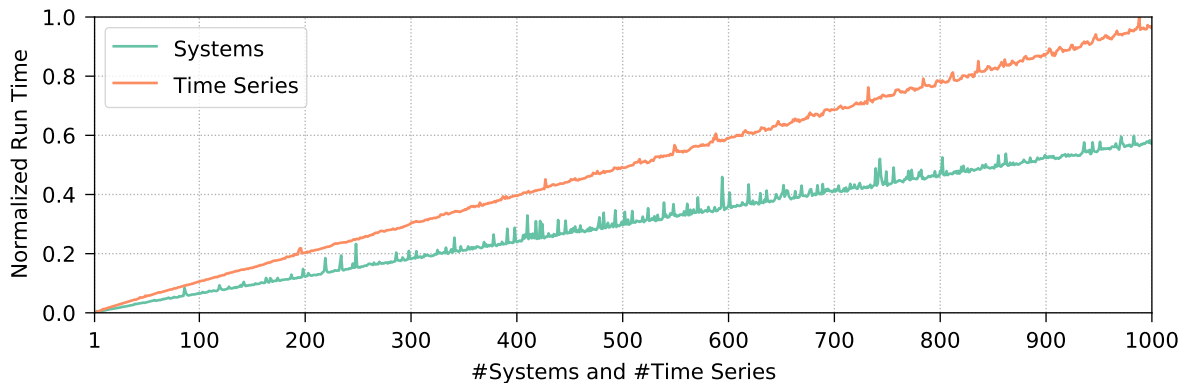


Figure 4.9: Normalized run times of the preprocessing framework when increasing the number of systems ($\#Systems$) and the number of time series ($\#Time\ Series$), revealing a linear trend in both cases. The normalization is based on the minimum and maximum run time of both measurements to allow a relative comparison between them.

4.4 Approach

Our approach for the multi-system event prediction can be summarized in two steps: data preparation and machine learning model fitting. The main idea is to create a configuration to extract multi-system time series and event data, potentially post-process the resulting feature vectors (e.g., if special treatment is necessary which cannot be accomplished with our preprocessing framework⁶) and then train various supervised machine learning models, whose objective is to find patterns, links and correlations between the time series input and the event/non-event output.

4.4.1 Data Preparation

The above introduced preprocessing framework considerably facilitates data preparation, since we no longer have to worry about the peculiarities of the multi-system environment with the topologies, events and time series. The only thing required to extract data is providing appropriate configs. However, this is an essential task, as the configuration heavily influences the resulting feature vectors, and thus, care must be taken when creating such a config. Some machine learning models and training/testing scenarios might require certain data adaptations and changes, so we can optionally post-process the CSV output generated by our framework.

⁶Of course, we could adapt our framework to also handle special cases. However, we deliberately decided against this because we wanted to keep the framework general (to a certain degree) without explicitly having to address every special case, which are often heavily dependent on the current problem, the domain and the chosen machine learning models.

4.4.2 Event Prediction

After the data preparation, the actual prediction can begin. Since event prediction is a supervised task, we require a labeled training set $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$ as well as a labeled test set $(\mathbf{X}_{\text{test}}, \mathbf{y}_{\text{test}})$, which we can achieve by either a standard train-test split or via cross-validation. Afterwards, we select the machine learning algorithm which we want to use for the subsequent experiments. This can be any supervised algorithm, such as decision trees, neural networks or random forests. For our multi-system environment and a given set of systems S , we then create three kinds of supervised machine learning models (using the selected algorithm).

Starting with the training phase, the first kind comprises single-system models M_{single} , where we only use data of the respective system for model fitting, more formally, $M_{\text{single}} = \{\text{fit}(s_{\text{train}}) \mid s \in S\}$, where s_{train} contains the training feature vectors $\mathbf{X}_{\text{train}}$ and the corresponding event labels $\mathbf{y}_{\text{train}}$ of system s . The second kind is a naive multi-system model m_{naive} , where we simply use the data of all systems for training, i.e., $m_{\text{naive}} = \text{fit}(S_{\text{train}})$, where S_{train} contains the training data merged across all systems. The third kind represents clustered multi-system models $M_{\text{clustered}}$, which are more sophisticated multi-system models. Here, we try to identify clusters $C \in \mathcal{C}$ within the available systems based on some criteria (e.g., the number of components of a system), and then we train a model for each cluster with the data of its contained systems. Formally speaking, $M_{\text{clustered}} = \{\text{fit}(C_{\text{train}}) \mid C \in \mathcal{C}\}$, where $\mathcal{C} = \{C_1, \dots, C_k \mid C_i \subset S \wedge \forall i \neq j : C_i \cap C_j = \emptyset\}$, k indicates the number of clusters, $\bigcup_{C \in \mathcal{C}} C = S$ and C_{train} contains the training data merged across all systems of cluster C .

In the testing phase, we now determine the performance of every model by extracting their predicted labels $\hat{\mathbf{y}}_s$ for every system s and then calculating appropriate evaluation metrics (cf. [Section 2.4.3.3 on p. 17](#)). For each single-system model, the test data of the corresponding system is used, i.e., $\hat{\mathbf{y}}_s = m_s.\text{predict}(s_{\text{test}}) \forall s \in S \wedge m_s \in M_{\text{single}}$, where s_{test} only contains the testing feature vectors \mathbf{X}_{test} of system s (without the true labels \mathbf{y}_{test}). Analogously, we proceed with our naive multi-system model, which results in $\hat{\mathbf{y}}_s = m_{\text{naive}}.\text{predict}(s_{\text{test}}) \forall s \in S$. The clustered multi-system models are also handled similarly. We only need to make sure that the test data of system s is passed to the clustered model that was trained with data from this system, more formally, $\hat{\mathbf{y}}_s = m_C.\text{predict}(s_{\text{test}}) \forall s \in S \wedge m_C \in M_{\text{clustered}} \wedge s \in C$. Ultimately, given $|S| = n$ systems ($|*|$ represents the set's cardinality), we thus obtain n predicted labels $\hat{\mathbf{y}}$ for each of our three kinds of machine learning models, which we can then compare to the true labels \mathbf{y}_{test} to determine the prediction quality.

4.5 Data for Evaluation

Once again, we use real-world data from our industry partner. We collect monitoring data from multiple, independent systems, which includes topologies, events and time series. Their structure is exactly as described in [Section 2.3 on p. 7](#) and we list what concrete parts thereof we need in the following:

- **Topology:** We select the service as our main component type, as we are interested in events occurring there. Services are connected to processes, but since processes do not have any time series data attached, we drop them and directly move on to the hosts on which they run (we thus have a shortened topology link $Service \rightarrow Host$ instead of $Service \rightarrow Process \rightarrow Host$). The last two component types we collect are disks and network interfaces that are both interlinked with host entities.
- **Events:** As already indicated above, we want to inspect events that occur on services. Specifically, we investigate whether a service slowdown, i.e., a performance-related event

where the average response time of a service appears to be slower than normal, can be predicted based on the time series of the connected components.

- Time series: This is where the actual data for the prediction comes from. For a given service, we collect all available time series metrics of all component types: 11 host series, 13 disk series and 10 network series, which are listed in detail in [Table 2.2 on p. 12](#). All time series are evenly spaced and available to us in one-minute resolution.

For the evaluation of our event prediction approach, we gathered monitoring data from 705 systems over the course of 20 days, ranging from 19.01.2018 09:00 UTC (Coordinated Universal Time) to 08.02.2018 09:00 UTC. During this observation period, 17733 slowdowns occurred on 2084 service entities in 434 systems, for which we present statistical insights in the following. Further details on the data exploration (e.g., the complete data overview of all 705 systems) can be found in the appendix (cf. [Section C.1 on p. 221](#)). For the visualizations, we use box plots, but we do not show outlier values in most cases to avoid overloading and skewing the plots. Instead, we provide tables that list detailed information to complement the box plots.

[Figure 4.10](#) and [Table 4.3](#) show the number of entities for each component type, averaged across the 434 systems. We can also see the total number of components (cf. column *Total*). For each of these components, we calculated the average number of connections to other entities, which is presented in [Figure 4.11](#) as well as [Table 4.4](#).

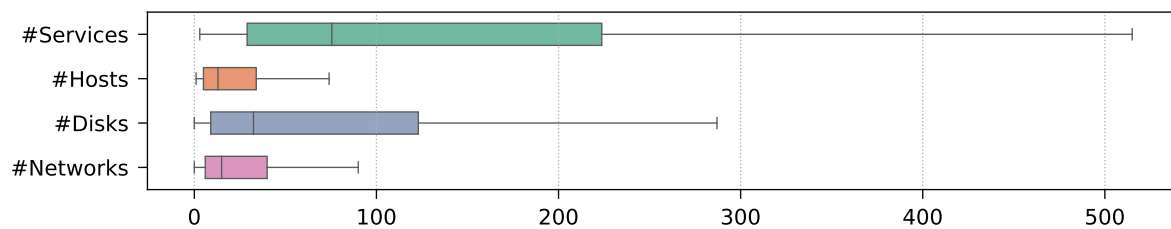


Figure 4.10: Component count statistics of the 434 systems, visualized with a box plot. Detailed information is available in [Table 4.3](#).

Component Type	Total	μ	σ	min	p_{10}	p_{25}	p_{50}	p_{75}	p_{90}	max
#Services	120528	277.71	826.85	3	13	29	75.5	223.75	434.2	11339
#Hosts	22058	50.82	157.84	1	2	5	13	34	100.6	2383
#Disks	75713	174.45	550.15	0	4	9	32.5	123	287	6025
#Networks	27921	64.33	166.14	0	2	6	15	40	144.4	1442

Table 4.3: Component count statistics of the 434 systems. μ = average, σ = standard deviation, p_i = $i\%$ percentile, min = minimum, max = maximum.

Moving on to the event statistics, [Figure 4.12](#) displays the system-averaged number of slowdowns, the number of all services, the number of services where an event occurred and the number of services where no event occurred ($\#Event\ Services + \#Non-Event\ Services = \#Services$). More details on these statistics can be looked up in [Table 4.5](#).

Finally, we present statistics on the time series data. The system-averaged available observation period $[From, To)$ for each of the 34 time series metrics is shown in [Figure 4.13](#).⁷

⁷Since the year is always the same (2018), we omit this information in all date formats.

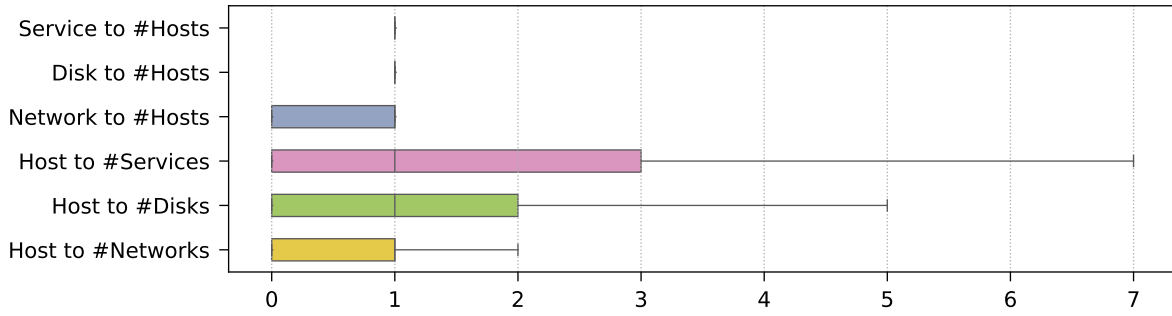


Figure 4.11: Connection count statistics of the 434 systems, visualized with a box plot. Detailed information is available in [Table 4.4](#).

Connection Type	μ	σ	min	p_{10}	p_{25}	p_{50}	p_{75}	p_{90}	max
Service to #Hosts	1.47	3.90	0	0	1	1	1	3	419
Disk to #Hosts	1	0.02	0	1	1	1	1	1	1
Network to #Hosts	0.64	0.48	0	0	0	1	1	1	1
Host to #Services	8.02	80.64	0	0	0	1	3	11	5671
Host to #Disks	3.44	38.42	0	0	0	1	2	6	2873
Host to #Networks	0.81	2.17	0	0	0	1	1	1	173

Table 4.4: Connection count statistics of the 434 systems. μ = average, σ = standard deviation, p_i = $i\%$ percentile, min = minimum, max = maximum.

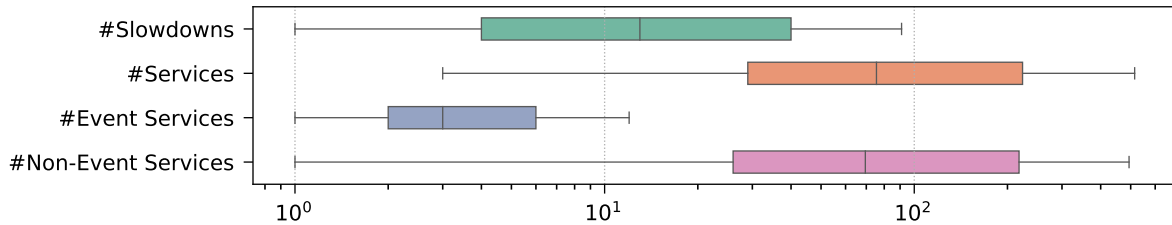


Figure 4.12: Event and corresponding component count statistics of the 434 systems where events occurred, visualized with a box plot on a logarithmic scale. Detailed information is available in [Table 4.5](#).

Counts	Total	μ	σ	min	p_{10}	p_{25}	p_{50}	p_{75}	p_{90}	max
#Slowdowns	17733	40.86	73.89	1	2	4	13	40	111.5	689
#Services	120528	277.71	826.85	3	13	29	75.5	223.75	434.2	11339
#Event Services	2084	4.80	5.79	1	1	2	3	6	10	61
#Non-Event S.	118444	272.91	825.77	1	11	26	69.5	218	415	11314

Table 4.5: Event and corresponding component count statistics of the 434 systems where events occurred. $S.$ is short for services. μ = average, σ = standard deviation, p_i = $i\%$ percentile, min = minimum, max = maximum.

We can see that the starting timestamp *From* falls indeed on the 19.01.2018 in most systems, but the end timestamp *To*, on the other hand, averages to February 7th. This is because for roughly half of the systems, we only could extract time series data until the 6th of February due to a data export error. Regarding the number of time series data points, an overview is displayed in [Figure 4.14](#), where the average number of data points per system is shown for each of the 34 time series kinds (details are listed in [Table 4.6](#)). In total, the 434 systems roughly contain twelve billion individual time series data points. Although this is a huge amount, many of the time series are incomplete due to data loss, data extraction errors or component unavailability. A detailed description of how the following time series data completeness was calculated can be found in [\[90\]](#). [Figure 4.15](#) shows the average completeness of our 34 metrics. *Normalized by Systems* means that the completeness was first calculated per system, i.e., the average component completeness within a system, and then the average was computed over all systems. *Normalized by Components* means that the completeness was calculated as the average over the completeness of all components of all systems. There is a strong trend of lower component-based completeness, indicating that there are some systems with a considerable number of components which have a low completeness score. [Figure 4.16](#) allows a more closer look into the data completeness of each of the 20 days of the export. This plot also clearly reveals missing data for the last two days, since for half of the systems, time series data was only collected up to 06.02.2018 rather than 08.02.2018 as already mentioned above. To get an even more in-depth view, we also present the time series completeness percentage of each individual system, which is shown in [Figure 4.17](#)⁸. The results are first sorted by the most complete system (top to bottom across both columns) and then by the most complete time series metric (left to right), i.e., the most upper left system has the highest data point availability (cf. system *fffa0* in [Figure 4.17a](#)), whereas the most lower right system is the least complete one (cf. system *11919* in [Figure 4.17c](#)).

Obviously, a significant amount of data is missing and incomplete, so extra care must be taken when processing and evaluating this dataset, which means creating appropriate framework configurations and applying post-processing options before training the machine learning models. Moreover, given the comparably low number of events during the entire observation period, we clearly operate on a dataset with a significant data imbalance (cf. [Section 2.4.2 on p. 15](#)). Both these matters must be addressed accordingly, where we present details in the next section.

4.6 Evaluation

With the exploratory data analysis from above in mind, simply using all 434 systems hardly makes any sense, so we first decided to only include those systems where at least 50 events occurred during the observation period. The reasoning behind this is the fact that for creating our single-system models, we can only use data from a single system. If a system does not have many events, it necessarily does not have many (positive) samples as well, which limits the learning capability of our machine learning model. Moreover, we also need to split the system data into training and testing datasets, which reduces the samples available for learning even further.⁹ Given the event statistics *#Slowdowns* in [Table 4.5](#), we can already see that we need to drop the majority of the systems to fulfill our minimum event requirement: Only 96 systems remained after this filtering step.

⁸Due to confidentiality, all systems are represented via a five-digit hash code.

⁹For instance, an 80/20 split would mean that we can use 40 of the 50 events for training and the remaining ten for testing.

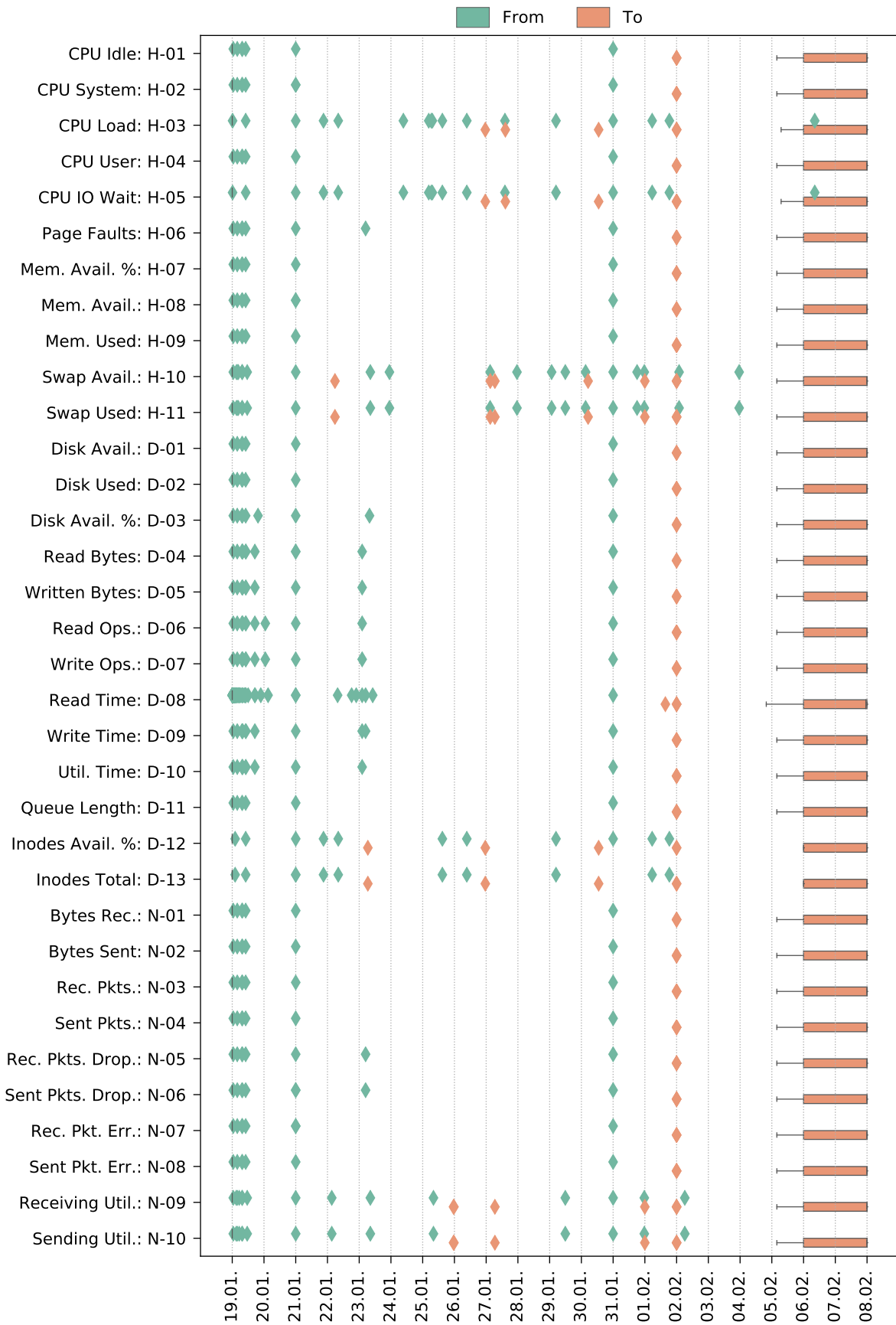


Figure 4.13: Time span statistics of the 434 systems given by $[From, To)$ markers in the format *day.month*, visualized with a box plot.

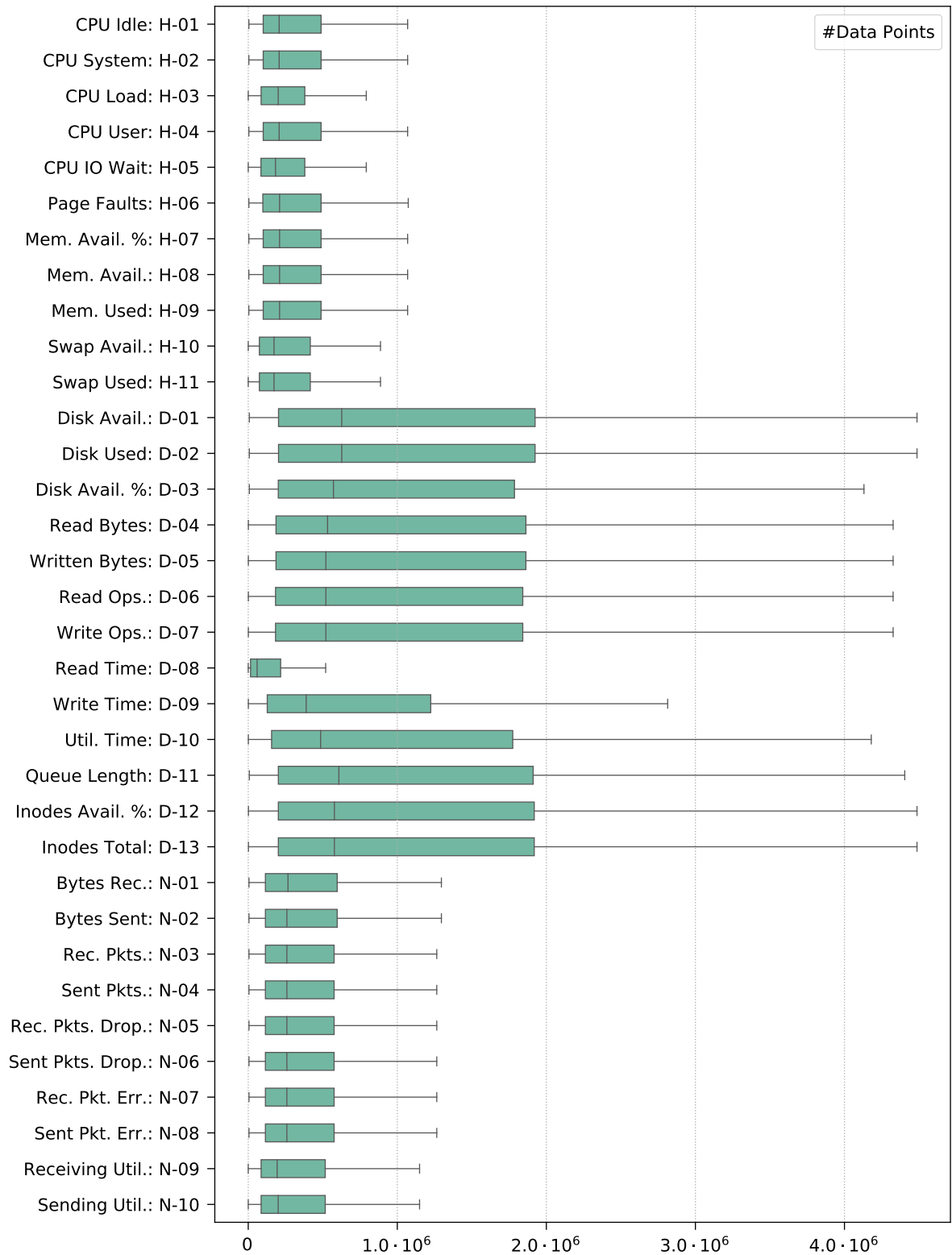


Figure 4.14: Time series data point statistics of the 434 systems, visualized with a box plot. Detailed information is available in [Table 4.6](#).

ID	#Sys.	Total	μ	σ	min	p_{50}	max
H-01	434	$189.59 \cdot 10^6$	$0.44 \cdot 10^6$	$0.61 \cdot 10^6$	4209	$0.21 \cdot 10^6$	$4.21 \cdot 10^6$
H-02	434	$189.44 \cdot 10^6$	$0.44 \cdot 10^6$	$0.61 \cdot 10^6$	4209	$0.21 \cdot 10^6$	$4.21 \cdot 10^6$
H-03	325	$123.21 \cdot 10^6$	$0.38 \cdot 10^6$	$0.57 \cdot 10^6$	21	$0.20 \cdot 10^6$	$4.11 \cdot 10^6$
H-04	434	$189.60 \cdot 10^6$	$0.44 \cdot 10^6$	$0.61 \cdot 10^6$	4209	$0.21 \cdot 10^6$	$4.21 \cdot 10^6$
H-05	325	$122.34 \cdot 10^6$	$0.38 \cdot 10^6$	$0.56 \cdot 10^6$	21	$0.18 \cdot 10^6$	$4.11 \cdot 10^6$
H-06	434	$190.37 \cdot 10^6$	$0.44 \cdot 10^6$	$0.62 \cdot 10^6$	4209	$0.21 \cdot 10^6$	$4.21 \cdot 10^6$
H-07	434	$190.55 \cdot 10^6$	$0.44 \cdot 10^6$	$0.62 \cdot 10^6$	4209	$0.21 \cdot 10^6$	$4.21 \cdot 10^6$
H-08	434	$190.44 \cdot 10^6$	$0.44 \cdot 10^6$	$0.62 \cdot 10^6$	4209	$0.21 \cdot 10^6$	$4.21 \cdot 10^6$
H-09	434	$190.22 \cdot 10^6$	$0.44 \cdot 10^6$	$0.62 \cdot 10^6$	4209	$0.21 \cdot 10^6$	$4.21 \cdot 10^6$
H-10	364	$130.19 \cdot 10^6$	$0.36 \cdot 10^6$	$0.50 \cdot 10^6$	6	$0.17 \cdot 10^6$	$3.13 \cdot 10^6$
H-11	364	$130.18 \cdot 10^6$	$0.36 \cdot 10^6$	$0.50 \cdot 10^6$	6	$0.17 \cdot 10^6$	$3.13 \cdot 10^6$
D-01	431	$760.11 \cdot 10^6$	$1.76 \cdot 10^6$	$3.07 \cdot 10^6$	7361	$0.63 \cdot 10^6$	$28.01 \cdot 10^6$
D-02	431	$757.96 \cdot 10^6$	$1.76 \cdot 10^6$	$3.04 \cdot 10^6$	7361	$0.63 \cdot 10^6$	$27.62 \cdot 10^6$
D-03	417	$686.05 \cdot 10^6$	$1.65 \cdot 10^6$	$2.97 \cdot 10^6$	7361	$0.57 \cdot 10^6$	$28.01 \cdot 10^6$
D-04	423	$708 \cdot 10^6$	$1.67 \cdot 10^6$	$3.07 \cdot 10^6$	773	$0.53 \cdot 10^6$	$28.01 \cdot 10^6$
D-05	423	$704.69 \cdot 10^6$	$1.67 \cdot 10^6$	$3.02 \cdot 10^6$	773	$0.52 \cdot 10^6$	$27.62 \cdot 10^6$
D-06	423	$704.02 \cdot 10^6$	$1.66 \cdot 10^6$	$3.07 \cdot 10^6$	773	$0.52 \cdot 10^6$	$28.02 \cdot 10^6$
D-07	423	$700.59 \cdot 10^6$	$1.66 \cdot 10^6$	$3.04 \cdot 10^6$	773	$0.52 \cdot 10^6$	$28.01 \cdot 10^6$
D-08	423	$83.18 \cdot 10^6$	$0.20 \cdot 10^6$	$0.35 \cdot 10^6$	7	59391	$2.71 \cdot 10^6$
D-09	423	$406.37 \cdot 10^6$	$0.96 \cdot 10^6$	$1.61 \cdot 10^6$	773	$0.39 \cdot 10^6$	$18.30 \cdot 10^6$
D-10	423	$670.60 \cdot 10^6$	$1.59 \cdot 10^6$	$3 \cdot 10^6$	773	$0.49 \cdot 10^6$	$27.98 \cdot 10^6$
D-11	426	$738.34 \cdot 10^6$	$1.73 \cdot 10^6$	$3.10 \cdot 10^6$	7419	$0.61 \cdot 10^6$	$28.01 \cdot 10^6$
D-12	298	$543.16 \cdot 10^6$	$1.82 \cdot 10^6$	$3.26 \cdot 10^6$	1331	$0.58 \cdot 10^6$	$27.62 \cdot 10^6$
D-13	298	$549.43 \cdot 10^6$	$1.84 \cdot 10^6$	$3.38 \cdot 10^6$	1331	$0.58 \cdot 10^6$	$27.84 \cdot 10^6$
N-01	432	$233.72 \cdot 10^6$	$0.54 \cdot 10^6$	$0.75 \cdot 10^6$	5019	$0.27 \cdot 10^6$	$5.06 \cdot 10^6$
N-02	432	$233.83 \cdot 10^6$	$0.54 \cdot 10^6$	$0.75 \cdot 10^6$	5019	$0.26 \cdot 10^6$	$5.06 \cdot 10^6$
N-03	426	$224.05 \cdot 10^6$	$0.53 \cdot 10^6$	$0.72 \cdot 10^6$	5019	$0.26 \cdot 10^6$	$5.06 \cdot 10^6$
N-04	426	$224.02 \cdot 10^6$	$0.53 \cdot 10^6$	$0.72 \cdot 10^6$	5019	$0.26 \cdot 10^6$	$5.06 \cdot 10^6$
N-05	426	$224.09 \cdot 10^6$	$0.53 \cdot 10^6$	$0.72 \cdot 10^6$	5019	$0.26 \cdot 10^6$	$5.06 \cdot 10^6$
N-06	426	$224.23 \cdot 10^6$	$0.53 \cdot 10^6$	$0.72 \cdot 10^6$	5019	$0.26 \cdot 10^6$	$5.06 \cdot 10^6$
N-07	426	$224.24 \cdot 10^6$	$0.53 \cdot 10^6$	$0.72 \cdot 10^6$	5019	$0.26 \cdot 10^6$	$5.06 \cdot 10^6$
N-08	426	$224.32 \cdot 10^6$	$0.53 \cdot 10^6$	$0.72 \cdot 10^6$	5019	$0.26 \cdot 10^6$	$4.94 \cdot 10^6$
N-09	382	$163.57 \cdot 10^6$	$0.43 \cdot 10^6$	$0.62 \cdot 10^6$	25	$0.19 \cdot 10^6$	$4.61 \cdot 10^6$
N-10	382	$163.39 \cdot 10^6$	$0.43 \cdot 10^6$	$0.62 \cdot 10^6$	25	$0.20 \cdot 10^6$	$4.61 \cdot 10^6$

Table 4.6: Time series data point statistics of the 434 systems, where $\#Sys.$ represents the actual number of systems that provide the particular metric. μ = average, σ = standard deviation, $p_i = i\%$ percentile, min = minimum, max = maximum.

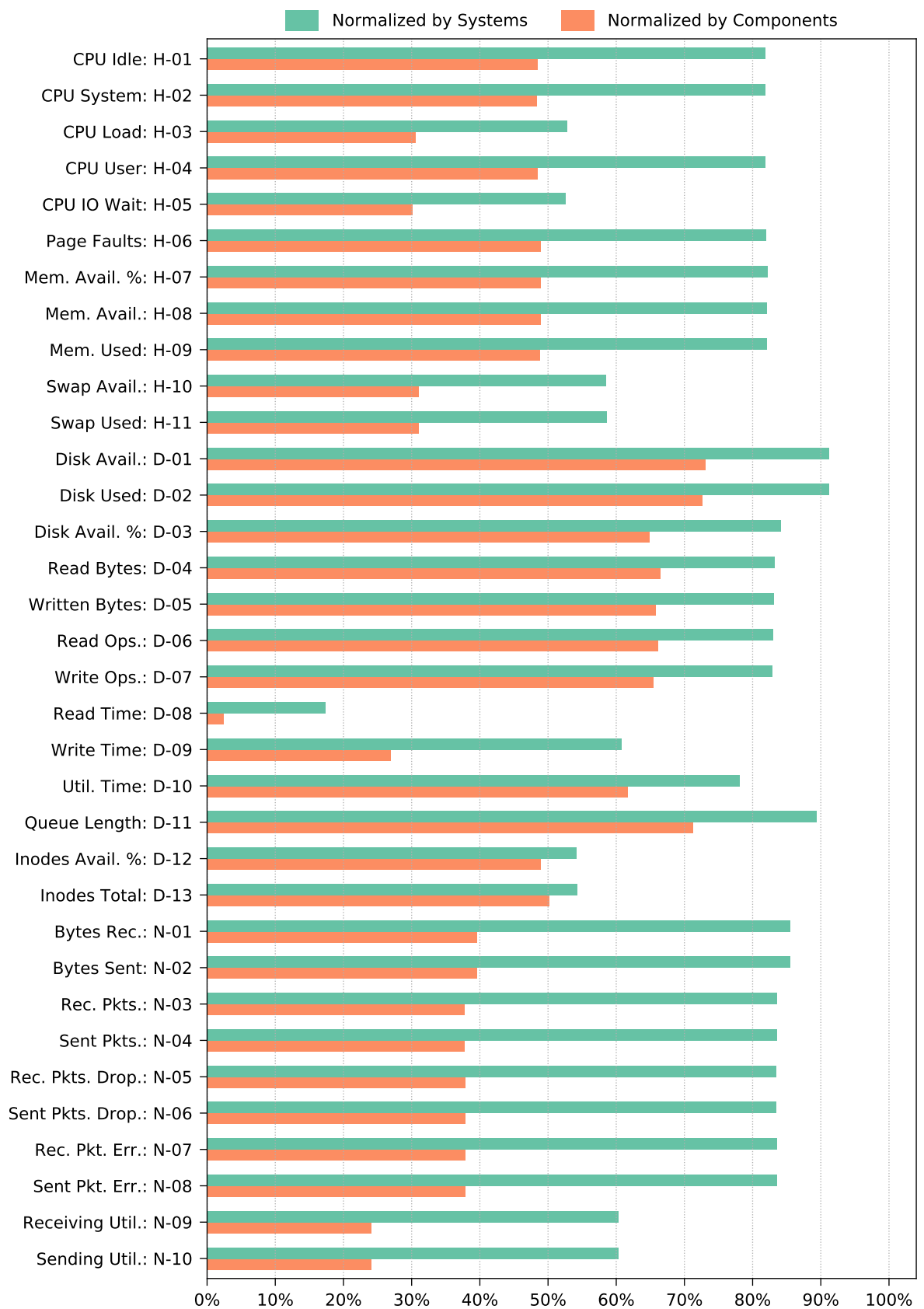


Figure 4.15: Time series data point completeness (in percent) of the 434 systems, normalized across all systems and all components, respectively.

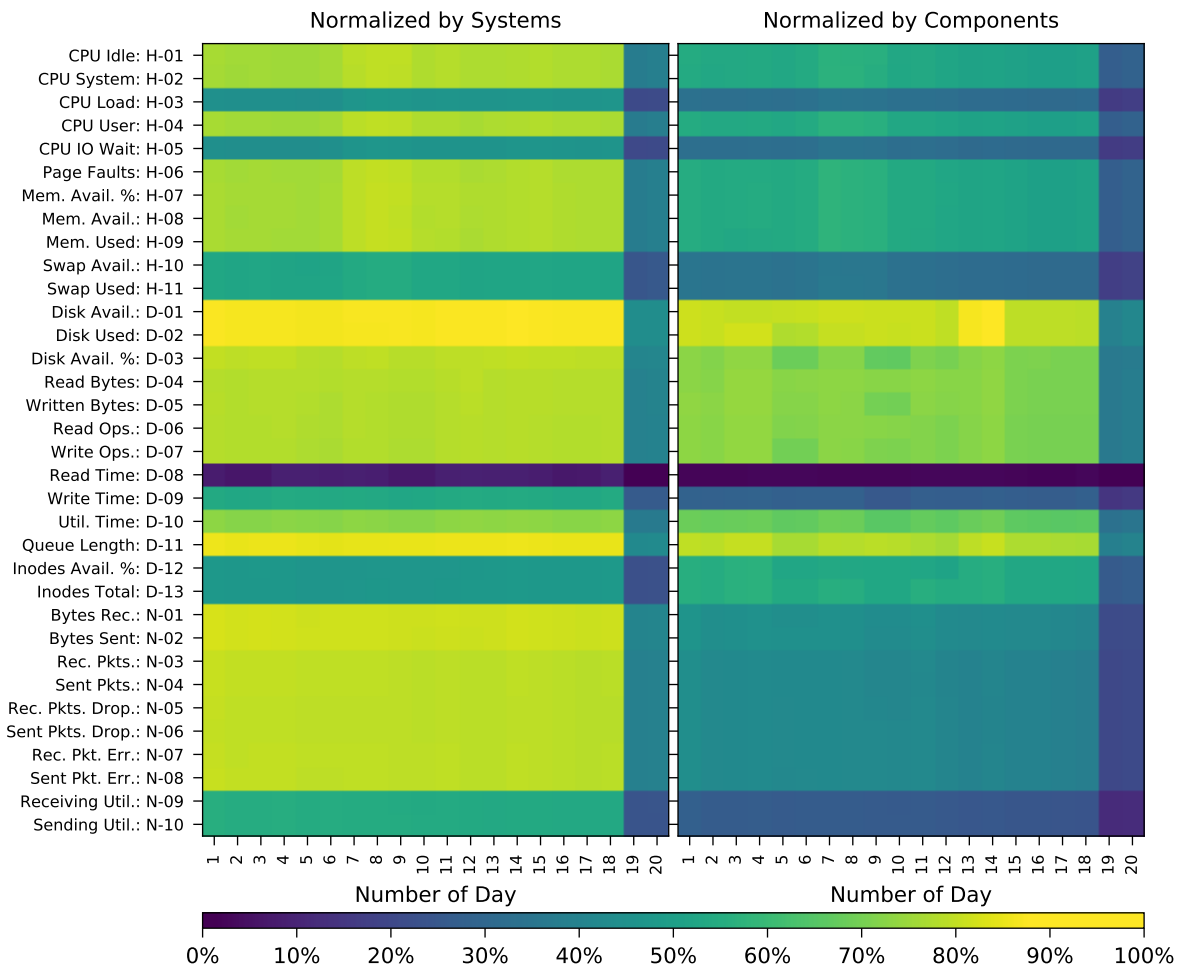
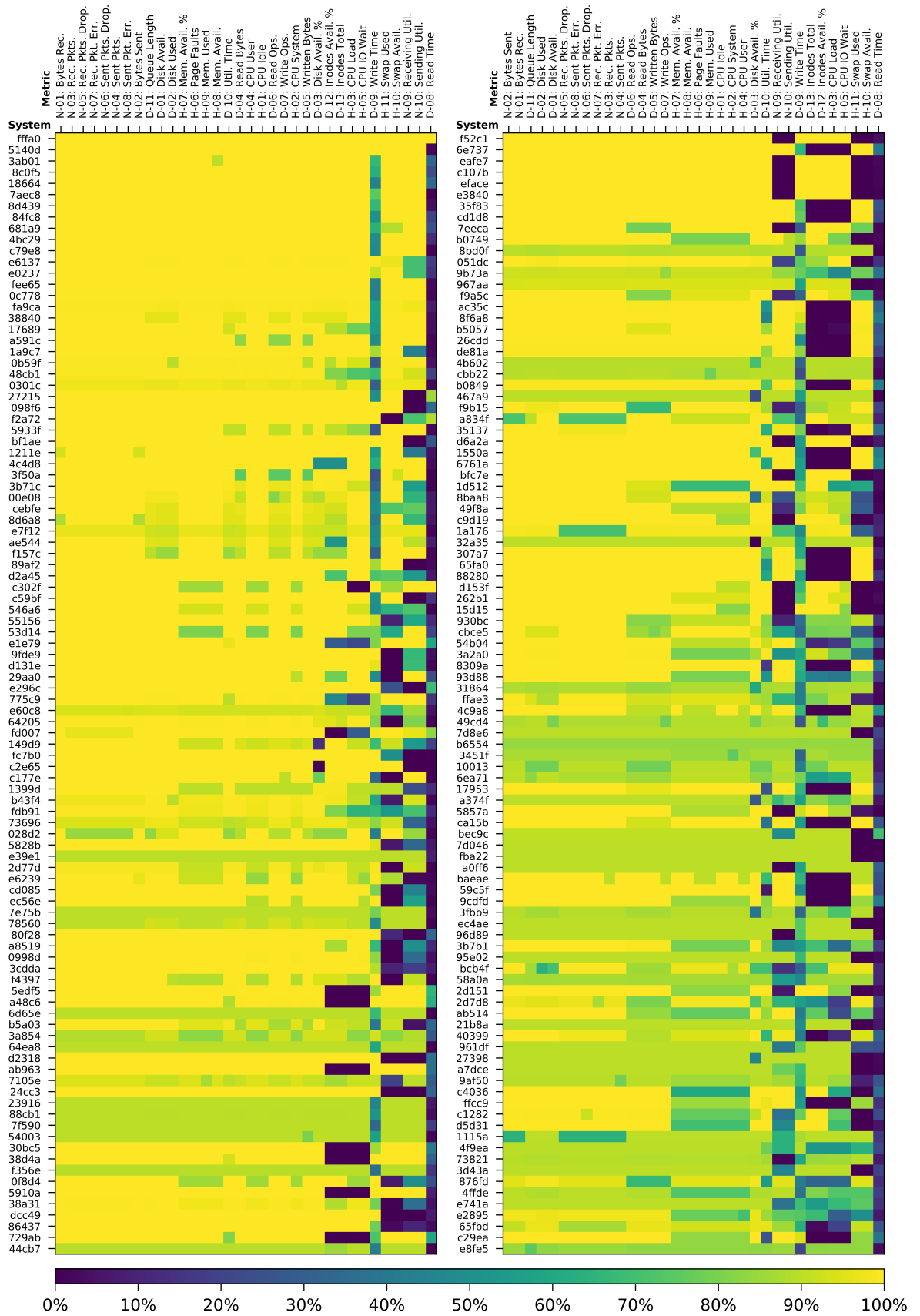
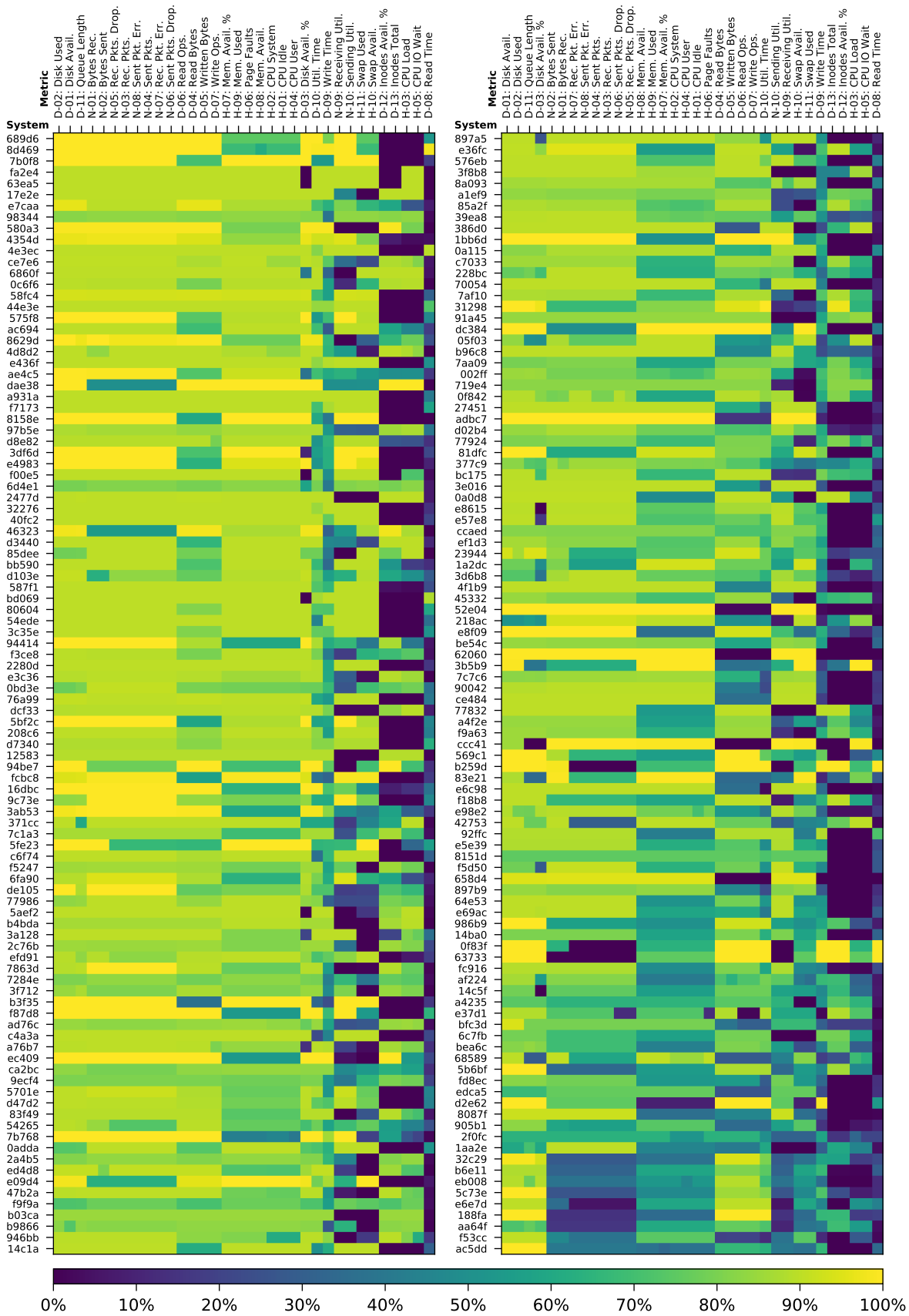


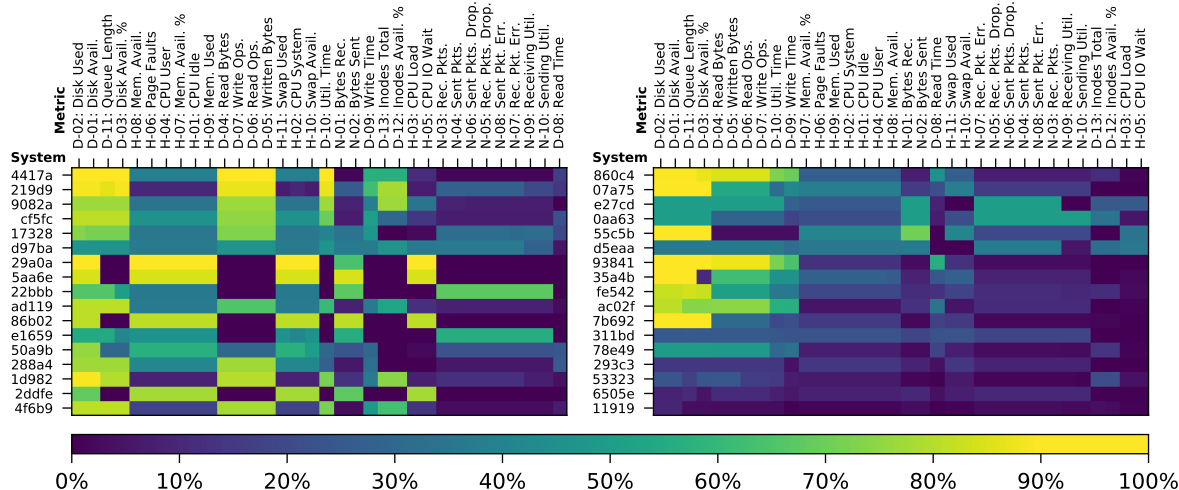
Figure 4.16: Time series data point completeness per day (in percent) of the 434 systems, normalized across all systems and all components, respectively.



(a) Time series data point completeness per system (in percent) of the 200 systems $S_1 = \{s_1, \dots, s_{200}\}$ out of all 434 systems $S = \{s_1, \dots, s_{434}\}$, i.e., $S_1 \subset S$, first sorted by the most complete system (top to bottom) and then by the most complete time series metric (left to right).



(b) Time series data point completeness per system (in percent) of the 200 systems $S_2 = \{s_{201}, \dots, s_{400}\}$ out of all 434 systems $S = \{s_1, \dots, s_{434}\}$, i.e., $S_2 \subset S$, first sorted by the most complete system (top to bottom) and then by the most complete time series metric (left to right).



(c) Time series data point completeness per system (in percent) of the 34 systems $S_3 = \{s_{401}, \dots, s_{434}\}$ out of all 434 systems $S = \{s_1, \dots, s_{434}\}$, i.e., $S_3 \subset S$, first sorted by the most complete system (top to bottom) and then by the most complete time series metric (left to right).

Figure 4.17: Time series data point completeness per system (in percent) of the 434 systems, first sorted by the most complete system (top to bottom) and then by the most complete time series metric (left to right).

4.6.1 Clustering

As introduced in the approach in [Section 4.4](#), we want to create more sophisticated multi-system prediction models based on some sort of clustering. We decided to start with a simple clustering by inspecting the entity count properties of the 96 systems, i.e., the number of services, hosts, disks and networks. For the purpose of clustering, we thus created feature vectors using these four values for each system and applied t-SNE to obtain a two-dimensional visualization of the entity counts, which is shown in [Figure 4.18](#)^[10]. We can see that clustering based on the number of services and number of disks seems to be a promising start, since high service as well as disk counts (indicated by the yellow color) form decent clusters.

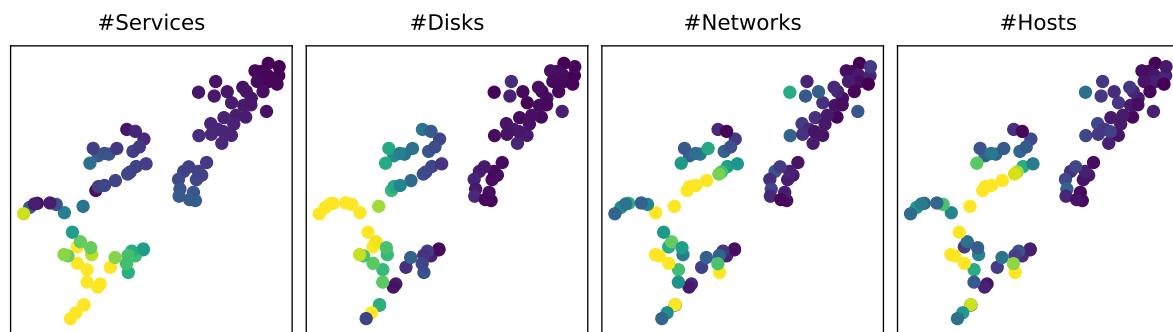


Figure 4.18: t-SNE visualization (perplexity = 20) of the entity counts of the 96 systems, tinted by the number of entities. Yellow tones indicate high entity counts, purple low entity counts.

With these observations in mind, we therefore crafted the following manual rules to extract four clusters:

¹⁰Since entity counts can have substantial outliers for very large systems, we clipped values bigger than the 90% percentile when tinting the resulting t-SNE data in order to get more usable colored plots.

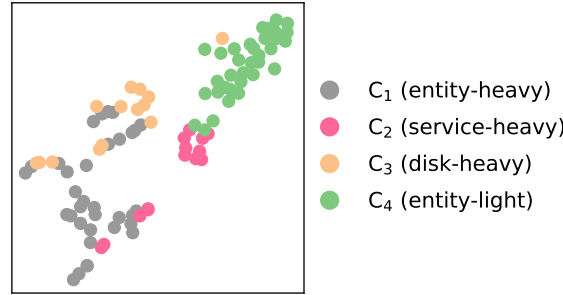


Figure 4.19: t-SNE visualization (perplexity = 20) of the entity counts of the 96 systems, tinted by the assigned cluster based on the four manual rules.

- C_1 (*entity-heavy*): The first cluster contains all systems that fulfill $s \geq \text{median}(S) \wedge d \geq \text{median}(D)$, where s is the number of services, d the number of disks, and $\text{median}(S)$ and $\text{median}(D)$ the median number of services and disks across all systems, respectively.
- C_2 (*service-heavy*): The second cluster contains all systems that fulfill $s \geq \text{median}(S) \wedge d < \text{median}(D)$.
- C_3 (*disk-heavy*): The third cluster contains all systems that fulfill $s < \text{median}(S) \wedge d \geq \text{median}(D)$.
- C_4 (*entity-light*): The fourth cluster contains the remaining systems, i.e., all systems that fulfill $s < \text{median}(S) \wedge d < \text{median}(D)$.

The system-cluster assignment of these four clusters is shown in [Figure 4.19](#). Regarding the cluster distribution, clusters C_1 and C_4 each contain 34 systems (35.4% of all 96 systems), whereas clusters C_2 and C_3 each contain 14 systems (14.6%).

4.6.2 Balanced Scenario

The first goal was to identify whether predictions in a multi-system environment can theoretically work in the first place. To this end, we created various balanced datasets (number of positive/event samples is equal to the number of negative/non-event samples). As mentioned earlier, our preprocessing framework is the main driver for creating these datasets, so we list the most important configuration settings here:

- We export data from all 96 systems over the entire observation period of 20 days, where the services are the main component type since the slowdown events occur on these entities.
- We use per-event sampling with a balanced ratio.
- For negative samples, we only select services where no event occurred.
- The balancing is applied to a per-system basis, so every system has equally many positive and negative samples.
- The missing data point mode is set to linear interpolation.
- If entire time series are missing, we use NaN values as a replacement.

- We discard NaN values in both the aggregation and combination functions with thresholds of 0.5 and 0.35, respectively, i.e., if there are fewer than 50% and 35% missing data points, the NaN values are discarded before the functions are applied. The 35% for the combination functions are chosen in a way that, for example, combining three entities where one does not have any data (all NaN) still results in a useful output.
- As an initial evaluation test, we set the lead time to 0, meaning that we want to predict events that follow right after the data in the observation windows.
- Each time series of each individual entity is standardized before extracting data with the observation windows.
- Finally, we create observation windows for all 34 time series metrics, each of which has the following eleven aggregation functions: minimum, maximum, average, standard deviation, 25% percentile, median (= 50% percentile), 75% percentile, skewness, kurtosis, slope of the fitted regression line and Pearson correlation. The corresponding combination functions are the minimum for the minimum aggregation function, the maximum for the maximum aggregation function and the median for the remaining aggregation functions. Each feature vector in the created output CSV thus has $34 \cdot 11 = 374$ entries.

In total, we created six configurations, which all have the above settings in common and only differ in the size of the observation windows, where we chose 5min, 10min, 15min, 30min, 45min and 60min (based on the suggestions from domain experts from our industry partner).

Unfortunately, our minimum events requirement does not mean that the selected systems have a high data completeness, so we had to additionally post-process the resulting CSV files returned by the framework. First, we discarded all columns, i.e., feature vector entries, which consisted of $\geq 99\%$ NaN values: 365 out of the 374 columns remained. Next, we removed all purely NaN-rows, i.e., feature vectors/samples without any actual data, and all remaining NaN values were set to 0. Since both positive and negative samples (rows) might be removed in this step, the initially balanced dataset might become unbalanced. Therefore, we re-sampled on a per-system basis, where we randomly dropped positive or negative samples (depending on which was the majority class) until the class distribution was equal again in each system. Of course, dropping positive samples can result in the fact that some systems do no longer fulfill our minimum events requirement, so we had to filter the dataset again and only keep those systems which still had at least 50 events. After this filtering step, each of the six CSV files ultimately contained 57 systems with a total of 13364^[11] balanced samples. This also affected our four system clusters: Clusters C_1 was reduced to 23 systems (40.4% of all 57 systems), cluster C_1 to 10 systems (17.5%), cluster C_2 to 11 systems (19.3%) and cluster C_4 to 13 systems (22.8%)

After the post-processing, we prepared the data for training and testing a supervised machine learning algorithm, where we opted for the default scikit-learn implementation [131] of the random forest model with 100 trees and no depth limit. We applied a randomly shuffled 80/20 train-test split on each of the six datasets, thereby ensuring that the class balance (per system) is maintained in both splits. Afterwards, we trained our naive multi-system model, the clustered multi-system models and the single-system models as follows:

- Naive multi-system model m_{naive} (short: ^[12] m): The naive model simply uses the training data of all 57 systems for model fitting. However, systems can differ in size, so we

¹¹More precisely, due to how the aggregation functions and the threshold-based discarding of NaN values work, the 45min dataset contained 13366 samples and the 60min dataset 13368 samples.

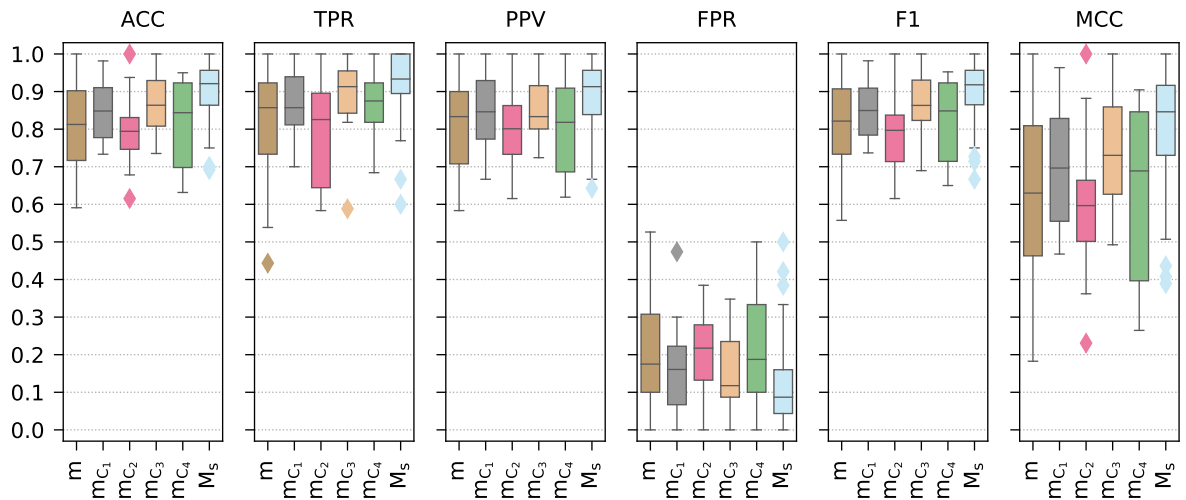
¹²The short form is used in all following figures and tables.

decided to apply a system balancing on the training set, where we randomly dropped pairs of positive and negative samples (in order to keep the class balance) until every system had equally many samples. Such a system balancing is useful to avoid that the machine learning model tries to generalize only using the data of larger systems while ignoring smaller systems (system overfitting). Of course, this reduces the available data even further. In our case, the smallest system had 50 events (50 positive samples), i.e., 100 total samples given the equal class balance. With a training split of 80% and the 57 systems, we thus got a training set size of $57 \cdot 100 \cdot 0.8 = 4560$ samples.

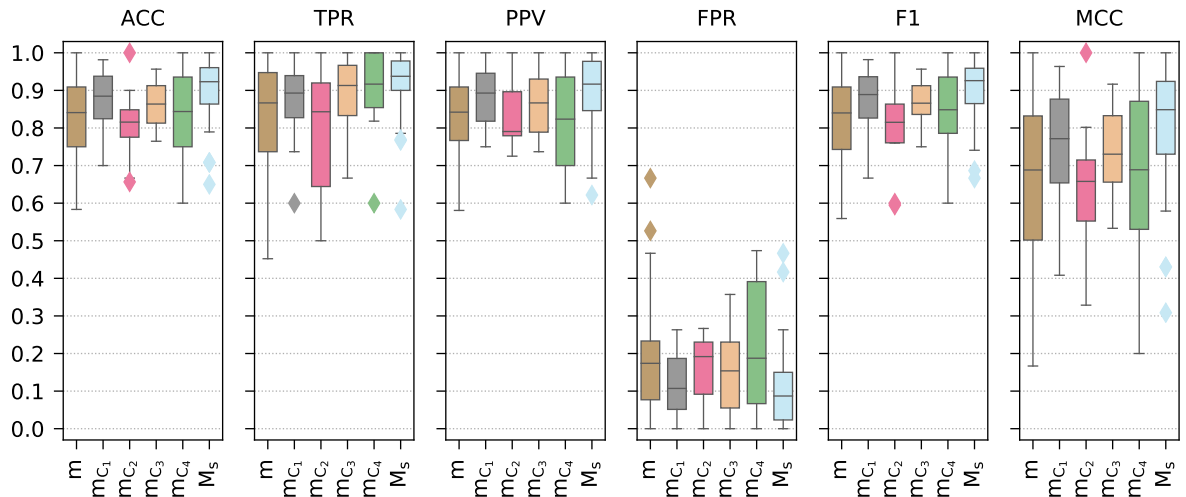
- Clustered multi-system models $m_{C_i} \in M_{\text{clustered}}$ (short: m_{C_i}): We have four clusters, so the first thing we did was to group the training data according to the system-cluster assignment. Again, to avoid system overfitting, we then applied the above mentioned system balancing on these four clustered training datasets. Cluster C_1 resulted in 1840 samples, cluster C_2 in 880 samples, cluster C_3 in 902 samples and cluster C_4 in 1040 samples. In sum, slightly more samples are available for training compared to the dataset of the naive multi-system model, which is to be expected since the system balancing is based on the system with the lowest number of samples, and this number can be higher in the different cluster datasets, meaning that in some clusters, not as many samples have to be dropped.
- Single-systems models $m_s \in M_{\text{single}}$ (short: M_s): In this case, the procedure is straightforward. There are 57 systems, so we trained 57 single-system models with their corresponding training data. In contrast to the multi-system models, no system balancing must be performed since we train each system individually, which means that system overfitting is not possible here.

In the testing phase, the test data is split into the 57 systems, and then the models which were trained with the corresponding systems are evaluated. The naive multi-system model is evaluated on the test data of all 57 systems (since it was trained on all 57 systems). The clustered multi-system models are evaluated on those test data subsets where the same systems were used for training. The single system models are evaluated on the test data of the corresponding system. The test data itself is not changed, so the naive multi-system model, the clustered multi-system models and the single-system models are evaluated on exactly the same data, which means that each of the three model kinds yields exactly 57 prediction results (one for each system). These results are presented in [Figure 4.20](#) with details listed in [Table 4.7](#), grouped by the six different observation window sizes.

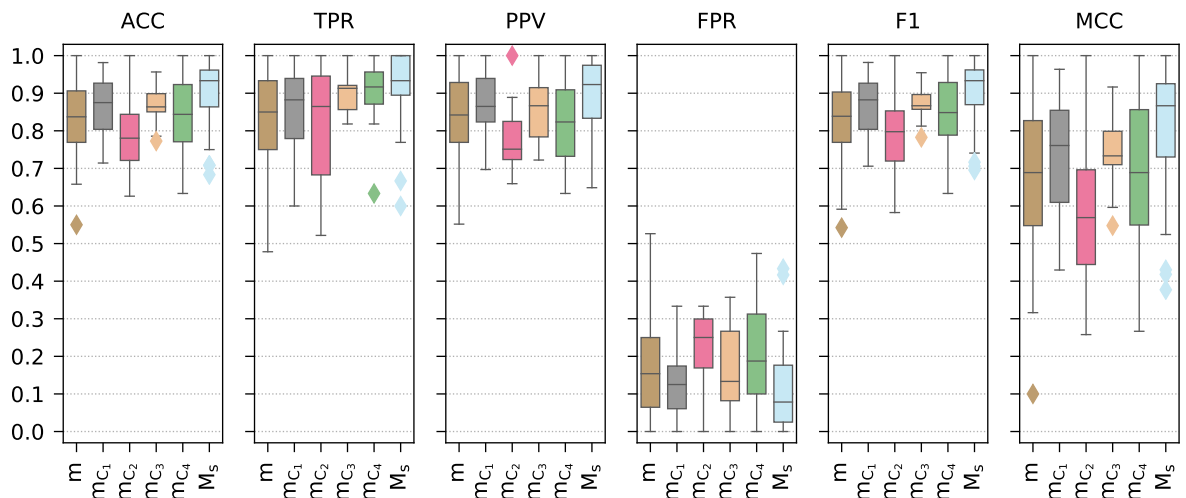
The results indicate that the single-system models M_s evidently performed the best in the majority of the cases, which is to be expected, as the data within a system tends to be more in unison compared to data of different systems. With the exception of the model m_{C_2} , we can also observe that the clustered multi-system models m_{C_i} generally performed better than the naive multi-system model m , meaning that our simple entity-based clustering did indeed have a positive effect on the prediction quality. Interestingly, the different observation window sizes do not seem to have a big impact. [Table 4.8](#) shows the evaluation metrics averaged across all tested models, revealing that the averages lie very close to each other. To determine whether these results are statistically significantly different, we conducted Wilcoxon signed-rank tests with a significance level of $\alpha = 0.01$ for all observation window combinations, i.e., comparing 5min to 10min, 5min to 15min, etc., which is shown in [Figure 4.21](#). With the exception of the 5min option, the tests confirmed that the different observation window sizes did indeed not affect the prediction quality in most cases, only 33/90 comparisons are truly different (23 of them can be attributed to the 5min window), but even then, the absolute differences are negligible (cf. [Table 4.8](#)). This might be a bit surprising at first, but a potential explanation



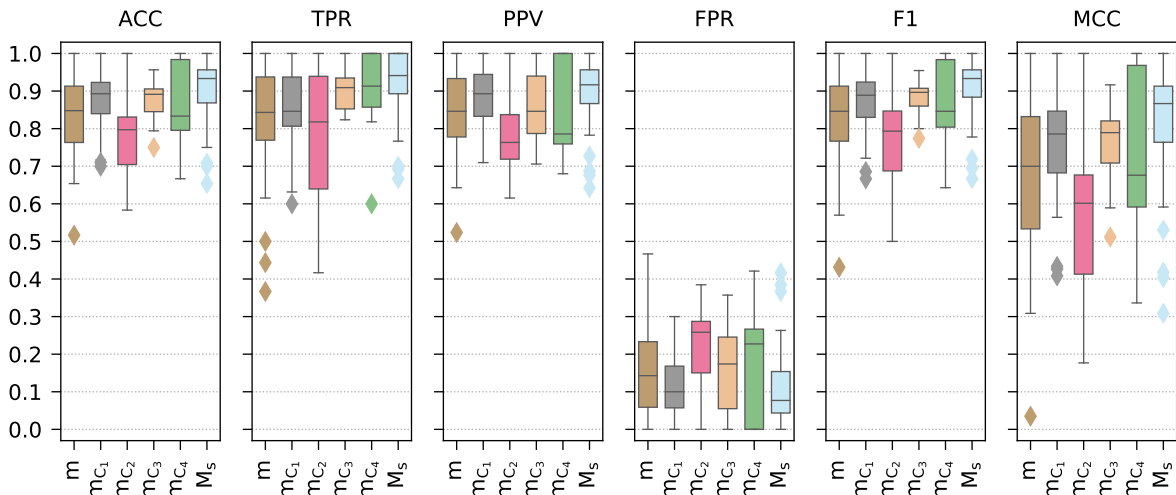
(a) Evaluation metrics for the 5min observation windows (cf. [Table 4.7a](#) for details).



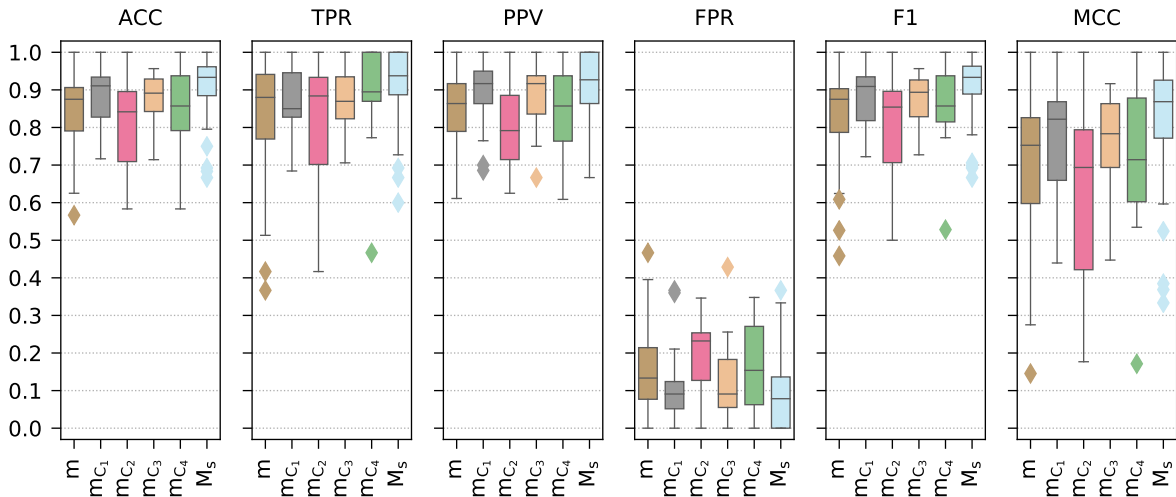
(b) Evaluation metrics for the 10min observation windows (cf. [Table 4.7b](#) for details).



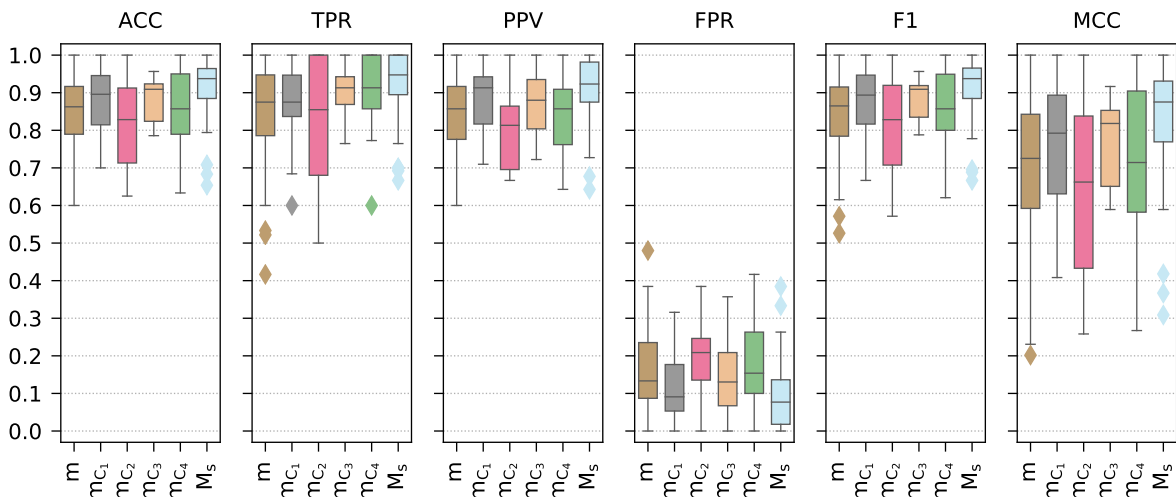
(c) Evaluation metrics for the 15min observation windows (cf. [Table 4.7c](#) for details).



(d) Evaluation metrics for the 30min observation windows (cf. [Table 4.7d](#) for details).



(e) Evaluation metrics for the 45min observation windows (cf. [Table 4.7e](#) for details).



(f) Evaluation metrics for the 60min observation windows (cf. [Table 4.7f](#) for details).

Figure 4.20: Evaluation metrics for different observation windows after running the models (m = naive multi-system model, m_{C_i} = clustered multi-system model, M_s = set of all single-system models). Detailed information is available in [Table 4.7](#).

	E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s
ACC	μ		0.82	0.85	0.80	<u>0.87</u>	0.82	0.90
	σ		<u>0.11</u>	0.08	<u>0.11</u>	0.08	0.12	0.08
	min		0.59	<u>0.73</u>	0.62	0.74	0.63	0.69
	p_{10}		0.67	0.74	0.67	<u>0.77</u>	0.67	0.78
	p_{25}		0.72	0.78	0.75	<u>0.81</u>	0.70	0.86
	p_{50}		0.81	0.85	0.79	<u>0.86</u>	0.84	0.92
	p_{75}		0.90	0.91	0.83	<u>0.93</u>	0.92	0.96
	p_{90}		0.94	0.94	0.94	<u>0.96</u>	0.94	1.00
	max		1.00	<u>0.98</u>	1.00	1.00	0.95	1.00
TPR	μ		0.82	0.87	0.80	<u>0.89</u>	0.86	0.92
	σ		0.13	0.08	0.16	<u>0.12</u>	<u>0.10</u>	0.08
	min		0.44	0.70	0.58	0.59	<u>0.68</u>	0.60
	p_{10}		0.64	<u>0.77</u>	0.59	0.82	0.71	0.82
	p_{25}		0.73	0.81	0.64	<u>0.84</u>	0.82	0.89
	p_{50}		0.86	0.86	0.83	<u>0.91</u>	0.88	0.93
	p_{75}		0.92	0.94	0.90	<u>0.96</u>	0.92	1.00
	p_{90}		1.00	0.97	1.00	1.00	<u>0.99</u>	1.00
	max		1.00	1.00	1.00	1.00	1.00	1.00
PPV	μ		0.82	0.85	0.80	<u>0.86</u>	0.80	0.89
	σ		0.12	0.09	<u>0.11</u>	0.09	0.13	0.09
	min		0.58	<u>0.67</u>	0.62	0.72	0.62	0.64
	p_{10}		0.67	0.73	0.71	<u>0.75</u>	0.64	0.76
	p_{25}		0.71	0.77	0.73	<u>0.80</u>	0.69	0.84
	p_{50}		0.83	<u>0.85</u>	0.80	0.83	0.82	0.91
	p_{75}		0.90	<u>0.93</u>	0.86	0.92	0.91	0.96
	p_{90}		1.00	<u>0.96</u>	0.90	1.00	0.93	1.00
	max		1.00	1.00	1.00	1.00	1.00	1.00
FPR	μ		0.19	0.16	0.20	<u>0.15</u>	0.23	0.12
	σ		<u>0.14</u>	0.11	0.11	0.11	0.16	0.11
	min		0.00	0.00	0.00	0.00	0.00	0.00
	p_{10}		0.00	<u>0.04</u>	0.08	0.00	0.07	0.00
	p_{25}		0.10	<u>0.07</u>	0.13	0.09	0.10	0.04
	p_{50}		0.18	0.16	0.22	<u>0.12</u>	0.19	0.09
	p_{75}		0.31	<u>0.22</u>	0.28	0.24	0.33	0.16
	p_{90}		0.38	0.28	0.31	<u>0.27</u>	0.42	0.26
	max		0.53	0.47	<u>0.38</u>	0.35	0.50	0.50
F1	μ		0.82	0.85	0.79	<u>0.87</u>	0.83	0.90
	σ		0.11	0.08	0.12	<u>0.09</u>	0.11	0.08
	min		0.56	0.74	0.62	<u>0.69</u>	0.65	0.67
	p_{10}		0.68	0.75	0.64	<u>0.78</u>	0.69	0.79
	p_{25}		0.73	0.78	0.71	<u>0.82</u>	0.71	0.86
	p_{50}		0.82	0.85	0.80	<u>0.86</u>	0.85	0.92
	p_{75}		0.91	0.91	0.84	<u>0.93</u>	0.92	0.96
	p_{90}		0.94	0.94	<u>0.95</u>	<u>0.95</u>	0.94	1.00
	max		1.00	<u>0.98</u>	1.00	1.00	0.95	1.00
MCC	μ		0.64	0.71	0.60	<u>0.74</u>	0.64	0.80
	σ		<u>0.21</u>	0.16	0.23	0.16	0.23	0.16
	min		0.18	<u>0.47</u>	0.23	0.49	0.26	0.39
	p_{10}		0.35	0.51	0.35	<u>0.55</u>	0.36	0.57
	p_{25}		0.46	0.55	0.50	<u>0.63</u>	0.40	0.73
	p_{50}		0.63	0.70	0.60	<u>0.73</u>	0.69	0.85
	p_{75}		0.81	0.83	0.66	<u>0.86</u>	0.85	0.92
	p_{90}		0.88	0.88	0.89	<u>0.92</u>	0.88	1.00
	max		1.00	<u>0.96</u>	1.00	1.00	0.90	1.00

	E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s
ACC	μ		0.83	<u>0.88</u>	0.81	0.86	0.84	0.91
	σ		0.11	<u>0.07</u>	0.10	0.06	0.13	0.08
	min		0.58	<u>0.70</u>	0.66	0.76	0.60	0.65
	p_{10}		0.68	0.79	0.67	<u>0.80</u>	0.70	0.82
	p_{25}		0.75	<u>0.82</u>	0.78	0.81	0.75	0.86
	p_{50}		0.84	<u>0.88</u>	0.82	0.86	0.84	0.92
	p_{75}		0.91	<u>0.94</u>	0.85	0.91	<u>0.94</u>	0.96
	p_{90}		0.94	0.95	0.91	0.96	<u>0.97</u>	1.00
	max		1.00	<u>0.98</u>	1.00	0.96	1.00	1.00
TPR	μ		0.84	0.87	0.78	0.88	<u>0.90</u>	0.92
	σ		0.14	<u>0.09</u>	0.19	0.11	<u>0.11</u>	0.08
	min		0.45	<u>0.60</u>	0.50	0.67	<u>0.60</u>	0.58
	p_{10}		0.60	0.75	0.50	0.71	0.82	<u>0.80</u>
	p_{25}		0.74	0.83	0.64	0.83	<u>0.85</u>	0.90
	p_{50}		0.87	0.89	0.84	0.91	<u>0.92</u>	0.94
	p_{75}		0.95	0.94	0.92	0.97	1.00	<u>0.98</u>
	p_{90}		1.00	0.96	<u>0.97</u>	1.00	1.00	1.00
	max		1.00	1.00	1.00	1.00	1.00	1.00
PPV	μ		0.83	<u>0.88</u>	0.83	0.86	0.81	0.90
	σ		0.11	0.08	0.10	<u>0.09</u>	0.14	<u>0.09</u>
	min		0.58	0.75	0.72	<u>0.74</u>	0.60	0.62
	p_{10}		0.66	<u>0.77</u>	0.75	0.75	0.65	0.79
	p_{25}		0.77	<u>0.82</u>	0.78	0.79	0.70	0.85
	p_{50}		0.84	<u>0.89</u>	0.79	0.87	0.82	0.92
	p_{75}		0.91	<u>0.95</u>	0.90	0.93	0.94	0.98
	p_{90}		0.98	0.97	1.00	0.96	<u>0.99</u>	1.00
	max		1.00	1.00	1.00	1.00	1.00	1.00
FPR	μ		0.18	<u>0.12</u>	0.16	0.15	0.22	0.11
	σ		0.14	0.08	<u>0.10</u>	0.12	0.17	<u>0.10</u>
	min		0.00	0.00	0.00	0.00	0.00	0.00
	p_{10}		0.02	0.03	0.00	0.04	<u>0.01</u>	0.00
	p_{25}		0.08	<u>0.05</u>	0.09	0.06	0.07	0.02
	p_{50}		0.17	<u>0.11</u>	0.19	0.15	0.19	0.09
	p_{75}		0.23	<u>0.19</u>	0.23	0.23	0.39	0.15
	p_{90}		0.36	<u>0.24</u>	0.25	0.30	0.41	0.22
	max		0.67	0.26	<u>0.27</u>	0.36	0.47	0.47
F1	μ		0.83	<u>0.87</u>	0.80	<u>0.87</u>	0.85	0.91
	σ		0.11	<u>0.08</u>	0.13	0.07	0.12	<u>0.08</u>
	min		0.56	<u>0.67</u>	0.59	0.75	0.60	<u>0.67</u>
	p_{10}		0.66	<u>0.79</u>	0.60	0.77	0.74	0.82
	p_{25}		0.74	0.83	0.76	<u>0.84</u>	0.79	0.86
	p_{50}		0.84	<u>0.89</u>	0.82	0.87	0.85	0.93
	p_{75}		0.91	<u>0.94</u>	0.86	0.91	<u>0.94</u>	0.96
	p_{90}		0.94	0.96	0.91	0.95	<u>0.97</u>	1.00
	max		1.00	<u>0.98</u>	1.00	0.96	1.00	1.00
MCC	μ		0.66	<u>0.75</u>	0.63	0.74	0.69	0.82
	σ		0.21	<u>0.14</u>	0.20	0.12	0.25	0.15
	min		0.17	<u>0.41</u>	0.33	0.53	0.20	0.31
	p_{10}		0.38	0.57	0.35	<u>0.62</u>	0.41	0.65
	p_{25}		0.50	0.65	0.55	<u>0.66</u>	0.53	0.73
	p_{50}		0.69	<u>0.77</u>	0.66	0.73	0.69	0.85
	p_{75}		0.83	<u>0.88</u>	0.71	0.83	0.87	0.92
	p_{90}		0.88	0.91	0.82	0.91	<u>0.94</u>	1.00
	max		1.00	<u>0.96</u>	1.00	0.92	1.00	1.00

(a) 5min observation window statistics.

(b) 10min observation window statistics.

	E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s
ACC	μ		0.83	0.86	0.79	<u>0.87</u>	0.84	0.90
	σ		0.10	<u>0.08</u>	0.11	0.06	0.11	<u>0.08</u>
	min		0.55	<u>0.71</u>	0.63	0.77	0.63	0.68
	p_{10}		0.70	0.76	0.69	<u>0.79</u>	0.72	0.81
	p_{25}		0.77	0.80	0.72	<u>0.85</u>	0.77	0.86
	p_{50}		0.84	<u>0.88</u>	0.78	0.86	0.84	0.93
	p_{75}		0.91	<u>0.93</u>	0.84	0.90	0.92	0.96
	p_{90}		0.95	0.94	0.94	0.93	<u>0.96</u>	1.00
	max		1.00	<u>0.98</u>	1.00	0.96	1.00	1.00
TPR	μ		0.83	0.86	0.81	<u>0.90</u>	<u>0.90</u>	0.92
	σ		0.13	0.10	0.17	0.06	0.10	<u>0.09</u>
	min		0.48	0.60	0.52	0.82	<u>0.63</u>	0.60
	p_{10}		0.64	0.74	0.61	<u>0.82</u>	0.83	0.80
	p_{25}		0.75	0.78	0.68	0.86	<u>0.87</u>	0.89
	p_{50}		0.85	0.88	0.86	0.91	<u>0.92</u>	0.93
	p_{75}		0.93	0.94	0.95	0.92	<u>0.96</u>	1.00
	p_{90}		1.00	<u>0.99</u>	1.00	0.96	1.00	1.00
	max		1.00	1.00	1.00	1.00	1.00	1.00
PPV	μ		0.84	<u>0.87</u>	0.78	0.86	0.82	0.90
	σ		0.11	0.08	0.10	<u>0.09</u>	0.12	<u>0.09</u>
	min		0.55	<u>0.70</u>	0.66	0.72	0.63	0.65
	p_{10}		0.70	<u>0.78</u>	0.70	0.75	0.66	0.80
	p_{25}		0.77	<u>0.82</u>	0.72	0.78	0.73	0.83
	p_{50}		0.84	0.86	0.75	<u>0.87</u>	0.82	0.92
	p_{75}		0.93	<u>0.94</u>	0.82	0.91	0.91	0.97
	p_{90}		1.00	0.96	0.90	0.95	<u>0.99</u>	1.00
	max		1.00	1.00	1.00	1.00	1.00	1.00
FPR	μ		0.17	<u>0.14</u>	0.22	0.16	0.21	0.11
	σ		0.12	0.09	0.11	0.12	0.16	<u>0.10</u>
	min		0.00	0.00	0.00	0.00	0.00	0.00
	p_{10}		0.00	0.04	0.11	0.04	<u>0.01</u>	0.00
	p_{25}		<u>0.06</u>	<u>0.06</u>	0.17	0.08	0.10	0.02
	p_{50}		0.15	<u>0.12</u>	0.25	0.13	0.19	0.08
	p_{75}		0.25	0.17	0.30	0.27	0.31	<u>0.18</u>
	p_{90}		0.33	<u>0.26</u>	0.33	0.28	0.42	0.23
	max		0.53	0.33	0.33	<u>0.36</u>	0.47	0.43
F1	μ		0.83	0.86	0.79	<u>0.87</u>	0.85	0.91
	σ		0.10	<u>0.08</u>	0.13	0.05	0.10	<u>0.08</u>
	min		0.54	<u>0.71</u>	0.58	0.78	0.63	0.70
	p_{10}		0.69	0.74	0.66	0.81	<u>0.76</u>	0.81
	p_{25}		0.77	0.80	0.72	<u>0.86</u>	0.79	0.87
	p_{50}		0.84	<u>0.88</u>	0.80	0.87	0.85	0.93
	p_{75}		0.90	<u>0.93</u>	0.85	0.90	<u>0.93</u>	0.96
	p_{90}		0.95	0.94	0.95	0.93	<u>0.96</u>	1.00
	max		1.00	<u>0.98</u>	1.00	0.95	1.00	1.00
MCC	μ		0.67	0.73	0.60	<u>0.74</u>	0.70	0.81
	σ		0.20	<u>0.15</u>	0.23	0.11	0.21	<u>0.15</u>
	min		0.10	<u>0.43</u>	0.26	0.55	0.27	0.38
	p_{10}		0.41	0.54	0.38	<u>0.60</u>	0.47	0.63
	p_{25}		0.55	0.61	0.44	<u>0.71</u>	0.55	0.73
	p_{50}		0.69	<u>0.76</u>	0.57	0.73	0.69	0.87
	p_{75}		0.83	0.85	0.70	0.80	<u>0.86</u>	0.93
	p_{90}		0.90	0.88	0.89	0.87	<u>0.93</u>	1.00
	max		1.00	<u>0.96</u>	1.00	0.92	1.00	1.00

(c) 15min observation window statistics.

	E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s
ACC	μ		0.84	<u>0.87</u>	0.78	<u>0.87</u>	0.86	0.91
	σ		0.10	0.09	0.13	0.06	0.11	<u>0.08</u>
	min		0.52	<u>0.70</u>	0.58	0.75	0.67	0.65
	p_{10}		0.72	0.73	0.61	<u>0.79</u>	0.75	0.83
	p_{25}		0.76	0.84	0.70	<u>0.85</u>	0.80	0.87
	p_{50}		0.85	<u>0.89</u>	0.80	<u>0.89</u>	0.83	0.93
	p_{75}		0.91	0.92	0.83	0.91	0.98	<u>0.96</u>
	p_{90}		0.97	0.97	0.94	0.92	1.00	<u>0.99</u>
	max		1.00	1.00	1.00	<u>0.96</u>	1.00	1.00
TPR	μ		0.83	0.85	0.79	<u>0.90</u>	<u>0.90</u>	0.92
	σ		0.14	0.11	0.20	0.06	0.11	<u>0.08</u>
	min		0.37	0.60	0.42	0.82	0.60	<u>0.67</u>
	p_{10}		0.65	0.74	0.60	0.85	<u>0.83</u>	0.82
	p_{25}		0.77	0.81	0.64	0.85	<u>0.86</u>	0.89
	p_{50}		0.84	0.85	0.82	<u>0.91</u>	<u>0.91</u>	0.94
	p_{75}		<u>0.94</u>	<u>0.94</u>	<u>0.94</u>	0.93	1.00	1.00
	p_{90}		1.00	<u>0.96</u>	1.00	1.00	1.00	1.00
	max		1.00	1.00	1.00	1.00	1.00	1.00
PPV	μ		0.85	<u>0.88</u>	0.77	0.86	0.84	0.90
	σ		0.11	0.09	0.12	<u>0.10</u>	0.12	0.09
	min		0.52	0.71	0.62	0.71	<u>0.68</u>	0.64
	p_{10}		0.71	0.75	0.62	<u>0.77</u>	0.70	0.81
	p_{25}		0.78	<u>0.83</u>	0.72	0.79	0.76	0.87
	p_{50}		0.85	<u>0.89</u>	0.76	0.85	0.79	0.92
	p_{75}		0.93	<u>0.94</u>	0.84	0.94	1.00	<u>0.96</u>
	p_{90}		1.00	1.00	<u>0.90</u>	1.00	1.00	1.00
	max		1.00	1.00	1.00	1.00	1.00	1.00
FPR	μ		0.16	<u>0.11</u>	0.22	0.16	0.18	0.10
	σ		0.12	0.09	0.11	0.12	0.14	<u>0.10</u>
	min		0.00	0.00	0.00	0.00	0.00	0.00
	p_{10}		0.00	0.00	<u>0.11</u>	0.00	0.00	0.00
	p_{25}		0.06	0.06	0.15	0.06	0.00	<u>0.04</u>
	p_{50}		0.14	<u>0.10</u>	0.26	0.17	0.23	0.08
	p_{75}		0.23	<u>0.17</u>	0.29	0.25	0.27	0.15
	p_{90}		0.33	0.21	0.32	0.27	0.30	<u>0.22</u>
	max		0.47	0.30	0.38	<u>0.36</u>	0.42	0.42
F1	μ		0.83	0.87	0.77	<u>0.88</u>	0.87	0.91
	σ		0.11	0.09	0.15	0.05	0.11	<u>0.07</u>
	min		0.43	<u>0.67</u>	0.50	0.77	0.64	<u>0.67</u>
	p_{10}		0.70	0.73	0.60	<u>0.80</u>	0.78	0.82
	p_{25}		0.77	0.83	0.69	<u>0.86</u>	0.80	0.88
	p_{50}		0.85	0.89	0.79	<u>0.90</u>	0.85	0.93
	p_{75}		0.91	0.92	0.85	0.91	0.98	<u>0.96</u>
	p_{90}		0.97	0.97	0.95	0.92	1.00	<u>0.99</u>
	max		1.00	1.00	1.00	<u>0.95</u>	1.00	1.00
MCC	μ		0.68	0.74	0.57	<u>0.76</u>	0.73	0.82
	σ		0.20	0.17	0.26	0.12	0.22	<u>0.15</u>
	min		0.03	<u>0.41</u>	0.18	0.51	0.34	0.31
	p_{10}		0.45	0.46	0.23	<u>0.59</u>	0.52	0.67
	p_{25}		0.53	0.68	0.41	<u>0.71</u>	0.59	0.76
	p_{50}		0.70	<u>0.79</u>	0.60	<u>0.79</u>	0.68	0.87
	p_{75}		0.83	0.85	0.68	0.82	0.97	<u>0.91</u>
	p_{90}		0.94	0.94	0.89	0.86	1.00	<u>0.98</u>
	max		1.00	1.00	1.00	<u>0.92</u>	1.00	1.00

(d) 30min observation window statistics.

	E	S	m	mC_1	mC_2	mC_3	mC_4	M_s		E	S	m	mC_1	mC_2	mC_3	mC_4	M_s	
ACC	μ		0.85	<u>0.88</u>	0.81	0.87	0.86	0.91	ACC	μ		0.84	<u>0.88</u>	0.81	<u>0.88</u>	0.86	0.92	
	σ		<u>0.10</u>	0.08	0.13	0.08	0.12	0.08		σ			0.10	0.09	0.13	0.06	0.11	<u>0.08</u>
	min		0.57	0.72	0.58	<u>0.71</u>	0.58	0.67		min			0.60	<u>0.70</u>	0.62	0.79	0.63	0.65
	p_{10}		0.71	0.75	0.68	<u>0.74</u>	<u>0.76</u>	0.82		p_{10}			0.71	0.73	0.65	<u>0.79</u>	0.75	0.82
	p_{25}		0.79	0.83	0.71	<u>0.84</u>	0.79	0.88		p_{25}			0.79	0.81	0.71	<u>0.82</u>	0.79	0.88
	p_{50}		0.88	<u>0.91</u>	0.84	0.89	0.86	0.93		p_{50}			0.86	0.90	0.83	<u>0.91</u>	0.86	0.94
	p_{75}		0.91	0.93	0.90	0.93	<u>0.94</u>	0.96		p_{75}			0.92	<u>0.95</u>	0.91	0.92	<u>0.95</u>	0.96
	p_{90}		0.96	0.97	0.94	0.95	<u>0.99</u>	1.00		p_{90}			0.95	0.98	0.94	0.95	<u>0.99</u>	1.00
	max		1.00	1.00	1.00	<u>0.96</u>	1.00	1.00		max			1.00	1.00	1.00	<u>0.96</u>	1.00	1.00
TPR	μ		0.84	0.87	0.82	0.87	<u>0.88</u>	0.92	TPR	μ		0.84	0.87	0.82	0.90	0.89	0.93	
	σ		0.15	0.09	0.19	<u>0.10</u>	0.14	0.09		σ			0.14	0.11	0.19	0.07	0.11	<u>0.09</u>
	min		0.37	<u>0.68</u>	0.42	0.71	0.47	0.60		min			0.42	0.60	0.50	0.76	0.60	<u>0.67</u>
	p_{10}		0.64	0.74	0.60	0.73	<u>0.79</u>	0.82		p_{10}			0.67	0.73	0.60	<u>0.80</u>	0.79	0.81
	p_{25}		0.77	0.83	0.70	0.82	<u>0.87</u>	0.89		p_{25}			0.79	0.84	0.68	<u>0.87</u>	0.86	0.89
	p_{50}		0.88	0.85	0.88	0.87	<u>0.89</u>	0.94		p_{50}			0.88	0.88	0.85	<u>0.91</u>	<u>0.91</u>	0.95
	p_{75}		0.94	<u>0.95</u>	0.93	0.93	1.00	1.00		p_{75}			<u>0.95</u>	<u>0.95</u>	1.00	0.94	1.00	1.00
	p_{90}		1.00	<u>0.99</u>	1.00	1.00	1.00	1.00		p_{90}			1.00	1.00	1.00	1.00	1.00	1.00
	max		1.00	1.00	1.00	1.00	1.00	1.00		max			1.00	1.00	1.00	1.00	1.00	1.00
PPV	μ		0.85	<u>0.90</u>	0.81	0.88	0.84	0.91	PPV	μ		0.84	<u>0.88</u>	0.80	0.87	0.84	0.91	
	σ		0.10	<u>0.09</u>	0.12	0.10	0.13	0.08		σ			0.11	<u>0.09</u>	0.11	<u>0.09</u>	0.12	0.08
	min		0.61	0.69	0.62	<u>0.67</u>	0.61	<u>0.67</u>		min			0.60	<u>0.71</u>	0.67	0.72	0.64	0.64
	p_{10}		0.72	<u>0.78</u>	0.70	0.75	0.72	0.83		p_{10}			0.70	0.74	0.67	<u>0.78</u>	0.70	0.81
	p_{25}		0.79	0.86	0.71	<u>0.84</u>	0.76	0.86		p_{25}			0.78	<u>0.82</u>	0.70	0.80	0.76	0.88
	p_{50}		0.86	<u>0.92</u>	0.79	<u>0.92</u>	0.86	0.93		p_{50}			0.86	<u>0.91</u>	0.81	0.88	0.86	0.92
	p_{75}		0.92	<u>0.95</u>	0.89	0.94	0.94	1.00		p_{75}			0.92	<u>0.94</u>	0.86	<u>0.94</u>	0.91	0.98
	p_{90}		1.00	1.00	<u>0.94</u>	1.00	1.00	1.00		p_{90}			1.00	1.00	<u>0.90</u>	1.00	1.00	1.00
	max		1.00	1.00	1.00	1.00	1.00	1.00		max			1.00	1.00	1.00	1.00	1.00	1.00
FPR	μ		0.15	<u>0.11</u>	0.19	0.13	0.17	0.09	FPR	μ		0.16	<u>0.12</u>	0.20	0.14	0.18	0.09	
	σ		<u>0.10</u>	<u>0.10</u>	0.11	0.13	0.13	0.09		σ			0.12	<u>0.10</u>	0.11	0.11	0.13	0.09
	min		0.00	0.00	0.00	0.00	0.00	0.00		min			0.00	0.00	0.00	0.00	0.00	0.00
	p_{10}		0.00	0.00	<u>0.05</u>	0.00	0.00	0.00		p_{10}			0.00	0.00	<u>0.11</u>	0.00	0.00	0.00
	p_{25}		0.08	<u>0.05</u>	0.13	0.06	0.06	0.00		p_{25}			0.09	<u>0.05</u>	0.14	0.07	0.10	0.02
	p_{50}		0.13	<u>0.09</u>	0.23	<u>0.09</u>	0.15	0.08		p_{50}			0.13	<u>0.09</u>	0.21	0.13	0.15	0.08
	p_{75}		0.21	0.12	0.25	0.18	0.27	<u>0.14</u>		p_{75}			0.24	<u>0.18</u>	0.25	0.21	0.26	0.14
	p_{90}		0.26	<u>0.20</u>	0.28	0.26	0.33	0.19		p_{90}			0.34	<u>0.28</u>	0.32	<u>0.28</u>	0.33	0.20
	max		0.47	<u>0.37</u>	0.35	0.43	0.35	<u>0.37</u>		max			0.48	0.32	0.38	<u>0.36</u>	0.42	0.38
F1	μ		0.84	<u>0.88</u>	0.81	0.87	0.86	0.91	F1	μ		0.84	0.87	0.81	<u>0.88</u>	0.86	0.92	
	σ		<u>0.11</u>	0.08	0.15	0.08	0.13	0.08		σ			0.11	0.09	0.14	0.06	0.11	<u>0.08</u>
	min		0.46	<u>0.72</u>	0.50	0.73	0.53	0.67		min			0.53	<u>0.67</u>	0.57	0.79	0.62	<u>0.67</u>
	p_{10}		0.68	0.77	0.65	0.75	<u>0.78</u>	0.82		p_{10}			0.70	0.74	0.63	<u>0.80</u>	0.78	0.82
	p_{25}		0.79	0.82	0.71	<u>0.83</u>	0.81	0.89		p_{25}			0.78	0.82	0.71	<u>0.83</u>	0.80	0.88
	p_{50}		0.88	<u>0.91</u>	0.85	0.89	0.86	0.93		p_{50}			0.86	0.89	0.83	<u>0.91</u>	0.86	0.94
	p_{75}		0.90	0.93	0.90	0.93	<u>0.94</u>	0.96		p_{75}			0.92	<u>0.95</u>	0.92	0.92	<u>0.95</u>	0.97
	p_{90}		0.96	0.97	0.95	0.95	<u>0.99</u>	1.00		p_{90}			0.96	0.98	0.95	0.95	<u>0.99</u>	1.00
	max		1.00	1.00	1.00	<u>0.96</u>	1.00	1.00		max			1.00	1.00	1.00	<u>0.96</u>	1.00	1.00
MCC	μ		0.70	<u>0.77</u>	0.63	0.75	0.72	0.83	MCC	μ		0.69	0.76	0.63	<u>0.77</u>	0.72	0.83	
	σ		0.19	<u>0.16</u>	0.26	<u>0.16</u>	0.23	0.15		σ			0.20	0.18	0.26	0.13	0.22	<u>0.16</u>
	min		0.15	<u>0.44</u>	0.18	0.45	0.17	0.33		min			0.20	<u>0.41</u>	0.26	0.59	0.27	0.31
	p_{10}		0.42	0.51	0.37	0.47	<u>0.54</u>	0.64		p_{10}			0.43	0.46	0.30	<u>0.60</u>	0.53	0.64
	p_{25}		0.60	0.66	0.42	<u>0.69</u>	0.60	0.77		p_{25}			0.59	0.63	0.43	<u>0.65</u>	0.58	0.77
	p_{50}		0.75	<u>0.82</u>	0.69	0.78	0.71	0.87		p_{50}			0.73	0.79	0.66	<u>0.82</u>	0.71	0.88
	p_{75}		0.83	0.87	0.79	0.86	<u>0.88</u>	0.93		p_{75}			0.84	0.89	0.84	0.85	<u>0.90</u>	0.93
	p_{90}		0.92	0.94	0.89	0.91	<u>0.98</u>	1.00		p_{90}			0.91	0.96	0.89	0.91	<u>0.98</u>	1.00
	max		1.00	1.00	1.00	<u>0.92</u>	1.00	1.00		max			1.00	1.00	1.00	<u>0.92</u>	1.00	1.00

(e) 45min observation window statistics.

(f) 60min observation window statistics.

Table 4.7: Statistics S of the evaluation metrics E for different observation windows after running the models (m = naive multi-system model, mC_i = clustered multi-system model, M_s = set of all single-system models). μ = average, σ = standard deviation, p_i = $i\%$ percentile, min = minimum, max = maximum. **Bold** = best results, underlined = second best results.

could be that the time series values leading up to a slowdown event are shaped in such a way that important characteristics can be extracted with all of our chosen window sizes.¹³ Moreover, this indicates that the characteristics appear to be present right up to the time of the event, since otherwise, the 5min observation window would not have performed as well.¹⁴

E	5min	10min	15min	30min	45min	60min
ACC	0.85 ± 0.10	0.86 ± 0.10	0.86 ± 0.09	0.87 ± 0.10	0.87 ± 0.10	0.87 ± 0.10
TPR	0.87 ± 0.12	0.87 ± 0.12	0.87 ± 0.12	0.87 ± 0.13	0.87 ± 0.13	0.88 ± 0.12
PPV	0.85 ± 0.11	0.86 ± 0.10	0.86 ± 0.10	0.87 ± 0.10	0.88 ± 0.10	0.87 ± 0.10
FPR	0.17 ± 0.13	0.15 ± 0.12	0.15 ± 0.12	0.14 ± 0.11	0.13 ± 0.10	0.14 ± 0.11
F1	0.85 ± 0.10	0.86 ± 0.10	0.86 ± 0.10	0.87 ± 0.10	0.87 ± 0.10	0.87 ± 0.10
MCC	0.71 ± 0.20	0.73 ± 0.19	0.73 ± 0.19	0.74 ± 0.19	0.75 ± 0.19	0.75 ± 0.19

Table 4.8: Average \pm standard deviation of the evaluation metrics E across all models, grouped by the observation windows.

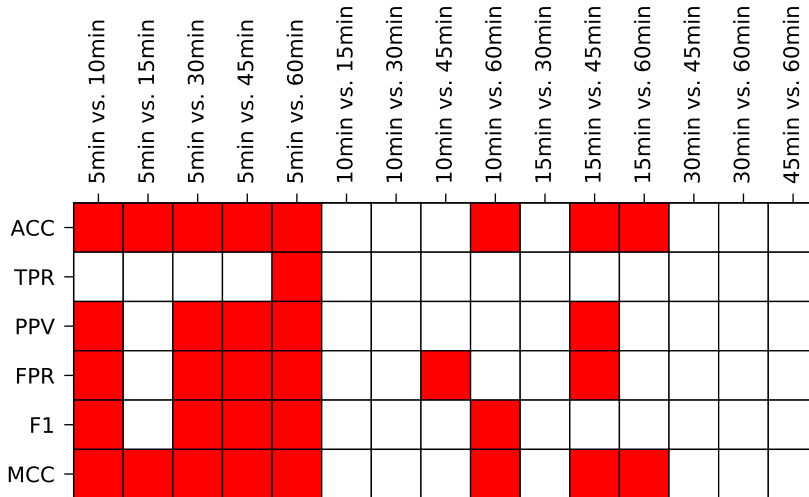


Figure 4.21: Statistical significance matrix when comparing the average evaluation metrics of the different window sizes as specified in [Table 4.8](#). Red-colored cells indicate that for window sizes $Xmin$ vs. $Ymin$, the results of window size $Xmin$ are statistically significantly worse compared to those of window size $Ymin$ (Wilcoxon signed-rank test, $\alpha = 0.01$). Worse means that all metrics except the false positive rate (FPR) are lower. For the FPR, worse means that the results are higher since the FPR is a lower-is-better metric.

With accuracies (ACC), recall (TPR), precision (PPV) and F1 scores close to 90%, Matthews correlation coefficients (MCC) around 0.75 and acceptably low false positive rates (FPR) below 15%, we concluded that event prediction in a multi-system environment does yield promising initial results. Hence, we continued with the actual scenario that is to be expected in a real-world setting, namely dealing with unbalanced data, as events can be seen as anomalies which happen much less often compared to the normal state of operation, leading to considerably more negative samples as well as new challenges.

¹³As a simple example, assume that there are large spikes in some time series just before an event occurrence, which is (sufficiently) captured by the aggregation functions such as the maximum, average or the slope of all of the chosen observation window sizes.

¹⁴The 5min observation window configuration did perform slightly worse than the other window sizes, but as mentioned above, with an average absolute difference of ≤ 0.04 , it is negligible.

4.6.3 Unbalanced Scenario

In the previous scenario, we assumed that events and non-events occurred equally frequently, which, of course, is an unrealistic assumption since service slowdown events happen much more rarely.¹⁵ Therefore, we cannot directly reuse the preprocessing framework configuration settings from before that apply per-event sampling with a balanced ratio because the test dataset would then be balanced. We thus need to separate the configurations for extracting the training data (still balanced to properly train the random forest model) from the ones for extracting the testing data (unbalanced), which, in turn, means that we need to specify the train-test split in advance. One option would be to use, for example, the first 80% of the available data for training and the following 20% for testing, but this could potentially introduce bias in case there are differences in the shapes of the time series, for instance, due to working days vs. weekends. To solve this problem, we split the entire observation period into day-sized chunks, which can then be used for a day-based cross-validation, where always one day is used for testing and all remaining days for training. As mentioned in [Section 4.5](#) and shown in [Figure 4.16](#), the data completeness is significantly lower in the last two days, so we decided to exclude them and only use the first 18 days of the export to avoid that entire day chunks only contain missing data and thus become useless for model evaluation. With the settled train-test split, we now list the configuration changes regarding the test data, the training configurations are exactly the same as introduced in the balanced scenario. Specifically, the testing configurations differ in the following:

- Instead of per-event sampling, we employ slide-through sampling, which represents the actual sampling process that can be expected in a production environment. Since this sampling procedure is applied to all service entities, it renders the specification of the negative sampling source (use only services where no event occurred) obsolete.
- Given the slide-through sampling, three more config settings must be specified: the initial sampling offset, the step size and the prediction window size. We opted for an offset of 0 to not miss any events (missing data can be discarded afterwards in the post-processing step), and a step size and prediction window size of both 60 to avoid overlaps. In a production environment, this would mean that we get data batches every 60 minutes and then have to predict whether an event occurs at some point in time within the following 60 minutes. To refer to this specific configuration, we will use the identifier ST-60-60 (slide-through sampling, 60min step size, 60min prediction window).

A keen reader might have noticed that the last setting leads to a varying lead time between 0 and 60 minutes since, given the fixed sampling point, the event might happen immediately (lead time 0) or at the very end of the prediction window (lead time 60). This makes the prediction task much more difficult, which we will see in the following evaluation results.

As the different observation window sizes from the evaluation experiments in the balanced scenario did not yield dramatically different values, we decided to start our investigation of the unbalanced test set with a 30min observation window. For comparability reasons, we used the same 57 systems.

In the post-processing phase, we applied the same steps as in the balanced setting for the training data, meaning that we discarded columns (feature vector entries) and rows (feature

¹⁵From [Table 4.5](#) we can already see that in 90% of all systems, there are only ten services where events occurred, compared to the 415 non-event services, and on these few event-services, only roughly 110 events happened over the course of our 20 days' worth of export data.

vectors/samples) based on the number of NaN values, and we re-sampled the data to achieve a system-wise class balance again. The post-processing of the testing data is analogous, except for the class balancing, which we skipped since we need to preserve the class imbalance to appropriately model a real-world scenario.

The training and testing phase differs because we now have separate datasets. We ran an 18-fold day-based cross-validation as illustrated in [Figure 4.22](#), meaning that we have 18 individual evaluations (18 iterations it_i), where always one day is used for testing and the remaining 17 are used for training the machine learning model. We again used a random forest classifier with 100 trees and no depth limit, and evaluated our three different model kinds, namely the naive multi-system model m , the clustered multi-system models m_{C_i} and the set of single-system models M_s . After running the cross-validation for each model kind, we grouped the prediction results by the systems, yielding 18 confusion matrices for each system. We then summed the corresponding entries of these 18 matrices to obtain single confusion matrix storing the result for a system just like in the balanced scenario above, which allows us to calculate the various evaluation metrics on a per-system basis. The results are presented in [Figure 4.23](#) with details listed in [Table 4.9](#).

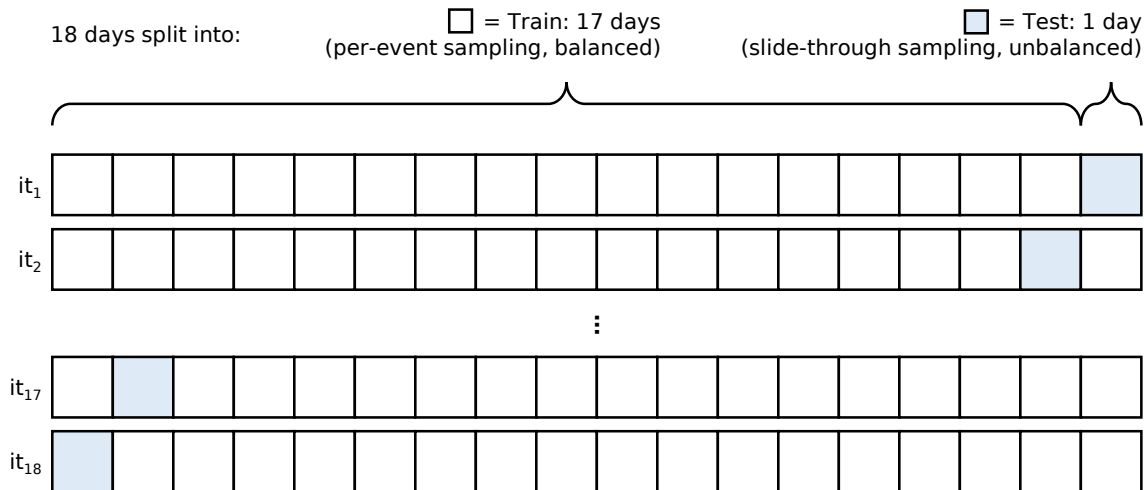


Figure 4.22: Illustration of the 18-fold day-based cross-validation.

It is obvious that the results from the real-world evaluation are significantly worse when compared to the balanced scenario. Since we are dealing with unbalanced data, metrics such as the accuracy (ACC) or true positive rate (TPR) can be misleading and should not be viewed individually but only in conjunction with other metrics. For instance, we might conclude that the single-system models M_s appear to be reasonably promising judging by the ACC and TPR with median values of 87% and 64%, respectively. However, when also looking at, e.g., the positive predictive values (precision; PPV), we immediately see that these models did not perform very well with a median value of only 2%. If we are interested in a single descriptive metric, then the Matthews Correlation Coefficient (MCC) is the most appropriate one, as it is robust against data imbalance. Unfortunately, not a single model performed well enough to be considered usable in practice. Both the median and average MCCs of each of the evaluated models are all ≤ 0.13 , with an average MCC of about 0.09 when merged across all models, which is only slightly better than a purely random prediction. The clustered multi-system models m_{C_3} and m_{C_4} , and the single-system models appear to be the best among the different model kinds, however, the absolute values are still far too low.

As hinted in the slide-through sampling configuration settings above, one problem could be that we have a prediction window size of 60min, resulting in varying lead times (cf.

E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s
ACC	μ	0.83	0.88	0.80	<u>0.85</u>	0.79	0.84
	σ	0.09	0.06	0.10	<u>0.08</u>	0.11	0.11
	min	<u>0.64</u>	0.73	0.59	0.73	0.62	0.49
	p_{10}	0.71	0.80	0.71	<u>0.76</u>	0.66	0.69
	p_{25}	0.77	0.84	0.75	<u>0.79</u>	0.70	<u>0.79</u>
	p_{50}	0.85	0.90	0.80	0.83	0.79	<u>0.87</u>
	p_{75}	0.90	<u>0.92</u>	0.84	0.93	0.90	0.93
	p_{90}	<u>0.92</u>	0.94	0.89	0.94	0.91	0.94
	max	<u>0.97</u>	<u>0.97</u>	0.96	0.96	0.94	0.99
TPR	μ	0.48	0.52	0.41	<u>0.56</u>	0.52	0.57
	σ	0.23	<u>0.21</u>	0.17	0.22	0.24	0.27
	min	0.02	<u>0.16</u>	0.19	0.19	<u>0.16</u>	0.00
	p_{10}	0.19	<u>0.28</u>	0.27	0.29	0.23	0.18
	p_{25}	0.31	<u>0.38</u>	0.30	0.44	<u>0.38</u>	0.35
	p_{50}	0.47	<u>0.53</u>	0.40	0.51	0.51	0.64
	p_{75}	0.63	0.67	0.46	0.71	<u>0.76</u>	0.77
	p_{90}	<u>0.80</u>	0.73	0.58	0.77	<u>0.80</u>	0.89
	max	<u>0.99</u>	1.00	0.77	0.95	0.86	1.00
PPV	μ	0.04	0.03	0.01	<u>0.05</u>	0.09	<u>0.05</u>
	σ	0.07	0.09	0.01	<u>0.03</u>	0.09	0.11
	min	<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.01	<u>0.00</u>
	p_{10}	0.00	0.00	0.00	<u>0.01</u>	0.02	0.00
	p_{25}	0.00	0.00	0.00	<u>0.03</u>	0.04	0.01
	p_{50}	0.01	0.01	0.01	0.04	0.04	<u>0.02</u>
	p_{75}	0.04	0.02	0.01	<u>0.07</u>	0.11	0.06
	p_{90}	0.09	0.05	0.01	0.09	0.23	<u>0.11</u>
	max	0.37	<u>0.43</u>	0.02	0.10	0.27	0.73
FPR	μ	0.16	0.12	0.20	<u>0.15</u>	0.20	0.16
	σ	<u>0.08</u>	0.06	0.10	<u>0.08</u>	0.11	0.11
	min	<u>0.03</u>	<u>0.03</u>	0.04	0.04	0.05	0.01
	p_{10}	<u>0.07</u>	0.06	0.11	0.06	0.09	0.06
	p_{25}	0.09	<u>0.08</u>	0.16	0.07	0.09	0.07
	p_{50}	0.15	0.10	0.20	0.17	0.21	<u>0.13</u>
	p_{75}	0.23	0.16	0.25	0.21	0.29	<u>0.19</u>
	p_{90}	0.28	0.20	0.29	<u>0.24</u>	0.32	0.32
	max	<u>0.36</u>	0.27	0.41	0.27	0.39	0.52
F1	μ	0.06	0.05	0.01	<u>0.09</u>	0.13	0.08
	σ	0.10	0.12	0.01	<u>0.06</u>	0.11	0.13
	min	<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.01	0.01	<u>0.00</u>
	p_{10}	0.00	0.00	0.00	<u>0.01</u>	0.03	0.00
	p_{25}	0.01	0.01	0.01	<u>0.05</u>	0.07	0.01
	p_{50}	0.02	0.01	0.01	<u>0.07</u>	0.08	0.03
	p_{75}	0.08	0.04	0.02	<u>0.12</u>	0.19	0.11
	p_{90}	0.15	0.10	0.03	0.16	0.31	<u>0.19</u>
	max	0.53	<u>0.60</u>	0.04	0.17	0.33	0.84
MCC	μ	0.08	0.09	0.03	0.13	<u>0.12</u>	0.11
	σ	0.10	0.13	0.03	0.09	<u>0.08</u>	0.14
	min	-0.03	<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.02	-0.02
	p_{10}	<u>0.01</u>	<u>0.01</u>	0.00	0.03	0.03	0.00
	p_{25}	0.02	0.03	0.01	0.06	<u>0.04</u>	0.02
	p_{50}	0.04	0.04	0.02	0.11	0.11	<u>0.07</u>
	p_{75}	0.12	0.10	0.04	0.18	<u>0.17</u>	0.18
	p_{90}	0.18	0.18	0.06	0.26	0.23	<u>0.24</u>
	max	0.59	<u>0.64</u>	0.11	0.28	0.26	0.85

Table 4.9: Statistics S of the evaluation metrics E for the unbalanced test data (config ST-60-60) with 30min observation windows after running the models (m = naive multi-system model, m_{C_i} = clustered multi-system model, M_s = set of all single-system models). μ = average, σ = standard deviation, p_i = $i\%$ percentile, min = minimum, max = maximum. **Bold** = best results, underlined = second best results.

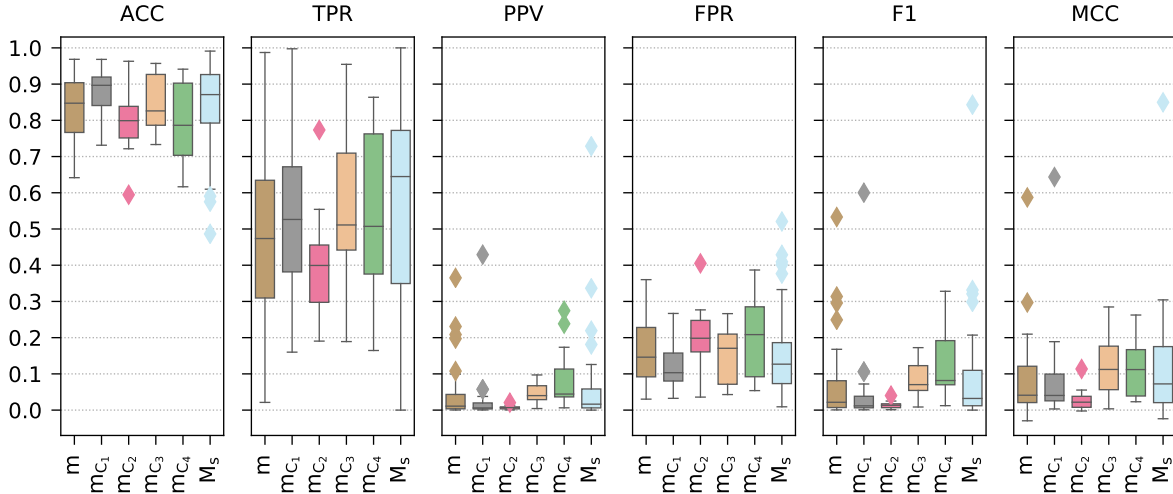
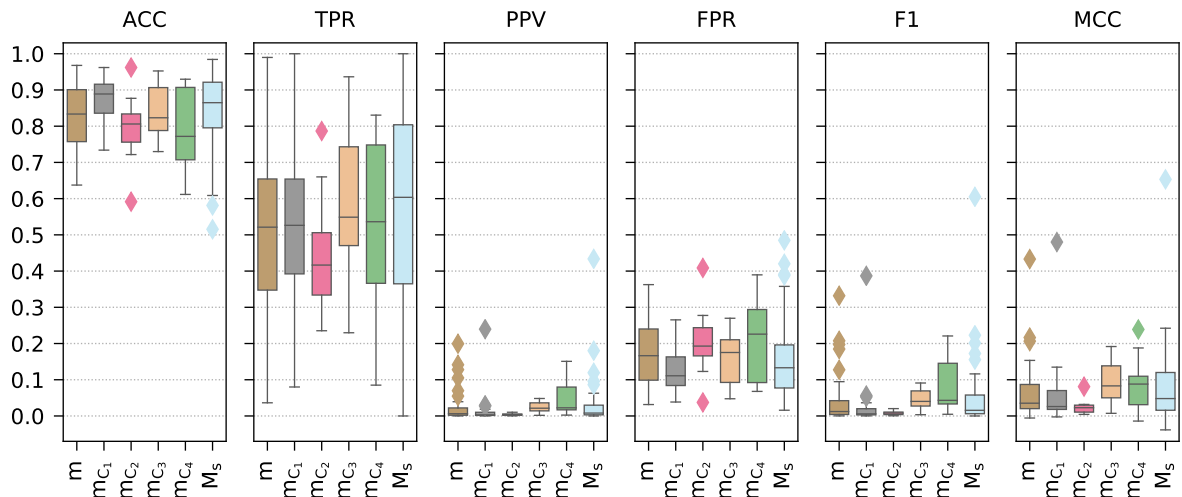


Figure 4.23: Evaluation metrics for the unbalanced test data (config ST-60-60) with 30min observation windows after running the models (m = naive multi-system model, m_{C_i} = clustered multi-system model, M_s = set of all single-system models). Detailed information is available in [Table 4.9](#).

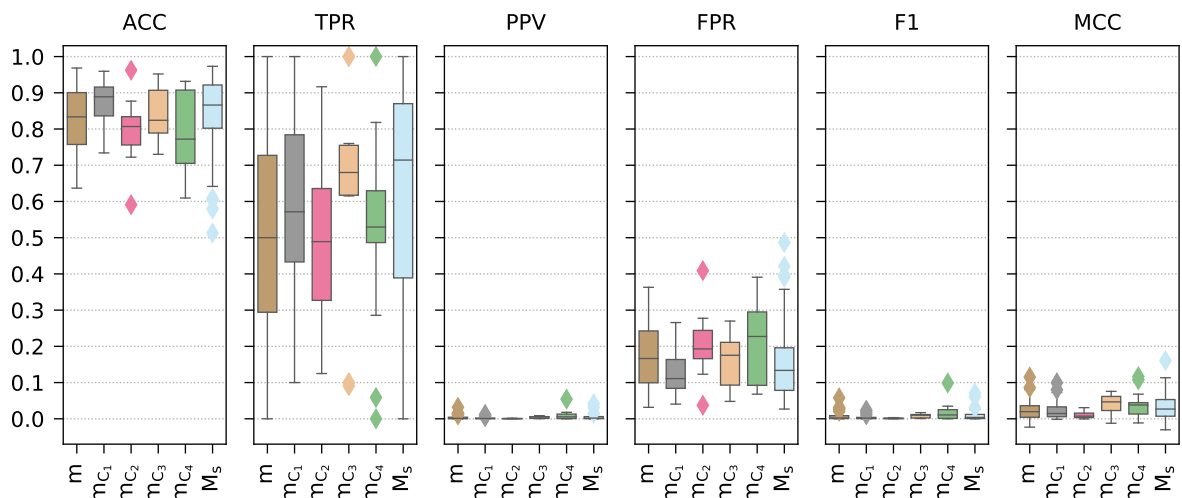
[Section 4.3.2.3](#)). For example, imagine some slide-through sampling timestamp t and two events at timestamps $t + 10$ and $t + 54$. Both events fall into the 60min prediction window ($t + 10 < t + 60$ and $t + 54 < t + 60$), but the lead times (10min and 54min) are significantly different. Our random forest models, on the other hand, were trained with a fixed lead time of zero (using per-event sampling). Naturally, the first idea that comes to mind is to simply extract the testing dataset with a zero lead time as well. However, this causes unavoidable issues. If we were to do that, i.e., setting the prediction window size to zero¹⁶ (resulting in a lead time of zero), then the question would arise which step size to choose. Keeping the 60min step size would mean that we only look at the very first minute and ignore the subsequent 59 minutes. Assuming that events occur randomly according to a uniform distribution, we would thus miss about $\frac{59}{60} \approx 98\%$ of all events. The logical next step would then be to reduce the step size, for example, down to the minimum of 1min. In this case, we would not have the problem of missing events since we slide through every possible timestamp in the observation period and check whether an event occurred at this timestamp. While this seems to be the ideal solution, it unfortunately only shifts the problem to the observation windows because they would now severely overlap. For instance, given the 30min observation windows, we would create a sample at timestamp t_i using the preceding 30 time series data points $\mathbf{x}_{t_i} = (x_{t_i-30}, x_{t_i-29}, \dots, x_{t_i-1})$. Assume that an event happened at this timestamp t_i , so we would label the sample as positive. With the step size of 1min, we would then proceed to timestamp t_{i+1} and again create a sample with the preceding 30 data points $\mathbf{x}_{t_{i+1}} = (x_{t_{i+1}-30}, x_{t_{i+1}-29}, \dots, x_{t_{i+1}-1}) = (x_{t_i-29}, x_{t_i-28}, \dots, x_{t_i})$. We can directly see that 29 values of \mathbf{x}_{t_i} and $\mathbf{x}_{t_{i+1}}$ are identical, resulting in an overlap of $\frac{29}{30} \approx 97\%$. If no event occurred at timestamp t_{i+1} , the sample would be a negative one, which would mean that we created two differently labeled samples that share 97% of the same data. A machine learning model would then nearly always predict both samples as either positive or negative. This leads to the question if appropriate sampling choices actually exist to escape from our predicament, which we discuss in more detail in [Section 4.7](#).

¹⁶Recall that a prediction window of size zero is a special case, where a sample is only classified as positive if an event occurred exactly at this sample's timestamp (cf. [Section 4.3.2.3](#))

With the problems above in mind, we at least wanted to evaluate two alternative testing datasets to see whether different settings result in better or worse predictions. To this end, we changed the slide-through sampling to a step size of 30min together with a prediction window of 30min in the first config ST-30-30 to reduce the varying lead times by 50% compared to the 60min configuration. For the second config ST-30-5, we again employed a step size of 30min but with a prediction window of only 5min, thereby accepting to lose events (the last 25 minutes are ignored in each step) but decreasing the variation in lead times even further. The results of both configurations are presented in [Figure 4.24](#) with details listed in [Table 4.10](#).



(a) Evaluation metrics for config ST-30-30 (cf. [Table 4.10a](#) for details).



(b) Evaluation metrics for config ST-30-5 (cf. [Table 4.10b](#) for details).

Figure 4.24: Evaluation metrics for the unbalanced test data obtained via different slide-through sampling configurations with 30min observation windows after running the models (m = naive multi-system model, m_{C_i} = clustered multi-system model, M_s = set of all single-system models). Detailed information is available in [Table 4.10](#).

The results reveal that the reduced lead time variation in the two new unbalanced testing configurations did not improve the predictions. In fact, both configs led to even worse results: The median and average MCCs of each of the evaluated models of configs ST-30-30 and ST-30-5 are all ≤ 0.09 and ≤ 0.05 , respectively, compared to the previous 0.13 of config ST-60-60. Merged across all models, we get an average MCC of about 0.07 and 0.03, respectively, compared to the previous 0.09.

	E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s		E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s
ACC	μ		0.83	0.87	0.80	<u>0.84</u>	0.79	<u>0.84</u>	ACC	μ		0.83	0.87	0.80	<u>0.84</u>	0.79	<u>0.84</u>
	σ		<u>0.08</u>	0.06	0.10	<u>0.08</u>	0.11	0.11		σ		<u>0.08</u>	0.06	0.10	<u>0.08</u>	0.11	0.11
	min		<u>0.64</u>	0.73	0.59	0.73	0.61	0.52		min		<u>0.64</u>	0.73	0.59	0.73	0.61	0.51
	p_{10}		0.71	0.78	0.71	<u>0.75</u>	0.69	0.69		p_{10}		0.71	0.78	0.71	<u>0.75</u>	0.70	0.68
	p_{25}		0.76	0.84	0.76	0.79	0.71	<u>0.80</u>		p_{25}		0.76	0.84	0.76	0.79	0.71	<u>0.80</u>
	p_{50}		0.83	0.89	0.81	0.82	0.77	<u>0.86</u>		p_{50}		0.83	0.89	0.81	0.82	0.77	<u>0.87</u>
	p_{75}		0.90	0.92	0.83	<u>0.91</u>	<u>0.91</u>	0.92		p_{75}		0.90	0.92	0.83	<u>0.91</u>	<u>0.91</u>	0.92
	max		<u>0.97</u>	0.96	0.96	0.95	0.93	0.98		max		0.97	<u>0.96</u>	<u>0.96</u>	0.95	0.93	0.97
TPR	μ		0.51	0.51	0.45	0.59	0.53	<u>0.58</u>	TPR	μ		0.51	0.57	0.48	0.63	0.52	<u>0.62</u>
	σ		<u>0.22</u>	<u>0.22</u>	0.17	<u>0.22</u>	0.26	0.27		σ		0.29	0.25	0.25	0.30	<u>0.28</u>	0.30
	min		0.04	0.08	0.24	<u>0.23</u>	0.09	0.00		min		0.00	<u>0.10</u>	0.12	0.09	<u>0.00</u>	0.00
	p_{10}		0.23	0.20	<u>0.31</u>	0.32	0.18	0.21		p_{10}		0.14	0.25	<u>0.19</u>	0.10	0.10	0.18
	p_{25}		0.35	<u>0.39</u>	0.33	0.47	0.37	0.36		p_{25}		0.29	0.43	0.33	0.62	<u>0.49</u>	0.39
	p_{50}		0.52	<u>0.53</u>	0.42	<u>0.55</u>	0.54	0.60		p_{50}		0.50	0.57	0.49	<u>0.68</u>	<u>0.53</u>	0.71
	p_{75}		0.65	0.65	0.51	0.74	<u>0.75</u>	0.80		p_{75}		0.73	<u>0.78</u>	0.64	0.75	0.63	0.87
	max		<u>0.82</u>	0.72	0.67	<u>0.82</u>	<u>0.82</u>	0.89		max		<u>0.92</u>	0.86	0.77	1.00	0.82	<u>0.92</u>
PPV	μ		0.02	0.02	0.00	0.02	0.05	<u>0.03</u>	PPV	μ		<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.01	<u>0.00</u>
	σ		0.04	0.05	0.00	<u>0.02</u>	0.05	0.06		σ		<u>0.01</u>	0.00	0.00	0.00	0.01	<u>0.01</u>
	min		0.00	0.00	0.00	0.00	0.00	0.00		min		0.00	0.00	0.00	0.00	0.00	0.00
	p_{10}		0.00	0.00	0.00	0.00	0.00	0.00		p_{10}		0.00	0.00	0.00	0.00	0.00	0.00
	p_{25}		0.00	0.00	0.00	<u>0.01</u>	0.02	0.00		p_{25}		0.00	0.00	0.00	0.00	0.00	0.00
	p_{50}		<u>0.01</u>	0.00	0.00	0.02	0.02	<u>0.01</u>		p_{50}		<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.01	<u>0.00</u>
	p_{75}		0.02	0.01	0.01	<u>0.04</u>	0.08	0.03		p_{75}		<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.01	0.01	0.01
	max		0.05	0.03	0.01	0.04	0.13	0.06		max		<u>0.01</u>	<u>0.01</u>	0.00	<u>0.01</u>	0.02	<u>0.01</u>
FPR	μ		0.17	0.13	0.20	<u>0.16</u>	0.20	<u>0.16</u>	FPR	μ		0.17	0.13	0.20	<u>0.16</u>	0.21	<u>0.16</u>
	σ		<u>0.08</u>	0.06	0.10	<u>0.08</u>	0.11	0.11		σ		<u>0.08</u>	0.06	0.10	<u>0.08</u>	0.11	0.11
	min		<u>0.03</u>	0.04	0.04	0.05	0.07	0.02		min		0.03	<u>0.04</u>	<u>0.04</u>	0.05	0.07	0.03
	p_{10}		0.07	0.05	0.11	0.07	0.09	<u>0.06</u>		p_{10}		0.07	0.05	0.11	0.07	0.09	<u>0.06</u>
	p_{25}		0.10	0.08	0.17	<u>0.09</u>	<u>0.09</u>	0.08		p_{25}		0.10	0.08	0.17	<u>0.09</u>	<u>0.09</u>	0.08
	p_{50}		0.17	0.11	0.19	0.18	0.23	<u>0.13</u>		p_{50}		0.17	0.11	0.19	0.18	0.23	<u>0.13</u>
	p_{75}		0.24	0.16	0.24	0.21	0.29	<u>0.20</u>		p_{75}		0.24	0.16	0.24	0.21	0.30	<u>0.20</u>
	max		<u>0.29</u>	0.22	0.29	<u>0.25</u>	0.30	0.31		max		<u>0.29</u>	0.22	0.29	<u>0.25</u>	0.30	0.32
F1	μ		0.04	0.03	0.01	<u>0.05</u>	0.08	<u>0.05</u>	F1	μ		<u>0.01</u>	0.00	0.00	<u>0.01</u>	0.02	<u>0.01</u>
	σ		0.06	0.08	0.01	<u>0.03</u>	0.08	0.09		σ		<u>0.01</u>	<u>0.01</u>	0.00	<u>0.01</u>	0.03	<u>0.01</u>
	min		0.00	0.00	0.00	0.00	0.00	0.00		min		0.00	0.00	0.00	0.00	0.00	0.00
	p_{10}		<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.01	0.01	<u>0.00</u>		p_{10}		0.00	0.00	0.00	0.00	0.00	0.00
	p_{25}		0.00	0.00	0.00	0.03	0.03	<u>0.01</u>		p_{25}		0.00	0.00	0.00	0.00	0.00	0.00
	p_{50}		0.01	0.01	0.01	0.04	0.04	<u>0.02</u>		p_{50}		<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.01	0.01	<u>0.00</u>
	p_{75}		0.04	0.02	0.01	<u>0.07</u>	0.15	0.06		p_{75}		<u>0.01</u>	0.00	0.00	<u>0.01</u>	0.03	<u>0.01</u>
	max		0.08	0.05	0.01	0.08	0.21	<u>0.11</u>		max		<u>0.02</u>	0.01	0.00	<u>0.02</u>	0.03	<u>0.02</u>
MCC	μ		0.06	0.06	0.03	0.09	0.09	<u>0.08</u>	MCC	μ		0.02	0.02	0.01	0.04	0.04	<u>0.03</u>
	σ		0.07	0.10	0.02	<u>0.06</u>	0.07	0.10		σ		<u>0.03</u>	<u>0.03</u>	0.01	<u>0.03</u>	0.04	0.04
	min		-0.01	<u>0.00</u>	<u>0.00</u>	0.01	-0.01	-0.04		min		-0.02	0.00	0.00	<u>-0.01</u>	<u>-0.01</u>	-0.03
	p_{10}		0.00	<u>0.01</u>	<u>0.01</u>	<u>0.01</u>	0.02	0.00		p_{10}		0.00	0.00	0.00	<u>-0.01</u>	0.00	0.00
	p_{25}		0.02	0.02	0.01	0.05	<u>0.03</u>	0.02		p_{25}		0.00	<u>0.01</u>	0.00	0.02	<u>0.01</u>	<u>0.01</u>
	p_{50}		0.04	0.03	0.02	<u>0.08</u>	0.09	0.05		p_{50}		0.02	0.01	0.01	0.05	<u>0.04</u>	0.03
	p_{75}		0.09	0.07	0.03	0.14	0.11	<u>0.12</u>		p_{75}		0.04	0.03	0.02	0.06	0.04	<u>0.05</u>
	max		0.13	0.12	0.04	0.19	<u>0.18</u>	<u>0.18</u>		max		0.06	0.07	0.03	0.07	0.10	<u>0.08</u>
			<u>0.43</u>	0.08	0.19	0.24	0.65				<u>0.12</u>	0.10	0.03	0.08	<u>0.12</u>	0.16	

(a) Config ST-30-30 statistics.

(b) Config ST-30-5 statistics.

Table 4.10: Statistics S of the evaluation metrics E for the unbalanced test data obtained via different slide-through sampling configurations with 30min observation windows after running the models (m = naive multi-system model, m_{C_i} = clustered multi-system model, M_s = set of all single-system models). μ = average, σ = standard deviation, $p_i = i\%$ percentile, min = minimum, max = maximum. **Bold** = best results, underlined = second best results.

These underwhelming results made us think about additional reasons why our models did not perform well, and we concluded that also the training phase can indirectly suffer from the data imbalance. Indirectly means that we do train the models appropriately with balanced data, but this data itself might not be adequately representative of the true data distribution, which is heavily influenced by the class imbalance. Our training configuration specifies that we randomly extract a negative sample for every positive sample (per-event sampling with a balanced ratio). If the events are rare, we obtain a very limited number of positive samples, which, in turn, leads to equally few negative samples as well. This can be problematic, as these few negative samples might only represent a small part of the actual, true negative data distribution. [Figure 4.25](#) visualizes this problem with a two-dimensional dataset. In the example, suppose we have a negative to positive class imbalance ratio of 99% to 1% and all available negative data is shown in the left plot (*Negative*). Due to this severe imbalance, we only sample $\frac{1}{99} \approx 1\%$ of all available negative data (middle plot), which does not accurately represent the original moon-shaped data distribution. However, if we sample more negative data such as, for instance, 40% (right plot), then the original data distribution can indeed be approximated, meaning that we cover many more important characteristics compared to the 1%. As we still require balanced data to train our machine learning models, we cannot simply adapt the balancing ratio. We thus need a way to increase the number of positive samples, which, in turn, results in sampling more negative data. This can be achieved with oversampling, where we utilize data augmentation.

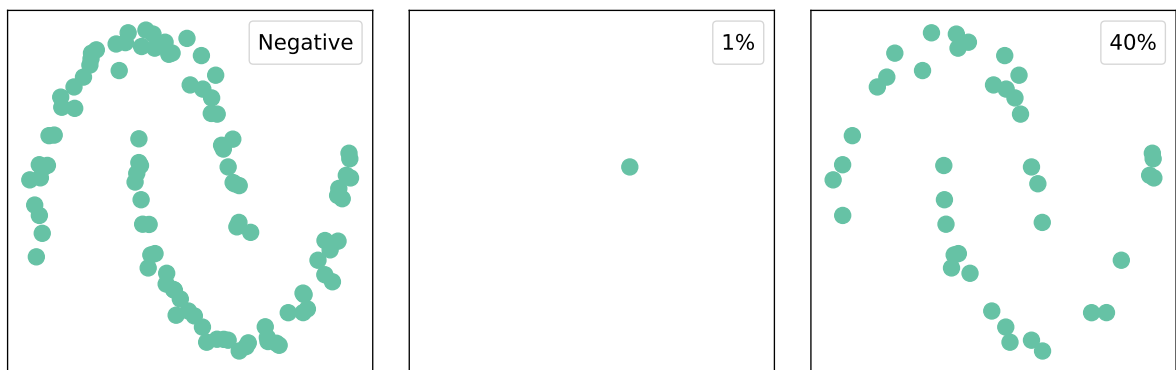


Figure 4.25: Sampling the moon-shaped negative data (left) with 1% (middle) compared to 40% (right), resulting in an incorrect/useless distribution approximation in the first case and an acceptable/usable approximation in the second case.

It might at first seem counterintuitive that the class imbalance ratio is affected by the sampling configuration, since the number of events throughout the observation period does not change. However, depending on the slide-through config settings, we might potentially not include every event (e.g., the prediction window is smaller than the step size, so some events are missed), or we get increasingly more negative samples the smaller the step size becomes (e.g., step size and prediction window size x compared to $\frac{x}{2}$ yields twice as many negative samples in the latter case, while the number of positive samples remains the same).

In our test datasets, config ST-60-60 leads to an extreme imbalance with 99.76% negative samples and only 0.24% positive samples. Configs ST-30-30 and ST-30-5 then even worsen the situation with 99.88% to 0.12% (twice the amount of negative samples) and 99.98% to 0.02% (missing approximately a fifth of all events), respectively.¹⁷ Data augmentation can help us in addressing this imbalance by creating additional positive samples in the training data.

¹⁷However, the ST-30-5 config can be ignored here because the low class imbalance comes from missing events (prediction window is smaller than the step size), which is not the case in our training configurations that employ per-event sampling where all events are recorded.

However, care must be taken to not arbitrarily alter the available positive samples, as it could happen that a sample is changed too much, thereby losing its original data characteristics or, in the worst case, even becoming similar to negative samples. A moderate augmentation is thus preferable, so we decided to apply time warping (cf. [Section 4.3.2.4](#)) with three knots and a small standard deviation of 0.05 in order not to distort our positive samples too much. In addition to the original sample, we set the number of augmentations to 250, meaning that we create 250 times more positive samples and, in turn, 250 times more negative samples in our per-event sampled training data. We chose this number based on the above test dataset imbalance statistics of the ST-60-60 config, where 250 additional positive samples per event then result in a new negative to positive ratio of 61.9% to 38.1%, and we thus sample $\frac{38.1}{61.9} \approx 61.6\%$ of the negative data, for which a much better data distribution approximation can be expected (cf. [Figure 4.25](#)). We exported new training data using the same training configurations as described above but with the addition of augmenting the positive samples. We refer to these configurations as ST-X-Y-augmented (slide-through sampling, Xmin step size, Ymin prediction window, augmented training data). Afterwards, we trained our models again with this new data and evaluated them on the test datasets. The results are presented in [Figure 4.26](#) with details listed in [Table 4.11](#).

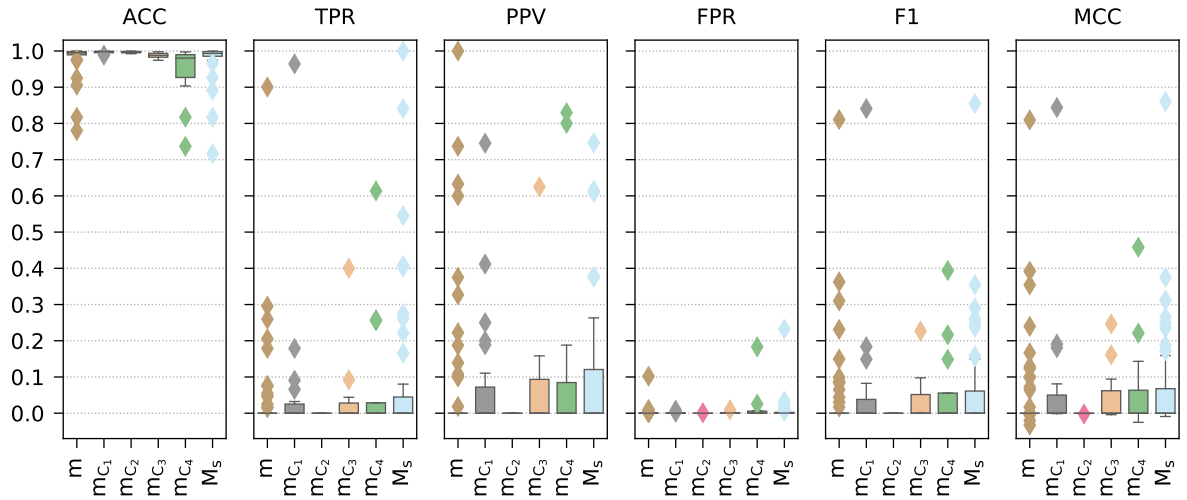
Unfortunately, augmenting the training data led to even worse results. While the accuracies (ACC) and the false positive rates (FPR) are close to perfect (100% and 0%, respectively), this is merely due to the fact that our models simply predicted the negative class in nearly all cases, all other metrics are the exact opposite. If all samples are predicted as negative, the evaluation metrics PPV and MCC cannot even be computed (division by zero), in which case we use zero as a fallback value. The median and average MCCs of each of the evaluated models of configs ST-60-60-augmented, ST-30-30-augmented and ST-30-5-augmented are all ≤ 0.07 (0.13 without augmentation), ≤ 0.05 (0.09 without augmentation) and ≤ 0.02 (0.05 without augmentation), respectively. Merged across all models, we get an average MCC of about 0.05 (0.09 without augmentation), 0.04 (0.07 without augmentation) and 0.02 (0.03 without augmentation).

Two questions arise from all the previous observations. First, can events be predicted with the slide-through sampling approach at all? Second, why are the results in the balanced scenario so much better? To answer both questions, we will introduce synthetic data and then discuss the imbalance problem once more upon revisiting the balanced data.

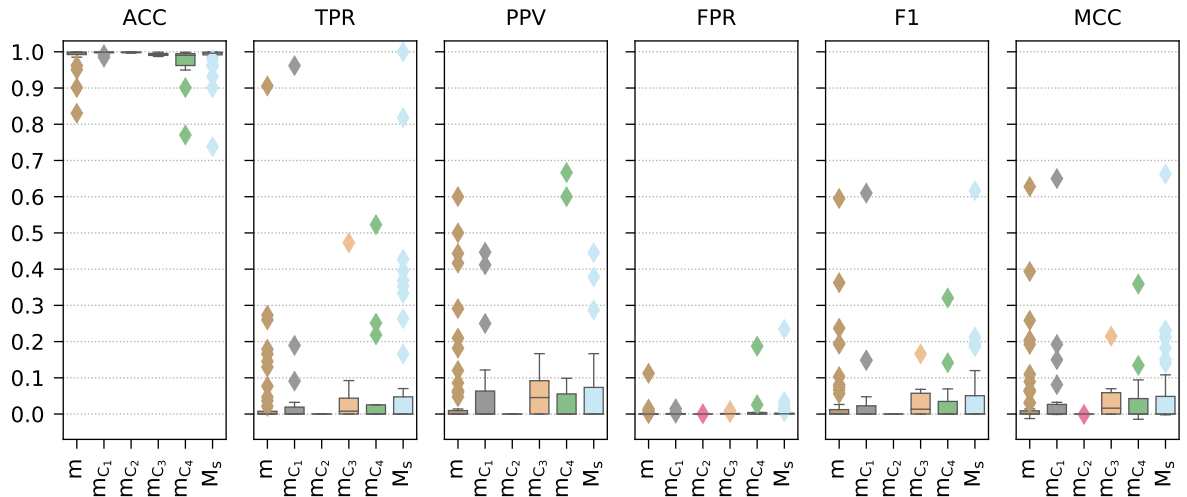
4.6.4 Synthetic Data

To analyze the impact of our slide-through sampling technique, the idea is to create synthetic data where we know that events can be predicted from the time series data because we change the corresponding series in the time window in front of the event. Since the unbalanced scenario already caused problems when evaluating our single-system models, we thus created a single synthetic system for this investigation, covering an export period of five days with the following characteristics and setup:

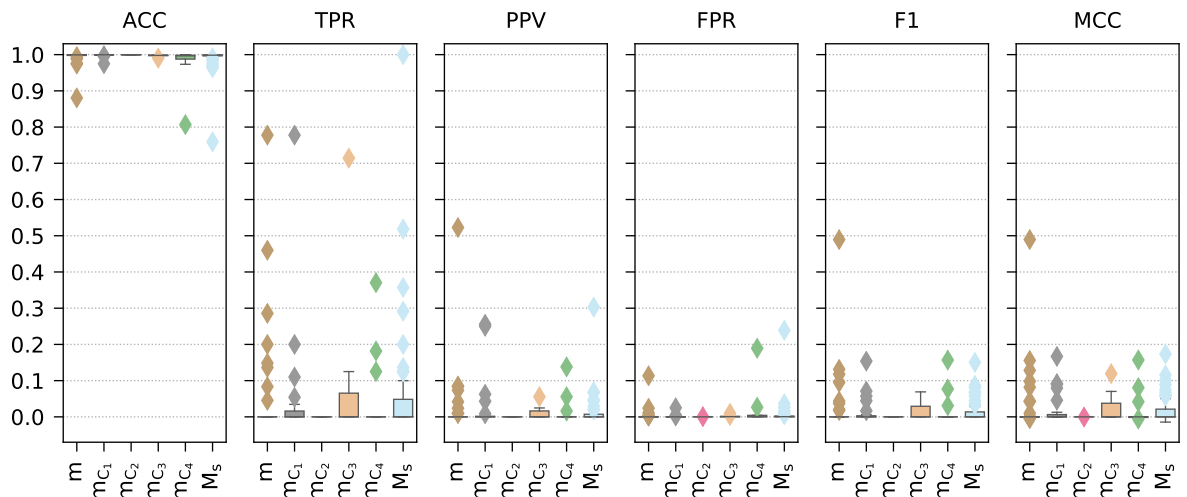
- **Components:** The system consists of 50 services, each running on a single host (i.e., 50 hosts in total). One to three disks and networks can be connected to a host, which are chosen randomly based on a uniform distribution. The services are the entities where synthetic events occur.



(a) Evaluation metrics for config ST-60-60-augmented (cf. [Table 4.11a](#) for details).



(b) Evaluation metrics for config ST-30-30-augmented (cf. [Table 4.11b](#) for details).



(c) Evaluation metrics for config ST-30-5-augmented (cf. [Table 4.11c](#) for details).

Figure 4.26: Evaluation metrics for the unbalanced test data obtained via different slide-through sampling configurations with 30min observation-system windows after running the models (m = naive multi-system model, m_{C_i} = clustered multi-system model, M_s = set of all single-system models) that were trained with augmented data (cf. [Table 4.11](#) for details).

	E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s		E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s	
ACC	μ		0.98	1.00	1.00	<u>0.99</u>	0.94	0.98	ACC	μ		<u>0.99</u>	1.00	1.00	<u>0.99</u>	0.96	<u>0.99</u>	
	σ		0.04	0.00	0.00	<u>0.01</u>	0.08	0.05		σ			<u>0.03</u>	0.00	0.00	0.00	0.06	0.04
	min		0.78	0.99	0.99	<u>0.97</u>	0.74	0.72		min			0.83	<u>0.99</u>	1.00	<u>0.99</u>	0.77	0.74
	p_{10}		<u>0.98</u>	0.99	0.99	<u>0.98</u>	0.83	0.97		p_{10}			<u>0.99</u>	<u>0.99</u>	1.00	<u>0.99</u>	0.91	0.98
	p_{25}		<u>0.99</u>	1.00	1.00	0.98	0.93	<u>0.99</u>		p_{25}			<u>0.99</u>	1.00	1.00	<u>0.99</u>	0.96	<u>0.99</u>
	p_{50}		<u>0.99</u>	1.00	1.00	<u>0.99</u>	0.98	<u>0.99</u>		p_{50}			1.00	1.00	1.00	<u>0.99</u>	<u>0.99</u>	1.00
	p_{75}		1.00	1.00	1.00	<u>0.99</u>	<u>0.99</u>	1.00		p_{75}			1.00	1.00	1.00	1.00	<u>0.99</u>	1.00
	p_{90}		1.00	1.00	1.00	<u>0.99</u>	<u>0.99</u>	1.00		p_{90}			1.00	1.00	1.00	1.00	1.00	1.00
max		1.00	1.00	1.00	1.00	1.00	1.00	max			1.00	1.00	1.00	1.00	1.00	1.00		
TPR	μ		0.04	0.06	0.00	0.05	0.09	<u>0.08</u>	TPR	μ		0.04	<u>0.06</u>	0.00	<u>0.06</u>	0.08	0.08	
	σ		0.13	0.20	0.00	<u>0.12</u>	0.18	0.20		σ			<u>0.13</u>	0.20	0.00	0.14	0.16	0.19
	min		0.00	0.00	0.00	0.00	0.00	0.00		min			0.00	0.00	0.00	0.00	0.00	0.00
	p_{10}		0.00	0.00	0.00	0.00	0.00	0.00		p_{10}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{25}		0.00	0.00	0.00	0.00	0.00	0.00		p_{25}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{50}		0.00	0.00	0.00	0.00	0.00	0.00		p_{50}			<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.01	<u>0.00</u>	<u>0.00</u>
	p_{75}		0.00	<u>0.03</u>	0.00	<u>0.03</u>	<u>0.03</u>	0.04		p_{75}			0.01	0.02	0.00	<u>0.04</u>	0.03	0.05
	p_{90}		0.08	0.09	0.00	0.09	<u>0.26</u>	0.27		p_{90}			0.14	0.08	0.00	0.09	<u>0.24</u>	0.34
max		0.90	<u>0.96</u>	0.00	0.40	0.61	1.00	max			0.91	<u>0.96</u>	0.00	0.47	<u>0.52</u>	1.00		
PPV	μ		0.08	0.08	0.00	<u>0.10</u>	0.15	0.09	PPV	μ		0.05	<u>0.06</u>	0.00	0.05	0.11	0.05	
	σ		0.20	0.18	0.00	0.18	0.30	<u>0.17</u>		σ			0.13	0.13	0.00	<u>0.06</u>	0.23	0.09
	min		0.00	0.00	0.00	0.00	0.00	0.00		min			0.00	0.00	0.00	0.00	0.00	0.00
	p_{10}		0.00	0.00	0.00	0.00	0.00	0.00		p_{10}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{25}		0.00	0.00	0.00	0.00	0.00	0.00		p_{25}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{50}		0.00	0.00	0.00	0.00	0.00	0.00		p_{50}			<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.05	<u>0.00</u>	<u>0.00</u>
	p_{75}		0.00	0.07	0.00	<u>0.09</u>	0.08	0.12		p_{75}			<u>0.01</u>	<u>0.06</u>	<u>0.00</u>	0.09	0.06	<u>0.07</u>
	p_{90}		<u>0.26</u>	0.24	0.00	0.16	0.68	<u>0.26</u>		p_{90}			0.19	<u>0.22</u>	0.00	0.10	0.50	0.15
max		1.00	0.75	0.00	0.62	<u>0.83</u>	0.75	max			<u>0.60</u>	0.45	0.00	0.17	0.67	0.45		
FPR	μ		0.00	0.00	0.00	0.00	0.02	<u>0.01</u>	FPR	μ		0.00	0.00	0.00	0.00	0.02	<u>0.01</u>	
	σ		<u>0.01</u>	0.00	0.00	0.00	0.05	0.03		σ			<u>0.01</u>	0.00	0.00	0.00	0.05	0.03
	min		0.00	0.00	0.00	0.00	0.00	0.00		min			0.00	0.00	0.00	0.00	0.00	0.00
	p_{10}		0.00	0.00	0.00	0.00	0.00	0.00		p_{10}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{25}		0.00	0.00	0.00	0.00	0.00	0.00		p_{25}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{50}		0.00	0.00	0.00	0.00	0.00	0.00		p_{50}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{75}		0.00	0.00	0.00	0.00	<u>0.01</u>	0.00		p_{75}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{90}		0.00	0.00	0.00	0.00	0.02	<u>0.01</u>		p_{90}			0.00	0.00	0.00	0.00	0.02	<u>0.01</u>
max		0.10	<u>0.01</u>	0.00	<u>0.01</u>	0.18	0.23	max			0.11	<u>0.01</u>	0.00	<u>0.01</u>	0.19	0.23		
F1	μ		<u>0.04</u>	0.06	0.00	<u>0.04</u>	0.06	0.06	F1	μ		<u>0.04</u>	0.05	0.00	0.03	0.05	0.05	
	σ		0.13	0.18	0.00	<u>0.07</u>	0.12	0.14		σ			0.10	0.13	0.00	<u>0.05</u>	0.09	0.10
	min		0.00	0.00	0.00	0.00	0.00	0.00		min			0.00	0.00	0.00	0.00	0.00	0.00
	p_{10}		0.00	0.00	0.00	0.00	0.00	0.00		p_{10}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{25}		0.00	0.00	0.00	0.00	0.00	0.00		p_{25}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{50}		0.00	0.00	0.00	0.00	0.00	0.00		p_{50}			<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.01	<u>0.00</u>	<u>0.00</u>
	p_{75}		0.00	0.04	0.00	<u>0.05</u>	0.06	0.06		p_{75}			0.01	0.02	0.00	0.06	0.03	<u>0.05</u>
	p_{90}		0.09	0.14	0.00	0.10	0.20	<u>0.19</u>		p_{90}			0.09	<u>0.13</u>	0.00	0.07	<u>0.13</u>	0.15
max		0.81	<u>0.84</u>	0.00	0.23	0.39	0.85	max			0.60	<u>0.61</u>	0.00	0.17	0.32	0.62		
MCC	μ		0.04	<u>0.06</u>	0.00	0.05	<u>0.06</u>	0.07	MCC	μ		<u>0.04</u>	0.05	0.00	<u>0.04</u>	0.05	0.05	
	σ		0.13	0.18	0.00	<u>0.08</u>	0.14	0.14		σ			0.11	0.14	0.00	<u>0.06</u>	0.10	0.11
	min		-0.03	0.00	0.00	0.00	-0.03	<u>-0.01</u>		min			<u>-0.01</u>	0.00	0.00	0.00	<u>-0.01</u>	0.00
	p_{10}		0.00	0.00	0.00	0.00	<u>-0.01</u>	0.00		p_{10}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{25}		0.00	0.00	0.00	0.00	0.00	0.00		p_{25}			0.00	0.00	0.00	0.00	0.00	0.00
	p_{50}		0.00	0.00	0.00	0.00	0.00	0.00		p_{50}			<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.02	<u>0.00</u>	<u>0.00</u>
	p_{75}		0.00	0.05	0.00	<u>0.06</u>	<u>0.06</u>	0.07		p_{75}			0.01	0.03	0.00	0.06	0.04	<u>0.05</u>
	p_{90}		0.13	<u>0.16</u>	0.00	<u>0.16</u>	0.21	0.21		p_{90}			0.10	<u>0.14</u>	0.00	0.07	0.13	0.16
max		0.81	<u>0.84</u>	0.00	0.25	0.46	0.86	max			0.63	<u>0.65</u>	0.00	0.21	0.36	0.66		

(a) Config ST-60-60-augmented statistics.

(b) Config ST-30-30-augmented statistics.

E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s
ACC	μ	1.00	1.00	1.00	1.00	0.98	<u>0.99</u>
	σ	0.02	<u>0.01</u>	0.00	0.00	0.05	<u>0.03</u>
	min	0.88	<u>0.97</u>	1.00	<u>0.99</u>	0.81	0.76
	p_{10}	1.00	1.00	1.00	1.00	0.97	<u>0.99</u>
	p_{25}	1.00	1.00	1.00	1.00	<u>0.99</u>	1.00
	p_{50}	1.00	1.00	1.00	1.00	1.00	1.00
	p_{75}	1.00	1.00	1.00	1.00	1.00	1.00
	p_{90}	1.00	1.00	1.00	1.00	1.00	1.00
	max	1.00	1.00	1.00	1.00	1.00	1.00
	TPR	μ	0.04	0.05	0.00	0.09	0.05
σ		0.13	0.16	0.00	0.21	<u>0.11</u>	0.16
min		0.00	0.00	0.00	0.00	0.00	0.00
p_{10}		0.00	0.00	0.00	0.00	0.00	0.00
p_{25}		0.00	0.00	0.00	0.00	0.00	0.00
p_{50}		0.00	0.00	0.00	0.00	0.00	0.00
p_{75}		0.00	0.02	0.00	0.07	0.00	<u>0.05</u>
p_{90}		0.10	0.10	0.00	0.12	0.17	<u>0.13</u>
max		<u>0.78</u>	<u>0.78</u>	0.00	0.71	0.37	1.00
PPV		μ	0.01	0.03	0.00	0.01	<u>0.02</u>
	σ	0.07	0.07	0.00	<u>0.02</u>	0.04	0.04
	min	0.00	0.00	0.00	<u>0.00</u>	0.00	0.00
	p_{10}	0.00	0.00	0.00	0.00	0.00	0.00
	p_{25}	0.00	0.00	0.00	0.00	0.00	0.00
	p_{50}	0.00	0.00	0.00	0.00	0.00	0.00
	p_{75}	0.00	0.00	0.00	0.02	0.00	<u>0.01</u>
	p_{90}	0.02	0.06	0.00	0.02	<u>0.05</u>	0.04
	max	0.52	0.26	0.00	0.06	0.14	<u>0.30</u>
	FPR	μ	0.00	0.00	0.00	0.00	0.02
σ		0.02	<u>0.01</u>	0.00	0.00	0.05	0.03
min		0.00	0.00	0.00	0.00	0.00	0.00
p_{10}		0.00	0.00	0.00	0.00	0.00	0.00
p_{25}		0.00	0.00	0.00	0.00	0.00	0.00
p_{50}		0.00	0.00	0.00	0.00	0.00	0.00
p_{75}		0.00	0.00	0.00	0.00	0.00	0.00
p_{90}		0.00	0.00	0.00	0.00	0.02	<u>0.01</u>
max		0.11	0.03	0.00	<u>0.01</u>	0.19	0.24
F1		μ	0.02	0.02	0.00	0.02	0.02
	σ	0.07	0.04	0.00	<u>0.02</u>	0.05	0.03
	min	0.00	0.00	0.00	<u>0.00</u>	0.00	0.00
	p_{10}	0.00	0.00	0.00	0.00	0.00	0.00
	p_{25}	0.00	0.00	0.00	0.00	0.00	0.00
	p_{50}	0.00	0.00	0.00	0.00	0.00	0.00
	p_{75}	0.00	0.00	0.00	0.03	0.00	<u>0.01</u>
	p_{90}	0.03	<u>0.05</u>	0.00	0.04	0.07	<u>0.05</u>
	max	0.49	0.15	0.00	0.07	<u>0.16</u>	0.15
	MCC	μ	0.02	0.02	<u>0.00</u>	0.02	0.02
σ		0.07	<u>0.04</u>	0.00	<u>0.04</u>	0.05	<u>0.04</u>
min		<u>-0.01</u>	0.00	0.00	0.00	<u>-0.01</u>	<u>-0.01</u>
p_{10}		0.00	0.00	0.00	0.00	0.00	0.00
p_{25}		0.00	0.00	0.00	0.00	0.00	0.00
p_{50}		0.00	0.00	0.00	0.00	0.00	0.00
p_{75}		0.00	0.01	0.00	0.04	0.00	<u>0.02</u>
p_{90}		0.04	0.09	0.00	<u>0.07</u>	<u>0.07</u>	0.06
max		0.49	<u>0.17</u>	0.00	0.12	0.16	<u>0.17</u>

(c) Config ST-30-5-augmented statistics.

Table 4.11: Statistics S of the evaluation metrics E for the unbalanced test data obtained via different slide-through sampling configurations with 30min observation windows after running the models (m = naive multi-system model, m_{C_i} = clustered multi-system model, M_s = set of all single-system models) that were trained with augmented data. μ = average, σ = standard deviation, p_i = $i\%$ percentile, min = minimum, max = maximum. **Bold** = best results, underlined = second best results.

- Time series metrics: Hosts are assigned the metric H-01 (CPU idle), disks the metric D-03 (disk available) and networks the metric N-09 (receiving utilization). Analogously to the real data, all these metrics are percentages and thus their values are within the interval $[0, 100]$.
- Time series data: The creation of the actual time series data is done via *signals*, where a signal determines the shape of the time series that is associated with a specific entity and metric. Given a set of candidate signals, we then randomly (uniform distribution) select one for each such entity and metric. [Table 4.12](#) presents an overview of the signals, including a short description and representative example time series in [Figure 4.27](#). Each time series has a resolution of one minute.

Metric	Signal	Description	Example
H-01	noise_const	Constant signal with noise.	Figure 4.27a
	noise_sinus	Periodic sinus-shaped signal with noise.	Figure 4.27b
D-03	noise_trend	Negative linear trend signal with noise.	Figure 4.27c
	curves	Random curve-shaped signal.	Figure 4.27d
	saw	Periodic saw-shaped signal.	Figure 4.27e
N-09	pulse	Signal with random pulses (sharp spikes).	Figure 4.27f
	noise_rect	Periodic rectangle-shaped signal with noise.	Figure 4.27g

Table 4.12: The set of candidate signals for each metric, including a reference to an example time series (cf. [Figure 4.27](#)) that was created with the corresponding signal.

- Events: This is the most important part since the events and their *effects* on the time series form the basis of the event prediction. Every time an event occurs on a service, the time series of the connected host, disk and network entities are affected as specified in [Table 4.13](#). We chose the effects in a way that multiple lengths and offsets are present in the data, so we can investigate the impact on the observation windows as well as the different slide-through step sizes and prediction windows. Furthermore, the magnitudes allow a clear distinction between normal data and event-affected data. Events are configured to happen on 50% of all services ($50 \cdot 50\% = 25$), and per service, 50 events occur at random timestamps (uniform distribution). For our five-day export, this means that we recorded $25 \cdot 50 = 1250$ events in total.

With the synthetic system ready to be studied, the next step is to create all necessary configurations. Again, separate training and testing configurations are required due to the different sampling techniques, so we applied our day-based cross-validation once more, i.e., always four days are used for training and the remaining day for testing. First, we decided to run a per-event-based testing config to see how well our approach performs theoretically. We used the same configuration settings as in the unbalanced scenario (cf. [Section 4.6.2](#)), minus all multi-system-related settings, as we only need to evaluate our single synthetic system. Here, we expect near perfect results since the synthetic events and their effects on the time series are clearly visible, and thus, our approach should easily be able to distinguish between negative and positive per-event samples. [Table 4.14](#) lists the evaluation metrics after running the random forest classifiers (100 trees, no depth limit).

Clearly, the results are indeed perfect or very close to perfect. In fact, none of the observation window runs predicted even a single false positive, and the maximum number of false negatives was four for the 60min observation window configuration. Moreover, similarly

Metric	Effect	Description	Example
H-01	peak	A rectangular peak of either length 10 and magnitude 100 with offset 0, or length 15 and magnitude 20 with offset 10 (chosen randomly based on a uniform distribution).	Figure 4.27h
D-03	warp	A signal change of length 30 based on magnitude warping (cf. Section 4.3.2.4) with 10 knots and a standard deviation (magnitude) of 3 with offset 0 or offset 15 (chosen randomly based on a uniform distribution).	Figure 4.27i
N-09	peak	A rectangular peak of length 5 and magnitude 100 with offset 0.	Figure 4.27j

Table 4.13: Signal change effects for each metric in case of an event occurrence, including a reference to an example of events affecting a concrete time series. The *offset* describes the time span between an event occurrence and the end of the “effect impact window”, which has a size of *length*. For example, offset 10 and length 15 mean that we go back 10 minutes from the event timestamp, and then the preceding 15 minutes (effect impact window) of the affected time series data are altered. The *magnitude* indicates how much the time series is changed.

E	5min	10min	15min	30min	45min	60min
ACC	1.0000	1.0000	0.9996	0.9996	0.9992	0.9984
TPR	1.0000	1.0000	0.9992	0.9992	0.9984	0.9968
PPV	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
FPR	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
F1	1.0000	1.0000	0.9996	0.9996	0.9992	0.9984
MCC	1.0000	1.0000	0.9992	0.9992	0.9984	0.9968

Table 4.14: Evaluation metrics E of the balanced per-event testing data, grouped by the observation windows.

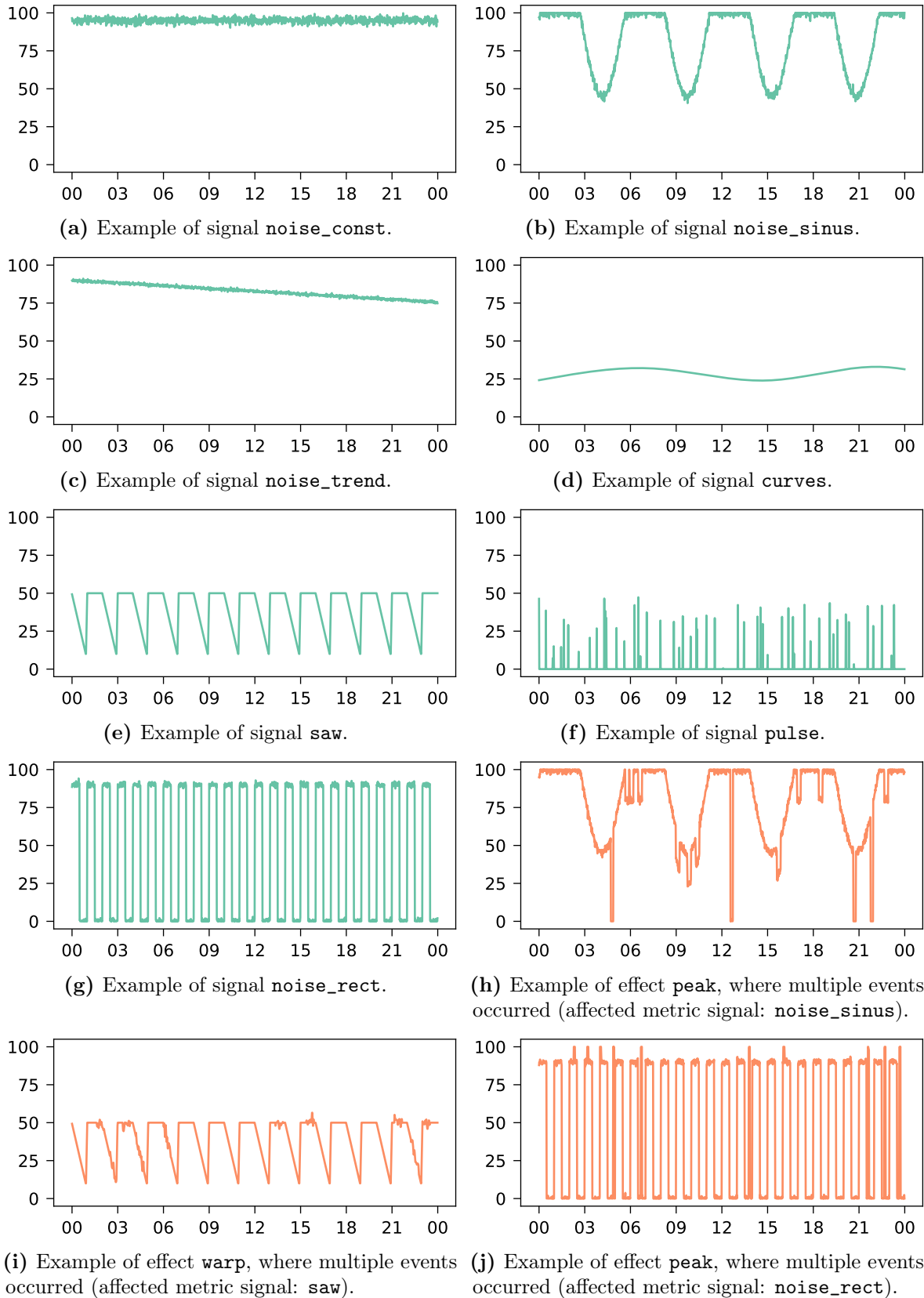


Figure 4.27: Examples of time series that were created with the corresponding signals as listed in [Table 4.12](#) ([4.27a](#) to [4.27g](#)), and examples showing how events influence the time series with their effects ([4.27h](#) to [4.27j](#)). The x-axis indicates the hours of the day in 24h format, e.g., 15 means 15:00 (3pm).

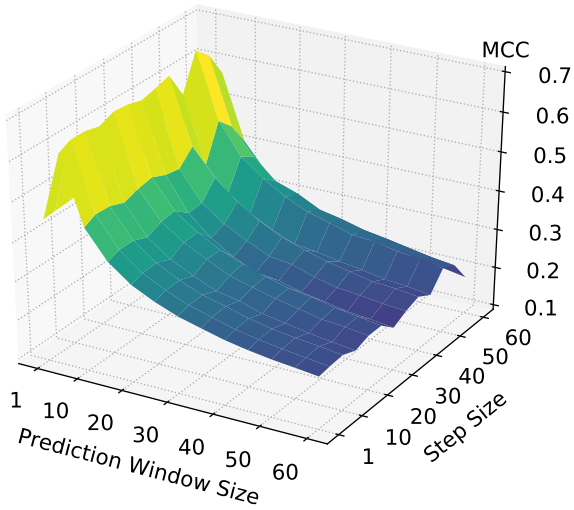
to the real-world results, we can see that the different observation windows apparently did not influence the prediction quality, which indicates that all window sizes were able to capture our synthetic event effects. Naturally, other synthetic data could yield a different outcome, but this would require a much more in-depth evaluation. However, this is not the focus of this section, as we actually want to investigate how the slide-through sampling performs in a known and controlled environment.

The main analysis is then achieved with various slide-through sampling configurations. To get a good overview, we decided to evaluate the full cross product of step sizes and prediction windows over the following 13 options: $\{1, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60\}$. This means that we create $13 \cdot 13 = 169$ slide-through configurations, ranging from ST-1-1¹⁸ to ST-60-60 with all possible combinations. Together with our six observation window sizes, we thus have to run $6 \cdot 169 = 1014$ individual configurations, all with a five-day cross-validation. Figure 4.28 shows the Matthews correlation coefficient (MCC) values that were achieved when running all these testing configurations, grouped by the observation windows. The results for the other evaluation metrics can be found in the appendix (cf. Section C.2 on p. 223).

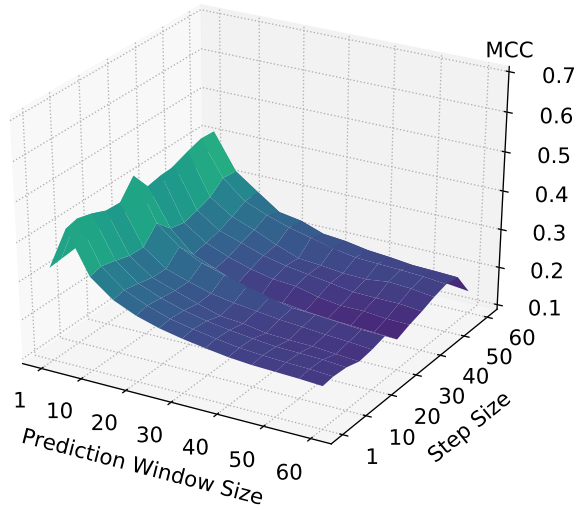
Before going into detail, it must be noted that we also tested configurations that potentially do not capture all events or count the same event multiple times. The former is the case when the prediction window size is smaller than the step size (problem already introduced in Section 4.6.3). The latter occurs if it is the other way around, i.e., the step size is smaller than the prediction window size, which leads to parts of the prediction window data being visited multiple times. However, as we can see in the MCC plots, the step size parameter evidently does not really impact the results. The prediction window size, on the other hand, is much more important, as are the observation windows. In our synthetic scenario, using 5min prediction windows yielded the best evaluation metric scores, and the 5min observation windows (followed by 15min) outperformed the other options. Unfortunately, it is now apparent that the slide-through sampling approach is not ideal for the task of event prediction. Given that we started from a near perfect prediction in the per-event case, we now dropped down to an MCC maximum of about 0.65, a loss of a formidable amount of 0.35. While an MCC of 0.65 is not too bad in its own right, we have to remember that the raw testing data is not different than the training data, i.e., the time series and events, including their effects, are the same. This means that we have a limited predictive capability despite the fact that the synthetic data should be (almost) perfectly predictable, as demonstrated in the per-event testing configurations.

We also discussed the potential problem of having an underrepresented negative class distribution in the training data, where oversampling techniques such as data augmentation can help. Since we have considerably more events that occur in our synthetic scenario (1250 events in five days in a single system), this issue is much less relevant, as chances are high that selecting a negative sample for every positive sample already results in a sufficient approximation of the negative class distribution. Nevertheless, we decided to run the 1014 testing configurations two more times, where the random forest models were trained with augmented data, in the first run with five additional augmentations and in the second run with ten additional augmentations. As augmentation function, we used time warping once more with the same parameters as in the real-world scenario (three knots, standard deviation of 0.05). The MCC results for the 5min observation windows in comparison to the non-augmented data are presented in Figure 4.29. The results for the remaining observation windows as well as all other evaluation metrics can be found in the appendix (cf. Section C.2 on p. 223).

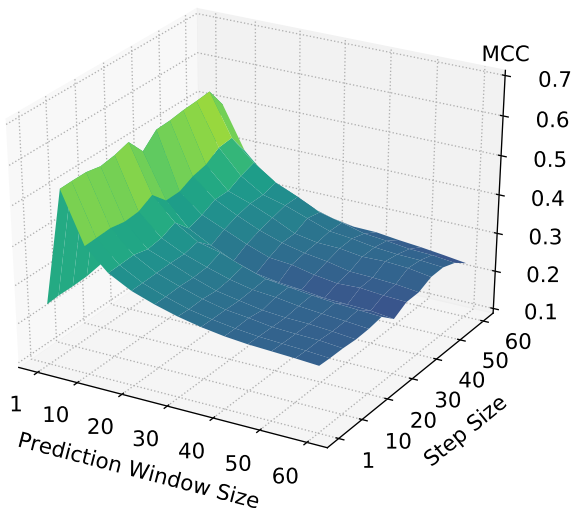
¹⁸As introduced earlier, ST-X-Y represents a slide-through (ST) sampling configuration with a step size of X and a prediction window size of Y.



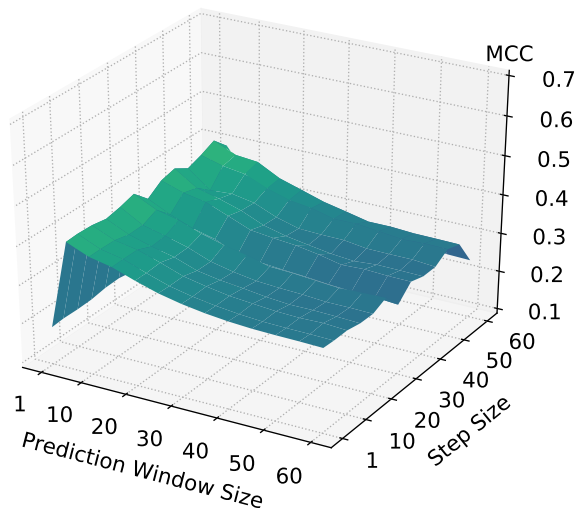
(a) MCC for the 5min observation windows.



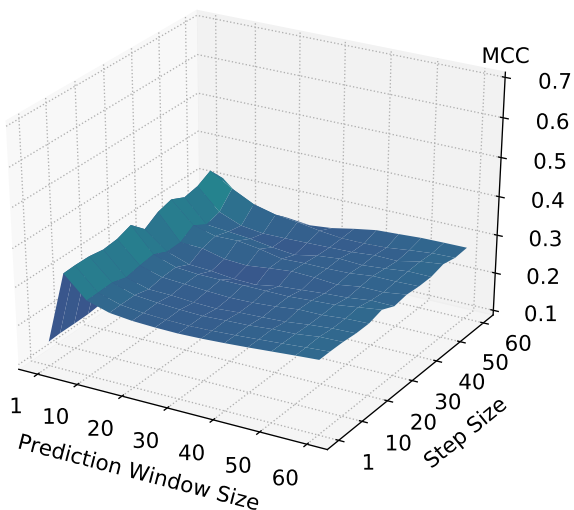
(b) MCC for the 10min observation windows.



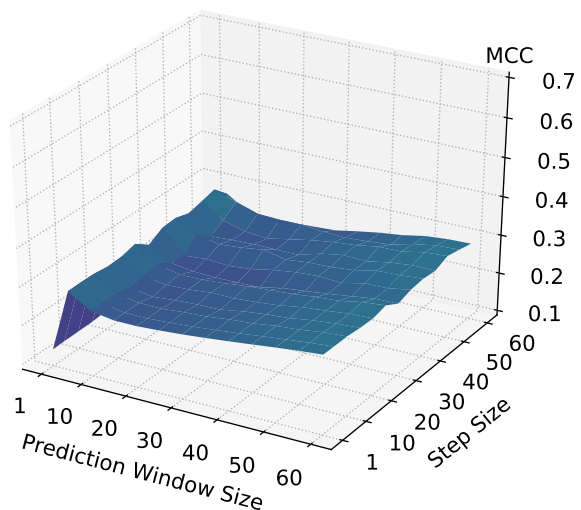
(c) MCC for the 15min observation windows.



(d) MCC for the 30min observation windows.

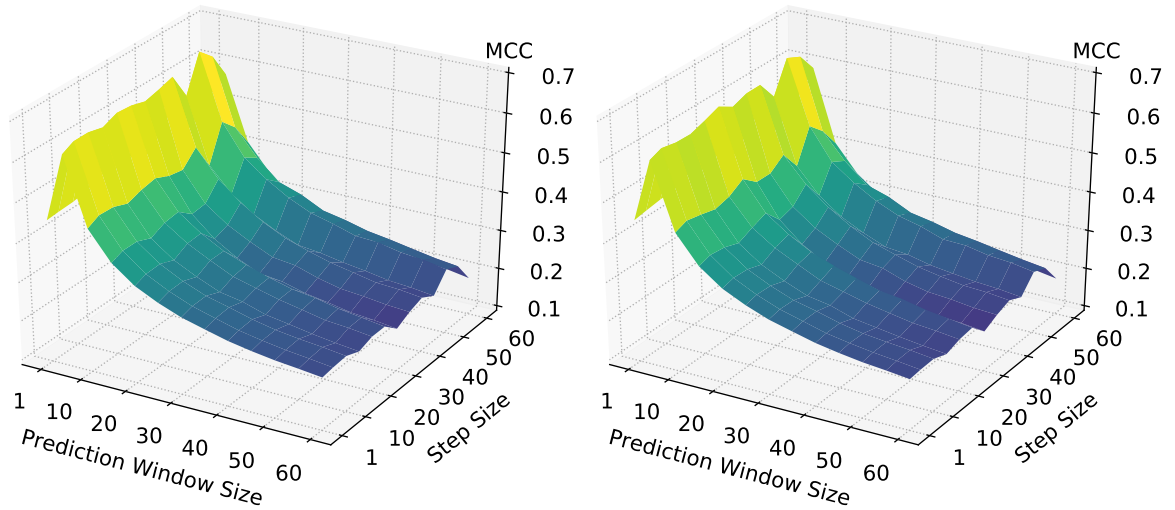


(e) MCC for the 45min observation windows.

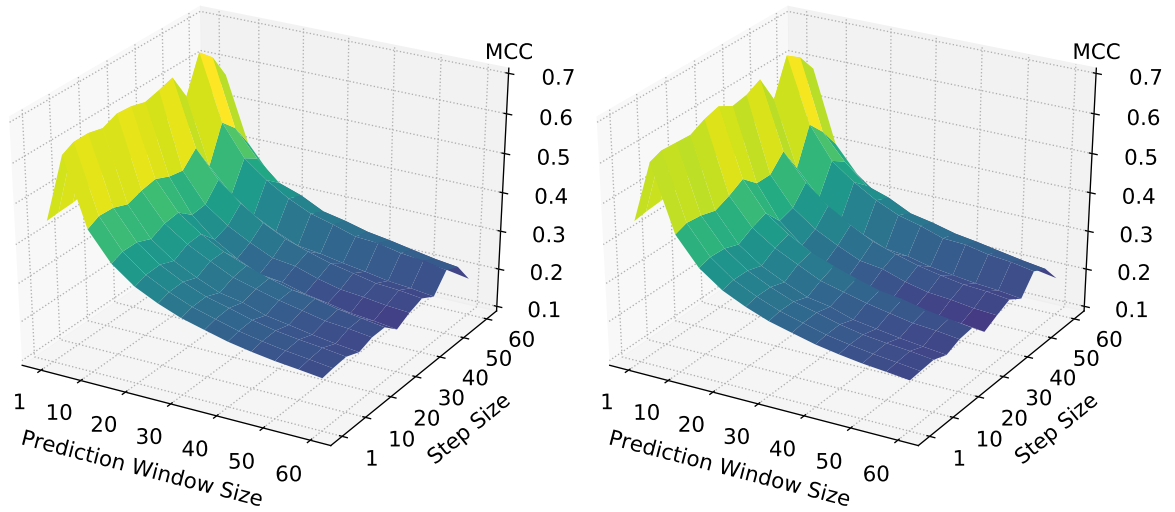


(f) MCC for the 60min observation windows.

Figure 4.28: MCC for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes. Yellow tones indicate better values, purple worse values.



(a) MCC for the 5min observation windows when trained with no augmentation (left) vs. 5-times augmented data (right).



(b) MCC for the 5min observation windows when trained with no augmentation (left) vs. 10-times augmented data (right).

Figure 4.29: MCC for the 5min observation windows after running slide-through sampling testing configurations with varying step sizes and prediction window sizes, achieved with models that were trained with 5-times and 10-times augmented data in comparison to the non-augmented data. Yellow tones indicate better values, purple worse values.

As expected, augmenting the data did not significantly change the prediction performance, indicating that the negative samples in the training data were already sufficient in the non-augmented case. To double check, we also evaluated the per-event sampling testing configurations, which resulted in the same (near perfect) predictions as well.

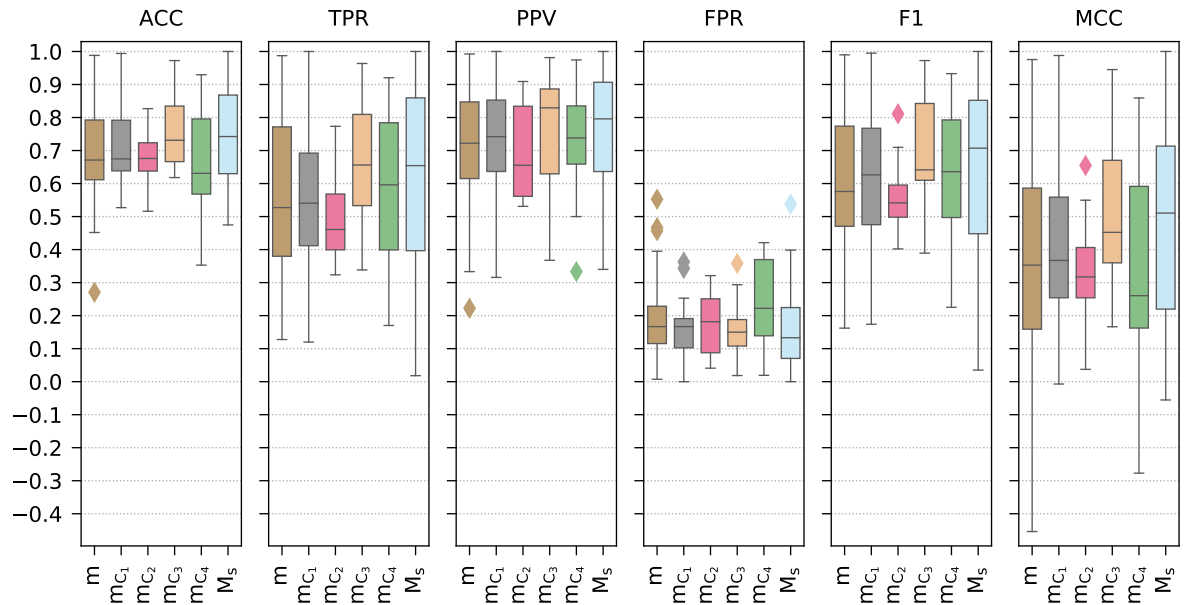
4.6.5 Balanced Scenario Revisited

We now know that the slide-through sampling approach is not optimal, but the results obtained in the unbalanced scenario are significantly worse than those in the balanced scenario. Given the initial, balanced results, we would have expected at least slightly better unbalanced results, even when accounting for the approach’s drawbacks. Recall that we had an MCC of roughly 0.75 in the balanced scenario compared to 0.03 up to 0.09 in the unbalanced scenario (depending on the concrete slide-through config). If we are conservative and round these values to 0.7 and 0.05, then this leads to a drop of $0.7 - 0.05 = 0.65$. From the synthetic evaluation, we can infer that the suboptimal slide-through sampling costs us at least 0.35, so let us again assume a more conservative value of 0.4. We subtract this “expected loss” from the original 0.7, which yields $0.7 - 0.4 = 0.3$ and thus a smaller drop of $0.3 - 0.05 = 0.25$. However, an MCC of 0.3 is still much better than our actually obtained 0.05, or, alternatively, there is still a drop of 0.25. Considering that augmenting the data resulted in even worse evaluation metrics, it might be that in the original, balanced scenario (balanced per-event sampling), we just were rather “lucky” with the randomly sampled data. Specifically, a possibility could have been that our random negative samples not only did not approximate the negative data distribution very well (cf. [Figure 4.25](#)), but also could, coincidentally, be separated/distinguished from the event samples comparatively easily. To check whether this was indeed the case, we decided to run the same 18-day cross-validation as introduced earlier,¹⁹ but this time, with both the training as well as the testing configurations set to per-event sampling. As the observation window size, we used 30 minutes. We refer to this configuration as PE (per-event sampling). Furthermore, we performed another evaluation where we ran the same training and testing configurations but with augmented training data. We refer to this configuration as PE-augmented (per-event sampling, augmented training data). This second experiment should reveal by how far the approximation of the negative sampling distribution was off. [Figure 4.30](#) shows the cross-validated results of the two configs PE and PE-augmented with details listed in [Table 4.15](#).

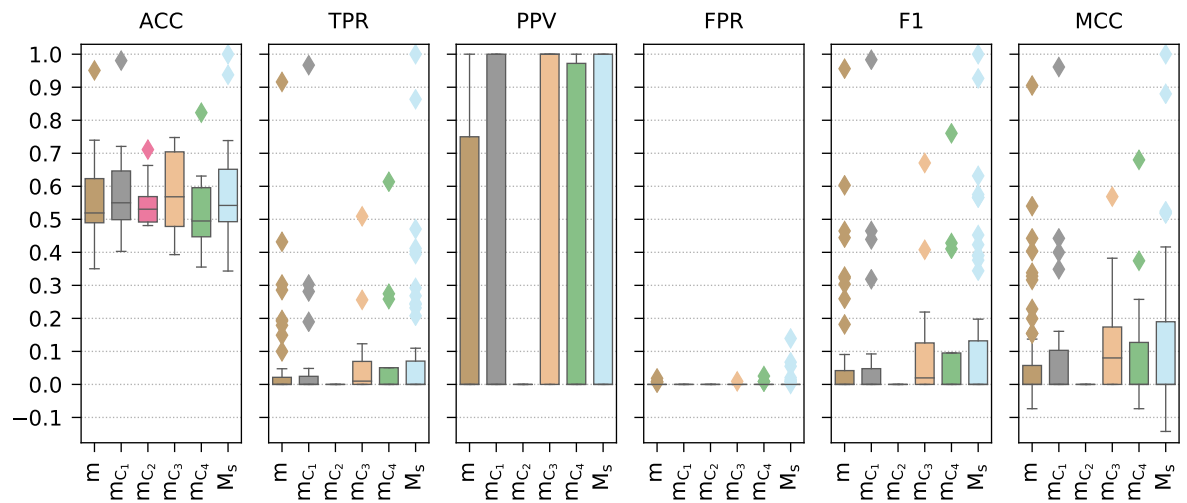
From the outcome of the first config (PE), we can clearly see that the new results are worse compared to our initial results. With the exception of the false positive rate (FPR) which increased only marginally, all metrics dropped by roughly between 0.1 and 0.3. When merged across all models, the new average MCC is around 0.4, which is about 0.3 lower than before (0.7), indicating that we were indeed “lucky” with our initial random sampling that led to quite promising first results. We now also get a much improved MCC estimation when taking the “expected loss” from the slide-through sampling approach into account. Here, we have a per-event MCC of about 0.4, and if we subtract this “expected loss” (based on the synthetic evaluation, we again assume the same conservative value of 0.4 from above), we then get an MCC estimation of $0.4 - 0.4 = 0$, which is much more in line with our actually obtained 0.05.

Finally, when analyzing the results of the second config (PE-augmented), we can conclude two things. First, as expected due to the enormous class imbalance, our initial approximation of the negative data distribution was indeed lacking, as the evaluation metrics changed significantly when augmenting the training data. Second, and more importantly, it seems that

¹⁹Since we use the day-based cross-validation, we again discard the last two days of the export (the two least complete days) to avoid “empty” days, i.e., days where no or almost no events occurred.



(a) Evaluation metrics for config PE (cf. Table 4.15a for details).



(b) Evaluation metrics for config PE-augmented (cf. Table 4.15b for details).

Figure 4.30: Evaluation metrics for the balanced test data obtained via different per-event sampling configurations with 30min observation windows after running the models (m = naive multi-system model, m_{C_i} = clustered multi-system model, M_s = set of all single-system models). Detailed information is available in Table 4.15.

	E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s		E	S	m	m_{C_1}	m_{C_2}	m_{C_3}	m_{C_4}	M_s	
ACC	μ		0.69	0.72	0.68	0.75	0.67	<u>0.74</u>	ACC	μ		0.55	0.58	0.55	0.58	0.52	<u>0.57</u>	
	σ		0.13	0.12	0.09	<u>0.11</u>	0.17	0.15		σ		<u>0.11</u>	0.12	0.08	0.13	0.12	0.12	0.12
	min		0.27	<u>0.53</u>	0.52	0.62	0.35	0.47		min		0.35	<u>0.40</u>	0.48	0.39	0.36	0.34	0.34
	p_{10}		0.54	<u>0.61</u>	0.60	0.65	0.49	0.54		p_{10}		0.45	<u>0.48</u>	0.49	0.45	0.41	0.47	0.47
	p_{25}		0.61	<u>0.64</u>	<u>0.64</u>	0.67	0.57	0.63		p_{25}		<u>0.49</u>	0.50	<u>0.49</u>	0.48	0.45	<u>0.49</u>	<u>0.49</u>
	p_{50}		0.67	0.67	0.68	<u>0.73</u>	0.63	0.74		p_{50}		0.52	<u>0.55</u>	0.53	0.57	0.49	0.54	0.54
	p_{75}		0.79	0.79	0.72	<u>0.83</u>	0.80	0.87		p_{75}		0.62	<u>0.65</u>	0.57	0.70	0.60	<u>0.65</u>	<u>0.65</u>
	p_{90}		0.87	0.88	0.76	0.87	0.91	0.92		p_{90}		0.70	0.69	0.67	0.73	0.63	<u>0.72</u>	<u>0.72</u>
max		<u>0.99</u>	<u>0.99</u>	0.83	0.97	<u>0.93</u>	1.00	max		0.95	<u>0.98</u>	0.71	0.75	0.82	1.00	1.00		
TPR	μ		0.56	0.56	0.49	0.66	0.59	<u>0.62</u>	TPR	μ		0.05	0.08	0.00	0.09	<u>0.09</u>	0.10	
	σ		0.23	0.22	0.14	<u>0.19</u>	0.24	0.26		σ		<u>0.15</u>	0.21	0.00	0.16	<u>0.18</u>	0.20	
	min		0.13	0.12	<u>0.32</u>	0.34	0.17	0.02		min		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{10}		0.23	0.33	<u>0.36</u>	0.49	0.28	0.24		p_{10}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{25}		0.38	<u>0.41</u>	0.40	0.53	0.40	0.40		p_{25}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{50}		0.53	0.54	0.46	0.66	0.60	<u>0.65</u>		p_{50}		<u>0.00</u>	<u>0.00</u>	0.00	0.01	<u>0.00</u>	<u>0.00</u>	
	p_{75}		0.77	0.69	0.57	<u>0.81</u>	0.78	0.86		p_{75}		0.02	0.02	0.00	0.07	<u>0.05</u>	0.07	
	p_{90}		0.87	<u>0.88</u>	0.62	0.84	0.86	0.91		p_{90}		0.18	0.26	0.00	0.26	<u>0.27</u>	0.33	
max		<u>0.99</u>	1.00	0.77	0.96	0.92	1.00	max		0.92	<u>0.97</u>	0.00	0.51	<u>0.61</u>	1.00			
PPV	μ		0.70	0.73	0.70	<u>0.75</u>	0.72	0.76	PPV	μ		0.27	0.39	0.00	0.63	0.37	<u>0.40</u>	
	σ		0.18	<u>0.18</u>	0.15	0.21	<u>0.18</u>	0.18		σ		<u>0.43</u>	0.50	0.00	0.50	0.49	0.48	
	min		0.22	<u>0.32</u>	0.53	<u>0.37</u>	0.33	0.34		min		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{10}		0.46	0.48	0.53	0.46	<u>0.51</u>	0.48		p_{10}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{25}		0.62	<u>0.64</u>	0.56	0.63	0.66	<u>0.64</u>		p_{25}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{50}		0.72	0.74	0.66	0.83	0.74	<u>0.80</u>		p_{50}		<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	1.00	<u>0.00</u>	<u>0.00</u>	
	p_{75}		0.85	0.85	0.83	<u>0.89</u>	0.84	0.91		p_{75}		0.75	1.00	0.00	1.00	<u>0.97</u>	1.00	
	p_{90}		0.90	<u>0.93</u>	0.91	<u>0.91</u>	<u>0.93</u>	0.95		p_{90}		1.00	1.00	<u>0.00</u>	1.00	1.00	1.00	
max		<u>0.99</u>	1.00	0.91	0.98	0.97	1.00	max		1.00	1.00	<u>0.00</u>	1.00	1.00	1.00			
FPR	μ		0.19	<u>0.16</u>	0.17	<u>0.16</u>	0.24	0.15	FPR	μ		0.00	0.00	0.00	0.00	0.00	<u>0.01</u>	
	σ		0.11	0.09	<u>0.10</u>	<u>0.10</u>	0.13	0.11		σ		0.00	0.00	0.00	0.00	<u>0.01</u>	0.02	
	min		<u>0.01</u>	0.00	0.04	0.02	0.02	0.00		min		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{10}		0.08	<u>0.05</u>	0.06	0.08	0.08	0.04		p_{10}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{25}		0.12	0.10	<u>0.09</u>	0.11	0.14	0.07		p_{25}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{50}		0.17	0.17	0.18	<u>0.15</u>	0.22	0.13		p_{50}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{75}		0.23	0.19	0.25	0.19	0.37	<u>0.22</u>		p_{75}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{90}		0.34	0.25	0.29	0.29	0.38	<u>0.26</u>		p_{90}		0.00	0.00	0.00	0.00	<u>0.01</u>	<u>0.01</u>	
max		0.55	<u>0.36</u>	0.32	<u>0.36</u>	0.42	0.54	max		0.02	0.00	0.00	<u>0.01</u>	0.03	0.14			
F1	μ		0.60	0.63	0.57	0.69	0.63	<u>0.66</u>	F1	μ		0.08	<u>0.11</u>	0.00	0.13	0.13	0.13	
	σ		0.20	0.20	0.12	<u>0.18</u>	0.21	0.24		σ		<u>0.18</u>	<u>0.24</u>	0.00	0.22	0.24	0.24	
	min		0.16	0.17	0.40	<u>0.39</u>	0.23	0.04		min		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{10}		0.33	0.43	0.46	<u>0.44</u>	0.38	0.34		p_{10}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{25}		0.47	0.48	<u>0.50</u>	0.61	<u>0.50</u>	0.45		p_{25}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{50}		0.58	0.63	0.54	<u>0.64</u>	<u>0.64</u>	0.71		p_{50}		<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.02	<u>0.00</u>	<u>0.00</u>	
	p_{75}		0.77	0.77	0.60	<u>0.84</u>	0.79	0.85		p_{75}		0.04	<u>0.05</u>	0.00	0.13	<u>0.10</u>	0.13	
	p_{90}		0.87	0.87	0.72	0.86	<u>0.90</u>	0.91		p_{90}		0.31	<u>0.42</u>	0.00	0.41	<u>0.42</u>	0.50	
max		<u>0.99</u>	<u>0.99</u>	0.81	0.97	0.93	1.00	max		0.96	<u>0.98</u>	0.00	0.67	0.76	1.00			
MCC	μ		0.38	0.42	0.34	0.50	0.35	<u>0.48</u>	MCC	μ		0.08	<u>0.12</u>	0.00	0.14	0.11	<u>0.12</u>	
	σ		0.27	0.26	0.17	<u>0.24</u>	0.34	0.30		σ		<u>0.17</u>	<u>0.23</u>	0.00	0.19	0.21	<u>0.23</u>	
	min		-0.45	-0.01	<u>0.04</u>	0.17	-0.28	-0.06		min		<u>-0.07</u>	0.00	0.00	0.00	<u>-0.07</u>	-0.14	
	p_{10}		0.09	0.08	0.21	<u>0.18</u>	0.04	0.07		p_{10}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{25}		0.16	<u>0.25</u>	<u>0.25</u>	0.36	0.16	0.22		p_{25}		0.00	0.00	0.00	0.00	0.00	0.00	
	p_{50}		0.35	0.37	0.32	<u>0.45</u>	0.26	0.51		p_{50}		<u>0.00</u>	<u>0.00</u>	<u>0.00</u>	0.08	<u>0.00</u>	<u>0.00</u>	
	p_{75}		0.59	0.56	0.41	<u>0.67</u>	0.59	0.71		p_{75}		0.06	0.10	0.00	<u>0.17</u>	0.13	0.19	
	p_{90}		0.75	0.76	0.56	0.73	<u>0.83</u>	0.84		p_{90}		0.32	<u>0.39</u>	0.00	0.38	0.35	0.41	
max		0.98	<u>0.99</u>	0.65	0.95	0.86	1.00	max		0.91	<u>0.96</u>	0.00	0.57	0.68	1.00			

(a) Config PE statistics.

(b) Config PE-augmented statistics.

Table 4.15: Statistics S of the evaluation metrics E for the balanced test data obtained via different per-event sampling configurations with 30min observation windows after running the models (m = naive multi-system model, m_{C_i} = clustered multi-system model, M_s = set of all single-system models). μ = average, σ = standard deviation, p_i = $i\%$ percentile, min = minimum, max = maximum. **Bold** = best results, underlined = second best results.

if we do have a better approximation, the negative and positive samples can no longer be successfully separated/distinguished. With the average MCC dropping to 0.1 (merged across all models), even per-event sampling, which does not have the drawbacks of the slide-through sampling approach, appears to bear no fruit. Ultimately, this might indicate that our initial hypothesis that we can predict service slowdown events based on infrastructure monitoring time series might be false, i.e., our multi-system monitoring data might just not contain enough information required to predict these events. However, much more testing would be necessary to confirm this statement, preferably also including entirely different approaches (e.g., anomaly detection) to cancel out any weaknesses and drawbacks that a classical supervised prediction approach such as ours brings along.

4.7 Discussion

This section covers a discussion of the results from both the real-world and synthetic scenarios. We also talk about our multi-system event prediction approach and present insights and lessons learned throughout the course of this project, including the encountered problems and potential threats to validity.

4.7.1 Lessons Learned

In the following, we present various lessons that we learned and general insights we gained when applying our approach on both the industrial, real-world data as well as on the synthetic dataset.

Tool support greatly facilitates the event prediction pipeline. While this might be obvious, we would still like to explicitly mention the immense workload support that our preprocessing framework enabled. Having simple configurations that are easy to change and adapt to the respective needs (e.g., different sampling techniques, balancing settings, time series data processing, etc.) drastically reduces the amount of time otherwise required for countless repetitive tasks. This is especially true for our application area, where we had to create numerous data exports, both for the real-world data and for the synthetic data.

The configuration search space is huge and can incur high computational costs. This is true for the configs of our preprocessing framework, the post-processing settings and the hyperparameter options of the selected machine learning models. Making sensible choices is essential because running every possible combination is computationally infeasible. Even when carefully picking the configuration settings, this can still quickly lead to many evaluations and potentially long computational run times, especially when dealing with large amounts of data such as our multi-system infrastructure monitoring dataset.

Slide-through sampling incurs problems for supervised learning approaches that limit the predictive capability. Our slide-through sampling approach and, in turn, our event prediction unfortunately suffer from some inherent drawbacks that cannot easily be fixed (difficult choice of parameters: overlapping observation windows, dropping events or counting multiples if the step size and prediction window size do not match). On further reflection, a classical supervised prediction approach might just not be the best choice when trying to identify and predict rare events in a multi-system environment. Perhaps anomaly detection techniques based on time series analysis [99, 101] might be better suited, which could be an interesting part of future research.

Initial results might be misleading. A thorough initial analysis is essential, since otherwise, one might be tempted to draw conclusion too quickly. Some might view this as self-evident,

but even in rigorous environments, mistakes can happen and sometimes potential issues can be overlooked, which is why we still want to mention it here. In our case, the first prediction results looked quite promising, so we directly continued with the slide-through sampling approach without taking a closer look at these results. While the subsequent evaluations were interesting and insightful, we nevertheless should have done more testing beforehand, which would have explained some things earlier and saved us a lot of unnecessary work and frustration, although the proverb “hindsight is easier than foresight” definitely applies here.

4.7.2 Problems and Limitations

The preprocessing framework only supports a limited number of preprocessing options. However, we carefully integrated all available options to cover a wide range of functionality, which should suffice for the majority of things the users want to accomplish. Moreover, new configuration settings that are required to fit more specific needs can easily be integrated. Our focus was on time-series-based event prediction using sampling techniques, so if the users’ goals do not match, e.g., just extracting data from the time series without any information about events, our framework cannot directly be applied since changes and adaptations are necessary.

The next potential problem is the fact that our data export put some strain on the infrastructure of the different software systems, which could have caused more service events. However, even if we were responsible for some additional events, the hypothesis that it is reflected in the infrastructure time series still holds, maybe even more so since those events were most likely caused due to our own increased load.

As already mentioned in the previous section, the number of configuration settings to evaluate can be a hard limit. One major factor in this regard are the available hardware resources we had at our disposal for running all tests on the industrial, real-world data.²⁰ All exports, preprocessing, post-processing and model training were done on a mobile workstation with an Intel Core i7-7820HQ 2.9GHz processor with four physical cores and eight threads, 32GB of main memory and a 1TB SSD (solid-state-drive) for storage. While this might not sound too bad at first, it must be noted that we had to process huge amounts of data, which quickly caused problems such as too little free disk space,²¹ out of memory errors²² or very long running processes.²³ Thus, we decided to run only a limited number of carefully chosen configurations, since we simply could not test more due to resource and time constraints.

All model training in the evaluation section was conducted with a lead time of zero, meaning that we simply wanted to predict whether an event occurred right after the observation window. Of course, in a production environment, we would be much more interested in lead times greater than zero, so system administrators could actually take preventative actions before an event occurs. Due to the rather poor results, the problems with the approach itself and limited hardware as well as time resources, we decided not to pursue such configurations any further, so this remains an open challenge to be studied in future work.

²⁰Due to confidentiality reasons, we could not use any additional resources.

²¹The compressed InfluxDB time series database alone takes up over 550GB of space, next to all other necessary applications and data (operating system, various programs and tools, user data). Extracting data with our preprocessing configs, notably slide-through and augmented datasets, therefore often resulted in running out of disk space. A solution to this recurring problem was only to delete (parts of) the extracted data after a successful evaluation. Naturally, in case of any changes to our evaluation pipeline, everything had to be extracted and run again, which was rather tiresome and tedious, especially considering the long run times.

²²For the augmented data with hundreds of millions of individual values, we even had to switch to a 32bit floating point number representation in order to successfully execute our evaluation pipeline.

²³Exporting data with the preprocessing framework often took hours, and evaluating the slide-through sampling configurations with augmented training data even ran for several days.

Lastly, a major problem was also the quality and the amount of the available data. Many time series had missing data and many systems had to be excluded due to too few events, which severely limited the actually usable data. It is difficult to say whether more data (e.g., several months), including more service slowdown events and ideally less missing time series values, would have impacted the outcome of the study, but it would at least have increased our confidence in our findings.

4.7.3 Threats to Validity

As already mentioned several times, the slide-through sampling approach is not ideal for the event prediction application. However, the evaluations in the synthetic scenario show that some configurations are much better than others, so the configurations we chose in the real-world scenario might potentially not be the best, i.e., we might have missed some which could have yielded better results. This threat also extends to all post-processing settings and to the selected supervised machine learning models. Regarding the latter, in our research group, Kahlhofer [90] measured the prediction performance using neural networks and achieved similar results. As pointed out in the previous section, all this would unfortunately require much more time and thus remains an open issue to be investigated in future work.

Despite the problems that the slide-through sampling introduces, it is the only sensible way of extracting data in a real-world environment because we do not know the events in advance. Hence, we cannot apply per-event sampling, even if we acknowledged the class imbalance by setting an appropriate sampling ratio. This inherent issue strengthens the point we discussed in the lessons learned, where we came to the conclusion that different approaches (e.g., anomaly detection) could be more suitable compared to the classical supervised prediction approach. Note that this problem is still specific to the data structure we analyzed. If, for instance, events are not based on a single point in time but are rather defined over a time period, the entire situation changes and everything must be reevaluated.

Independent of the chosen approach, some issues might arise from the infrastructure monitoring data itself. One question is whether the resolution of one minute is enough or too coarse. The initial hypothesis was that the service events could be caused by an “accumulative” effect (build-up), e.g., the free disk space slowly decreases or the memory consumption increases, causes garbage collections and thus high CPU load, which then ultimately triggers an event. Such a step-wise accumulation is very likely to occur over an extended period of time rather than spontaneously. Moreover, the events that are created are based on a heuristic that takes historic data in minute resolution into account as well, so we are confident that the resolution of one minute is sufficient. However, there is still the ultimate issue that it is unclear whether the service slowdowns can even be explained with the available time series data, as the ground truth is unknown (all the experiments are based on the initial hypothesis). If the data does not contain such information, there simply are no correct configurations, post-processing options, machine learning models or approaches in general because we cannot learn from something that is not there in the first place.

There are also some potential threats concerning our synthetic data. First, the best step size and prediction window size might only count for this dataset, and these options might also be a lucky coincident because they possibly simply matched the event distance most closely. For instance, assume that events started at timestamp 0 and happened every 40 minutes, then slide-through sampling configurations such as ST-40-Y (replace Y with any prediction window size) will most likely perform very well since they exactly match the event occurrences, which is comparable to per-event sampling. However, events are generated randomly based on a uniform distribution, which mitigates this problem since it is highly improbable that multiple

events occur exactly the same time period (in the example, 40min) apart. We also applied cross-validation to make full use of all events in the synthetic data, reducing the probability of such coincidences even further. A second point of discussion is the interpretation of the results achieved with unequal step sizes and prediction window sizes, which might potentially impact the evaluation metrics in a misleading way. In case the step size is larger than the prediction window size, we miss/skip events. If we skip too many, then only very few remain, where a single correct or incorrect prediction changes some metrics significantly. For example, assume the extreme case that we only observe a single event. If we classify this sample correctly as positive, then the true positive rate (TPR) is at 100%, whereas classifying the sample as negative results in a TPR of 0%. Albeit not wrong, one must take this situation into consideration and not blindly choose the configuration settings that achieved the best TPR. The same is true when inspecting the results obtained with configurations where the step size is smaller than the prediction window size. Here, we observe more events than there actually are, since the prediction windows overlap and we thus create more positive samples if an event occurred within this overlap. Again, this has an impact on the class imbalance, and we have to be careful not to draw conclusions too prematurely. If we keep both these situations in mind, inspect all the data and use a robust evaluation metric such as the Matthews Correlation Coefficient (MCC) as a first guidance, we can start interpreting the results.

The last thing to discuss are the external threats to validity. We only used time series and special/specific events from our industry partner (service events are created with custom heuristics), so we cannot really generalize to other settings and domains. The preprocessing framework as well as the event prediction approach, however, are of general value and thus could be applied to other domains where a topology, events and time series are available. However, different results and therefore different conclusions are to be expected since the outcome is very data dependent.

4.8 Related Work

This section analyzes related work and covers topics ranging from time series processing, to entire frameworks and tool pipelines, up to classification, detection and prediction systems which are based on logs or monitoring data. This also includes alternative approaches that could be interesting to apply in our multi-system environment as well.

4.8.1 Time Series Processing

While research on data processing in preparation for machine learning reaches back quite a bit in time [19], (big data) time series processing in particular has become more popular in recent years [60, 100, 66, 196]. There exists some work that does not rely heavily on preprocessing and feature engineering [91], but in general, approaches that incorporate some form of processing are widely applied. In the area of data mining, Wilson [193] listed three main processing techniques that can be used to transform and reduce the dimensionality of the raw time series data, since raw data is often not directly applicable. The techniques are piecewise approximation, identification of important points and symbolic representation. In [125], the authors investigated the classification performance of a neural network model when trained with a feature-based time series representation, which included the average, standard deviation, skewness and kurtosis (we go into much more detail on this topic in Chapter 5). They achieved better results with the processed time series variants. Jansen et al. [86] predicted machine failures based on multivariate time series, for which they created a data preprocessing pipeline covering data selection, cleaning, sampling, standardization and

more. We analyzed all this related work and extracted essential components which we then integrated into our own preprocessing framework, making adaptations where necessary to fit more special requirements imposed by our multi-system environment.

4.8.2 Time-Series-based Frameworks

Of course, we are not the first to present work on entire frameworks that incorporate time series processing. Many sophisticated frameworks have been proposed in literature, some focusing more on the data processing part itself, while others value time series handling or visualizations, or present an entire workflow pipeline. However, to the best of our knowledge, none have yet considered a multi-system environment, especially in combination with a dynamic topology with connected components and multivariate time series. Similar to their earlier work on a system for interactive design and control of a time series preprocessing pipeline [16], Bernard et al. [15] introduced an iterative segmentation workflow that includes an immediate visual analysis, so users can directly see how different segmentation parameters affect the time series processing results and make changes accordingly. The segmentation pipeline covers various steps such as data cleaning, sampling and normalization. The visual-analytics-guided framework TimeCleanser [71] was primarily designed for cleaning time series data, where the authors included 39 time-induced data quality problems derived from related work. Users can see via visual guidance how different processing techniques handle such problems, and they can also directly apply appropriate fixes. In [168], a data preprocessing framework for addressing missing values, data cleaning and data reduction with the goal of improving the prediction accuracy for datasets with missing values was proposed. They evaluated their approach with data from the area of power grid systems and concluded that using the framework improved the precision. Frenzel et al. [57] created a web-interface-based framework for handling time series data, including storage, various preprocessing options, an analysis step and visualizations. There is also one work that includes the system architecture in contrast to most other approaches that view a system as a whole or do not connect its individual components: Hora [135] is an online failure prediction framework which combines a failure propagation model (taking the system components and their connections into account) with individual component failure predictors that are based on various internal system metrics (e.g., CPU or memory measurements).

4.8.3 More General Frameworks

Of course, time series are not the only type of data that are of interest to researchers. Many more general frameworks exist that support other data kinds as well and provide other or extended processing support, some of which could also prove useful for our preprocessing framework in future work. For instance, Corrales et al. [38] presented DQF4CT as a conceptual data quality framework in the area of machine learning classification. Together with an ontology that helps users to select correct data cleaning techniques, the framework provides a guidance for preprocessing data to address quality issues. In [120], the authors proposed YALE, a large and flexible framework for various data mining tasks, which is based on so-called operators that can be arbitrarily combined into operator trees. An operator is a function that takes defined inputs, performs some actions and returns outputs. The authors provided several predefined operators that correspond to certain parts of the processing pipeline, including parsing different data sources, providing several machine learning algorithms, and offering multiple visualization and preprocessing options (e.g., discretization, filtering, normalization, sampling or dimensionality reduction). In [118], Merriënboer et al. introduced a two-part framework for preprocessing data and then training neural networks. The first part is called

Blocks and provides many preprocessing options such as standardization, data traversal and more data-specific techniques (e.g., image cropping or n-grams for texts). The second part is called Fuel, which uses Blocks and trains neural networks on the processed data, including visualizations and serialization of the results afterwards. TFX [12] is a TensorFlow-based [1] implementation of a general-purpose machine learning platform that covers many steps and components of a machine learning pipeline. The framework includes data analysis (statistical overview of the data), transformations (feature mappings), validation (detecting anomalies based on expected data that is defined via a so-called schema), model training, model evaluation and model validation (with automatic comparison of current results to previous model results), and serving (guidance how to deploy everything to production). A major focus was put on avoiding glue code, ensuring that the data is correctly applied to the machine learning phase (e.g., same processing steps for training and testing), and enabling an iterative and continuous workflow with unified configurations.

4.8.4 Log-Data-based Approaches

Continuing with the field of classification, detection and prediction, we start by presenting related work that utilizes log data. Fronza et al. [58] used log messages (must include the performed operation, the severity and the timestamp) in conjunction with Random Indexing [148] for text processing to train a support vector machine for failure prediction. They evaluated three months' worth of log data from six different subsystems of an anonymous company, preprocessed their data (duplicate removal, dropping entries with missing data) and tested four support vector machine models: three different kernels and a weighted version to handle data imbalance. The latter performed better than the rest in terms of true positive rate (TPR). In [134], the authors presented a framework for system event classification and prediction based on preprocessed event logs in a large-scale system. The first step was to process the logs, which included normalization, filtering and the manual labeling of the events by a system administrator. In the prediction part, they used sequences of log messages with the goal to find patterns that can lead to events within a specified prediction window. The authors evaluated their approach on log data from a supercomputer with various machine learning models as well as different prediction parameter settings (e.g., number of past observations, lead time or prediction window size), and they reported promising first results. Another failure prediction based on events in logs was presented in [150]. Here, the authors focused on log preprocessing to increase the prediction accuracy. They accomplished this goal by extracting event sequences (time conversions, discarding irrelevant information, assigning identifiers, and specifying the lead time as well as the data window size) with grouping/clustering and filtering those sequences afterwards. In the evaluation of a telecommunication system, they obtained 45% better results with the grouping and filtering approach. Yu et al. [199] compared the predictive capability of two online failure prediction approaches: period-based (input are n consecutive intervals of log messages) and event-driven (input are n events). The recorded log data included the identifier, the message, the affected component, the error code and severity, and a timestamp. Using a Bayesian-based predictor, they tested the two approaches on data from a high performance computing (HPC) setup, and they concluded that the period-based approach was more suitable for long-term predictions (hours up to days), whereas the results of the event-based approach indicated short-term predictability (minutes). The goal in [43] was to predict system node failures with short lead times (minutes), again in an HPC environment. The authors used unmodified log files to train a long short-term memory (LSTM) neural network, which yielded promising results. Astekin et al. [7] presented a log-based framework for unsupervised anomaly detection. The log data (timestamp, thread identifier, severity, message, etc.) is first transformed into a set of selected features, which are then normalized and put

into an unsupervised machine learning model, which acts as the anomaly detector. The last step does not require any preceding training, as the normal system behavior is automatically extracted based on the dominant components in the data (can “explain” the majority of the data), where outliers can then be considered as anomalous. In the study, the authors opted for principal component analysis (PCA) as the anomaly detector, and when applied to the manually labeled dataset introduced in [195], they obtained good results with precision and recall of about 97.6% and 66.5%, respectively.

4.8.5 Monitoring-Data-based Approaches

Similar to our setup, the approaches listed here operate on monitoring data to predict or detect different kinds of data in software systems. However, it must be noted that despite the similarity in the utilized data (monitoring metrics), the tasks and, more importantly, the events of interest are often significantly different, which emphasizes that the results can be very dependent on the concrete application domain. Williams et al. [190] developed a tool called Tiresias, a combination of a threshold-based anomaly detection and failure prediction heuristics. As data, the authors used low-level performance metrics such as CPU usage, free memory, context switches or sent packets over networks, which are collected at every node of the system. Before failures can be predicted, their approach requires fault-free runs of the system under test in order to model the normal behavior. In the evaluation, a system with three nodes was tested with manually injected failures (e.g., memory leaks or packet losses), which Tiresias was able to identify with false positive rates (FPR) as low as 2.5%. In [4], the authors inspected program crash faults. They introduced forced crashes by thread exhaustion (increasing the number of threads) and memory exhaustion (allocating more memory), and then extracted feature vectors from a set of metrics such as CPU workload, response time, disk usage or number of threads. Various machine learning techniques were applied to identify the system state (green = OK, orange = 10min before crash, red = 5min before crash) of a fixed test system with three servers. Bodik et al. [20] used performance metrics of servers of a data center for performance crisis identification. Several metrics (e.g., CPU load) are collected in epochs of x minutes, which are then summarized via quantiles and classified as cold, normal or hot based on a comparison to past, normal data. The results are then called fingerprints, optionally with reduced metrics due to a feature selection process. To identify performance crises, multiple fingerprints are grouped and then checked for their cold, normal or hot status. The authors evaluated their approach on four months of metric data exported from a production system, where the performance crises were manually labeled by a human operator. They tested different versions (e.g., all metrics vs. selected metrics) and reported good results. As already mentioned earlier, Hora [135] is an architecture-aware online failure prediction system. Metrics such as CPU or memory utilization are collected at the level of components, which are then used to incorporate component-based failure probabilities alongside a failure propagation path through the system. Sharma et al. [164] presented CloudPD to detect performance problems and faults in cloud environments. They collected various virtual machine and host metrics (e.g., page faults, context switches, latency, CPU and memory utilization or cache misses) and automatically created events based on models for normal behavior, where such events may indicate the presence of a problem or anomaly. Their problem determination engine in conjunction with their problem diagnosis engine can then be used to identify and classify anomalies. In [207], the authors described an approach on detecting performance anomalies at the task level in a black-box manner. Historic data is used to model the normal operating behavior patterns via unsupervised machine learning. As data, they used fine-grained thread-level metrics such as the command line parameters of applications, process identifiers, CPU time information, IO and network information, and

the CPU consumption of the thread. In the online detection phase, new data is then checked against these normal patterns and anomalies are reported in case of deviations, which can then be exactly pinpointed due to the fine resolution up to the individual task level. The authors manually introduced known anomalies (e.g., high CPU or memory consumption at specific points in time) in a two-server setup and reported high precision and recall rates. Tan et al. [175] used host performance metrics such as CPU and memory utilization for an adaptive online anomaly prediction system called ALERT. Depending on the platform, ALERT collects data from between 20 and 66 different metrics, which are the main driver of the anomaly detector. This detector labels the data as normal, anomaly and alert using decision trees as supervised learning models, and afterwards, clusters of common execution contexts are created and trained individually. 250 IBM servers were used in the evaluation, where the authors manually injected faults (e.g., memory leaks or infinite loops). ALERT yielded better prediction results than three other models they trained for comparison purposes. In [128], Ozcelik and Yilmaz presented work on how predictions can be improved when viewing the system under test as a white box (rather than a black box) by recording additional data with hardware counters and a minimal amount of software instrumentation. Lan et al. [99] introduced an unsupervised approach for automatically detecting system anomalies. First, data transformation is applied to handle large and inhomogeneous data (e.g., CPU, memory, IO or network metrics) from different sources. Afterwards, the feature extraction techniques principal component analysis (PCA) and independent component analysis are used to reduce the dimensionality, the results of which are then the input for an unsupervised model that extracts the normal behavior and can thus detect outliers via deviations. In the evaluation, the authors manually injected system faults that their approach could successfully identify in many cases. Dean et al. [47] also used unsupervised learning in conjunction with performance metrics (e.g., CPU, memory, disk and network metrics) to predict anomalies. Their unsupervised models of choice were Self Organizing Maps (SOM), which map high dimensional input (the metrics feature vector) into a low dimensional space. While no labels are necessary, the approach requires a training phase with data from a normal, fault-free system execution to properly initialize the SOM. The authors evaluated their implementation on various real-world distributed systems and achieved a TPR of up to 98% as well as an FPR of 1.7% with a lead time of up to 47 seconds.

4.8.6 Reliability Prediction

There also exists the area of reliability prediction regarding both hardware and software, which deals with (potentially) long-term predictions by utilizing previous, historic failures themselves rather than log or monitoring data. It would be interesting to see whether such approaches could also work in our multi-system environment. Fu and Xu [59] predicted the time-between-failure (TBF) by using information of temporal and spatial correlation of previous failures in an HPC environment. The correlation data and failure data, which is called the failure signature, are then the input for an (arbitrary) prediction algorithm. In [31], the goal was to predict the TBF of cars with the help of a support vector machine. Time series data of turbo chargers were additionally used in combination with the miles-to-failure of the engines of cars. Staying in the vehicular transportation domain, Fink et al. [55] used neural networks on 3.5 years' worth of railway turnout degradation data to make long-term predictions up to 6.5 months. The work in [5] presents details on how the TBF can be forecast using autoregressive integrated moving average (ARIMA) models. They predicted the TBF of 16 real-world software systems. Jaiswal and Malhotra [85] used various machine learning techniques such as neural networks and support vector machines to predict both cumulative failures as well as the TBF of five real-world datasets. Begum and Dohi [13] also investigated

the prediction of the cumulative number of failures in the early stages of software testing. They trained neural networks to predict failures up to 20 days. A comparison of the long-term predictive capabilities between software reliability (growth) models and data-driven models, i.e., machine-learning-based models, was presented in [130]. They evaluated the models on eight different datasets, which were later also tested with improved data-driven models [106, 107].

4.9 Outlook

There are various open issues and challenges we could tackle in future work. For example, we could test many more configurations using our own approach. While the results from our evaluation were underwhelming, different configuration settings and different machine learning models (including hyperparameter tuning) could still offer more detailed insights. However, the most obvious task would be to apply different approaches in the multi-system event prediction/detection setting. As mentioned earlier, anomaly detection techniques [99, 47, 101, 207] could be a promising start. Moreover, service slowdowns are not the only events our industry partner collects. Multiple other event types exist, so it would be interesting to investigate how well all those other events can be predicted or detected. We could then also gather more data, both in the sense of longer system observation periods (to address the significant class imbalance and obtain a higher absolute number of events) as well as more time series metrics (currently limited to host, disk and network metrics).

Given the results from our evaluation and considering the possibility that the data might not contain enough information to explain and thus predict service slowdowns, we decided to not pursue the event prediction any further. Instead, we thought about how we could minimize or perhaps avoid this uncertainty altogether, and we concluded that utilizing just the time series themselves without the events would be the ideal next step in analyzing our multi-system setting. Dropping the events and solely focusing on the time series data also has the additional benefit of drastically reducing the overall complexity, as we do no longer need to take the system topologies with all component connections into account. Nonetheless, we can still investigate many aspects of our multi-system environment, since time series analysis is a powerful tool on its own. For instance, clustering multi-system time series could reveal interesting patterns, commonalities and general insights, which is why we decided to focus on this topic in the last part of our project.

Chapter 5

Time Series Clustering

In the final chapter, we introduce an approach on how to cluster multi-system time series data with the goal to reveal interesting patterns, insights and commonalities across multiple systems. We start by proposing time series characteristics, i.e., features that can be extracted from time series, which are then used as input for various unsupervised machine learning models to cluster the data. Most parts of this chapter are described in [158] and in [152].

5.1 Motivation

Time series are ubiquitous, especially monitoring data in software systems, which continuously collect data to analyze the system state and behavior. As a consequence, the amount of data becomes increasingly larger, so automated analyses and tools are required. However, extracting information from time series is a challenging task, not only because of the huge data quantities but also because methods often only work within a certain domain or are tailored to specific systems. Researchers have thus developed a variety of approaches that range from utilizing features calculated from raw time series to reduce their dimensionality [187, 64, 3, 36, 110] and their application in numerous domains [191, 82, 101, 62, 113], up to analyzing monitoring data from real-world software systems [29] and proposing automated machine learning [75] for classification and regression problems. However, applying clustering algorithms on monitoring data to extract clusters/groups of similar time series is still an open challenge in research, especially when the data originates from multiple, different and independent software systems. We could then develop cluster-specific tools that could be applied in all the systems that are part of these clusters.

We thus propose an automated time series clustering approach that uses infrastructure monitoring data from a multi-system environment. The main objective is to find and extract clusters within the entire time series of a particular monitoring metric, such as all time series of the available memory or the CPU utilization, where entire means that we inspect the time series as a whole (in contrast to partial observation windows as we did in the event prediction chapter). Clustering multi-system data can be a useful technique to identify common patterns (e.g., spikes, periodicity/seasonality or fluctuations) and to gain general insights, which can, in turn, be used for further processing, most prominently, for building cluster-specific models and tools. There are two major benefits. First, there is no longer the need to develop tools for each system individually. If n systems have some characteristics in common that form the basis for those tools, then only a single cross-system tool needs to be created instead of n tools. The second benefit is the fact that we can use data from multiple systems to create a powerful,

“global” model.¹ For example, one possible application could be a sophisticated, aggregated forecasting model utilizing time series from multiple systems, where various internal models are used to best fit a certain time series kind, i.e., one internal forecasting model for each identified common cross-system pattern [9]. Such a model could then be even further improved by incorporating automatic method recommendation [11].

Our approach is based on time series features, i.e., we first calculate a set of characteristics/features from the (entire) raw time series data, which are then used for further processing. This means that we represent each entire time series with a set of features. For example, given a time series of length l (l data points), we calculate certain features such as the minimum, maximum and average, which results in a feature-based time series representation. Here, the l data points are represented with three features as visualized in Figure 5.1. This feature calculation is done for all n time series of our dataset. To this end, we collected a variety of characteristics from literature, but we extend them by providing a human-interpretable grouping into feature sets that cover distributional, temporal and complexity properties, and statistical tests of time series. Given labeled datasets, i.e., time series with the known, true cluster assignments (represented by the labels), we use these features set to first determine their importance to potentially filter out any non-essential feature sets. The importance is a measure of how useful and expressive a certain feature is when trying to identify the true clusters, and our approach relies on the built-in feature importance of the random forest classifier [24] (cf. Section 2.4.3.4 on p. 20). If we have several different feature sets, we check how important all individual features are. Using the example from above, it could be that the three features minimum, maximum and average are less important compared to all the other features of the different feature sets, so we might drop this particular feature set and not use it for further processing, thereby reducing our number of feature sets and thus the number of experiments we have to conduct later (beneficial in terms of computational costs). After this filtering step, we create so-called *methods*, which are triplets of the feature set, optional post-processing of the features, and the clustering model. The labeled datasets are then used once again for determining the clustering performance of all created methods, i.e., for each method, we calculate an external evaluation metric (e.g., adjusted Rand index) using the true cluster assignments from the labeled data and the cluster assignments predicted by this method. This yields a ranking of the methods best suited for clustering the provided data. Finally, we can choose one of the top-performing methods for clustering unlabeled data, e.g., to find patterns within our multi-system infrastructure monitoring time series. We then extend our approach by a run-time cost model that allows users to select the best clustering methods while also taking their run-time costs into consideration. This is a useful addition since simply always selecting the top method (which could potentially be the most expensive one in terms of run time) might not be the best choice in an industrial, real-world scenario, where computations are often outsourced to cloud-computing infrastructures, thereby incurring additional monetary costs. In the evaluation, we utilize the publicly available UCR time series archive [45, 44] as well as two real-world multi-system infrastructure monitoring datasets from our industry partner.

5.2 Data Requirements and Assumptions

Just as it was the case in the two previous chapters, our approach can be applied to all kinds of datasets as long as certain requirements are fulfilled, although they are much less restrictive in comparison, since we no longer need topologies or events. In fact, our approach is not

¹This idea is analogous to our multi-system event prediction models introduced in Chapter 4, where we used data from multiple systems in the training phase.

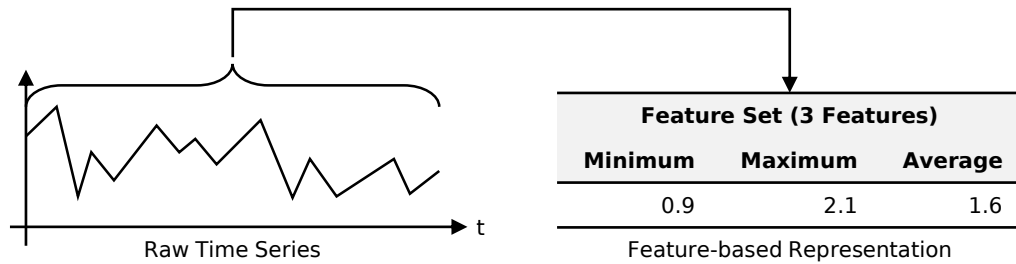


Figure 5.1: Example calculating a set of three features (minimum, maximum, average) of the (entire) raw time series data to create a feature-based representation.

limited to the software or hardware domain but can in principle be applied everywhere, as we only require time series. However, there are still some assumptions. While time series can have different lengths, they should be evenly spaced (cf. [Section 2.3.4 on p. 11](#)) because some features we calculate rely on this property. Of course, if users provide their own features that are independent of the spacing, any type of time series may be used, so this is optional. A hard requirement is that labeled data must be available in addition to the unlabeled data that should ultimately be clustered. In order for the approach to yield the best results, both the labeled as well as unlabeled data should come from the same domain, ideally, where the labeled data has the same labels that should be retrieved from the unlabeled data (e.g., via manual labeling or historic data). Often, such labeled data is not available in real-world scenarios or expensive to get, so we provide two alternatives. In the first alternative, we assume that there are different time series “sources” that are still from the same domain. A source determines how its time series are shaped and which properties and characteristics they have, and different sources should be sufficiently diverging. All sources can yield unlabeled time series since the labels are automatically determined by simply assigning the respective source as label, i.e., all time series of source *A* are assigned the label *A*, all time series of source *B* are assigned the label *B*, etc. For example, in the area of infrastructure monitoring, two such sources could be the *CPU* utilization metric and the metric measuring the free *disk* space, and all time series of the *CPU* metric (first source) would be assigned the label *CPU* and all time series of the *disk* metric (second source) would be assigned the label *disk*. The example shown in [Figure 5.2](#) illustrates this automatic label assignment. In the second alternative, no additional data needs to be provided at all, and solely publicly available labeled data such as the UCR time series archive is used instead. We discuss all the advantages and drawbacks in [Section 5.6](#) and [Section 5.7](#), where we also go into detail regarding the labeled data requirement.

5.3 Time Series Characteristics

Our approach operates on feature-based time series, so we carefully created our own time series characteristics (TSC) collection. As shown in [Figure 5.1](#), features are used instead of the raw data points to represent the (entire) time series, and a feature can be any function that takes the raw data as input and yields a feature value as output (e.g., very simple features could be the minimum, maximum or average, and more complex features could be autocorrelation or statistical tests). In our careful selection of features, we focused on avoiding excessive complexity (e.g., features that require entire models to be trained), on keeping run-time costs manageable, and on choosing a representative and diverse subset of features that capture different time series properties. We collected several features/characteristics used throughout the literature [\[187, 64, 114, 82, 36, 35, 8\]](#) and dropped redundant, highly complex and computationally expensive ones, e.g., coefficients of an ARIMA model [\[81\]](#), sample

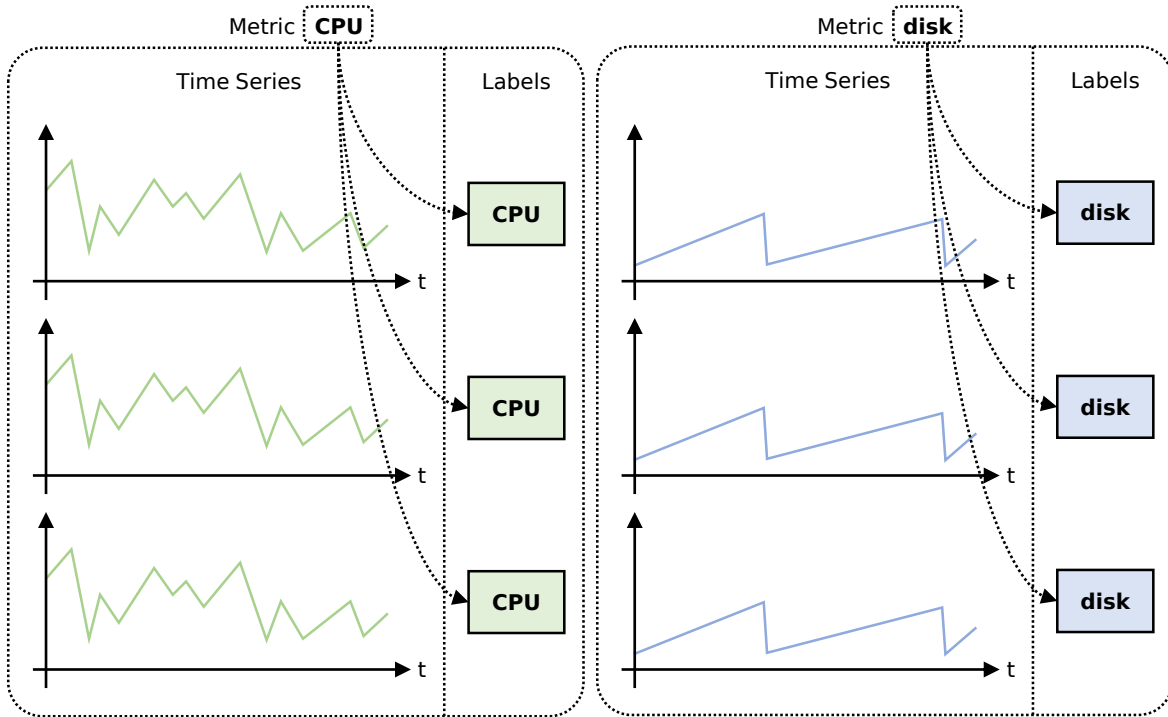


Figure 5.2: Example of the automatic label assignment based on two unlabeled time series sources: the metric *CPU* and the metric *disk*. All time series of a source/metric are assigned the corresponding metric name as label, which results in the labeled dataset.

entropy [144] or Teräsvirta et al.’s neural network linearity test [178]. We then grouped the remaining features based on which properties of the time series they represent, i.e., the “domain” of the features. Table 5.1 lists the final selection of 43 base features, assigned to the following four main groups, including more detailed subgroups:²

- *Distributional features* independent of the temporal structure of the data, i.e., viewing the time series as a set of unrelated values, e.g., various global statistics (cf. Table 5.1a).
- *Temporal features* representing the temporal structure and dependency of the data, e.g., periodicity, correlation or trends (cf. Table 5.1b).
- *Complexity features* measuring the “randomness” of a time series (can be time-dependent as well), e.g., entropy, flats or peaks (cf. Table 5.1c).
- *Test features* based on statistical tests for time series, e.g., unit root or stationarity tests (cf. Table 5.1d).

Besides improving understandability, these groups allow us to investigate certain aspects of a time series. For instance, an engineer only interested in time-independent characteristics could simply use the distributional feature set.

Some features are parameterizable, for example, the features of the *distributional dispersion blockwise* subgroup have a block size parameter b , where a time series is first separated into multiple consecutive, non-overlapping blocks of size b . Another example is the lags parameter l of the *autocorrelation* feature that specifies with how many lags l the correlation should be

²Our implementation of the selected time series characteristics is publicly available at <https://github.com/cdl-mevss-m3/Time-Series-Characteristics>.

Subgroup	Feature	Description
Disp	kurtosis	Measure of tailedness.
	skewness	Measure of asymmetry.
	shift [210]	Mean minus the median of those values that are smaller than the mean.
DispB	lumpiness [82]	Variance of the variances of blocks.
	stability [82]	Variance of the means of blocks.
Duplicates	normalized_duplicates_max	Number of duplicates that have the maximum value of the data.
	normalized_duplicates_min	Number of duplicates that have the minimum value of the data.
	percentage_of_reoccurring_datapoints	Number of unique duplicates compared to the number of unique values.
	percentage_of_reoccurring_values	Number of duplicates compared to the length of the data.
Distribution	percentage_of_unique_values	Number of unique values compared to the length of the data.
	quantile	Threshold below which $x\%$ of the ordered values of the data are, giving a hint on the distribution.
	ratio_beyond_r_sigma	Ratio of values that are more than a factor $r \cdot \sigma$ away from the mean.
	ratio_large_standard_deviation	Ratio between the standard deviation and the (max – min) range of the data (based on the “range rule of thumb” [139]).

(a) Distributional features distributed among four subgroups: dispersion (Disp), dispersion blockwise (DispB), duplicates and distribution.

Subgroup	Feature	Description
Disp	mean_abs_change	Average absolute difference of two consecutive values.
	mean_second_derivative_central	Measure of the rate of change.
DispB	level_shift [82]	Maximum difference in mean between consecutive blocks.
	variance_change [82]	Maximum difference in variance between consecutive blocks.
Sim	hurst [79]	Measure of long-term memory of a time series, related to auto-correlation.
	autocorrelation	Correlation of a signal with a lagged version of itself.
Freq	periodicity	Power (intensity) of specified frequencies in the signal (based on the periodogram [162]).
	agg_periodogram	Results of user-defined aggregation functions (e.g., fivenum) calculated on the periodogram [162].
Linearity	linear_trend_slope	Measure of linearity: slope.
	linear_trend_rvalue2	Measure of linearity: r^2 (coefficient of determination).
	agg_linear_trend_slope	Variance-aggregated slopes of blocks.
	agg_linear_trend_rvalue2	Mean-aggregated r^2 of blocks.
	c3 [159]	Measure of non-linearity (originally from the physics domain).
	time_reversal_asymmetry_statistic [160]	Asymmetry of the time series if reversed, which can be a measure of non-linearity.

(b) Temporal features distributed among five subgroups: dispersion (Disp), dispersion blockwise (DispB), similarity (Sim), frequency (Freq) and linearity.

Subgroup	Feature	Description
Entropy	binned_entropy	Fast entropy estimation based on equidistant bins.
	kullback_leibler_score (KL score) [82]	Maximum difference of KL divergences between consecutive blocks, where the KL divergence is a measure of how two probability distributions differ [96].
	index_of_kullback_leibler_score [82]	Relative location where the maximum KL score was found.
Comp	cid_ce [10]	Measure of complexity invariance.
	permutation_analysis (custom)	Measure of complexity through permutation. Details for this custom feature are provided in the appendix in Section D.1
	swinging_door_compression_rate [25]	Compression ratio of the signal under a given error tolerance ϵ .
Flatness	normalized_crossing_points	Number of times a time series crosses the mean line (based on <i>fickleness</i> [114]).
	normalized_above_mean	Number of values that are higher than the mean.
	normalized_below_mean	Number of values that are lower than the mean.
	normalized_longest_strike_above_mean	Relative length of the longest series of consecutive values above the mean.
	normalized_longest_strike_below_mean	Relative length of the longest series of consecutive values below the mean.
	flat_spots [82]	Maximum run-length of values when divided into quantile-based bins.
Peaks	normalized_number_peaks	Number of peaks, where a peak of support n is defined as a value which is bigger than its n left and n right neighbors.
	step_changes [103]	Number of times the time series significantly shifts its value range.

(c) Complexity features distributed among four subgroups: entropy, complexity miscellaneous (Comp), flatness and peaks.

Subgroup	Feature	Description
-	adf [51]	Augmented Dickey-Fuller (ADF) test for unit root presence.
	kpss [97]	Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test for stationarity.

(d) Statistical test features without any further subgroups.

Table 5.1: Features of our time series characteristics (TSC), distributed among four main groups and 13 subgroups. Features without an explicit reference all refer to [36, 35].

Group	Subgroup	#BF	#PF
Distributional	Dispersion	3	3
	Dispersion blockwise	2	10
	Duplicates	5	5
	Distribution	3	16
Temporal	Dispersion	2	2
	Dispersion blockwise	2	10
	Similarity	2	17
	Frequency	2	17
	Linearity	6	44
Complexity	Entropy	3	13
	Complexity miscellaneous	3	5
	Flatness	6	15
	Peaks	2	8
Statistical Tests	-	2	2

Table 5.2: Time series characteristics (TSC): number of base features $\#BF$ (43 in total) and number of parameterized features $\#PF$ (167 in total with the default parameterization) per (sub)group.

calculated. Other examples include parameters for thresholds (e.g., parameter r of the `ratio_beyond_r_sigma` feature) or functions (e.g., minimum, maximum and average when calculating aggregated periodogram statistics with feature `agg_periodogram`). Parameterizable features yield results for each parameter value, increasing the total number of features. Given the 43 base features, our default parameterization yields 167 individual feature values, which is presented in more detail in [Table 5.2](#).

To ensure comparability with other features and to enable easier interpretation, we normalized the selected features. Moreover, the normalization supports clustering, since many unsupervised models rely on distance measures which are affected by scale (cf. [Section 2.4.4 on p. 21](#)), i.e., high-value features would dominate the distance calculation. Some features are inherently normalized (e.g., all percentages), while others can be normalized by calculating their theoretical minimum and maximum (e.g., `binned_entropy` with $\min = 0$ and $\max = \log b$, where b is the number of bins). In a few cases, the normalization can only be done under the assumption that the data is standardized (zero mean, unit variance), so all time series are standardized by default beforehand. However, some features are theoretically unbounded (e.g., `skewness` or `shift`) and thus need to be handled differently. We address this problem with a 2-pass approach, first determining the sample minima and maxima of the dataset of interest, and then robustly scaling the corresponding features with the 5% percentile as lower bound and the 95% percentile as upper bound. This ensures that 90% of the data is within the interval $[0, 1]$ without outliers distorting the results.

5.4 Approach

Our approach can be split into multiple, partially optional stages, where an overview is presented in [Figure 5.3](#). As mentioned when discussing the data requirements in [Section 5.2](#), we need both labeled data for choosing the right *method* and unlabeled data on which we then perform the actual clustering. A method is a triplet consisting of an unsupervised clustering model, a

feature set and potential options how to post-process the feature set. Starting with a collection of features, we first create various feature sets $\mathbf{F} = \{F_1, F_2, \dots, F_f\}$ that we want to investigate, where each set F_i consists of selected features. For example, the TSC groups and subgroups introduced in the previous section could be such feature sets. Next, we optionally determine their feature importance using the labeled data in conjunction with a random forest classifier, where we leverage the random forest's built-in feature importance (cf. [Section 2.4.3.4 on p. 20](#)). We drop the ones that are irrelevant, yielding filtered feature sets $\mathbf{F}' = \{F'_1, F'_2, \dots, F'_{f'}\}$.³ These sets can then optionally be post-processed with so-called *variants* (examples for variants are normalizing the features of a set, scaling them to a certain value range or dropping correlated features). If so, we combine the feature sets with all variants $\mathbf{V} = \{V_1, V_2, \dots, V_v\}$, i.e., we create the cross product of $\mathbf{F}' \times \mathbf{V} = \{(F'_j, V_k) \mid j \in [1, f'], k \in [1, v]\}$. To complete our method triplets, we then combine the results with all unsupervised clustering models $\mathbf{M} = \{M_1, M_2, \dots, M_m\}$ that the users want to run on their data. Again, we create the cross product of $\mathbf{M} \times \mathbf{F}' \times \mathbf{V} = \{(M_i, F'_j, V_k) \mid i \in [1, m], j \in [1, f'], k \in [1, v]\}$, which is the set of method candidates.⁴ Utilizing the labeled data once more, we check for every method (M_i, F'_j, V_k) how well it performs, i.e., how close the predicted cluster assignments are to the true cluster assignments that are given by the labels. We can then rank all methods and present the results to the users, who can choose one of the top-performing methods for clustering the unlabeled data.

In [Figure 5.4](#), we provide a small example of the all the steps necessary to create the set of candidate methods. Assume that we have six features f_1 to f_6 and that we create three (non-overlapping) feature sets $F_1 = \{f_1, f_2\}$, $F_2 = \{f_3, f_4\}$ and $F_3 = \{f_5, f_6\}$ that use these six features. The features could be any valid function that takes the raw time series data as input and produces some output (the feature value). For instance, feature set F_2 could be $\{f_3, f_4\} = \{\text{minimum}, \text{median}\}$, i.e., feature f_3 calculates the minimum of the input time series and feature f_4 the median (cf. [Figure 5.1](#)). Using the labeled data (five time series), we calculate the three feature sets and determine their importance utilizing the built-in feature importance of a random forest classifier (cf. [Section 2.4.3.4 on p. 20](#)) to rank the three feature sets and potentially drop irrelevant/non-important ones (details are presented in [Section 5.4.1](#)). Assume that feature set F_1 turned out to be not important, so we drop this set, which results in the filtered feature sets $\mathbf{F}' = \{F_2, F_3\}$. We then combine these filtered feature sets with our two variants V_1 (e.g., clipping the features of a set to the range $[0, 1]$) and V_2 (e.g., normalizing the features of a set). Finally, we also combine our two models M_1 (e.g., k-means) and M_2 (e.g., agglomerative hierarchical clustering), which ultimately results in eight candidate methods.

We continue this example by showing the ranking procedure in [Figure 5.5](#). Given the eight candidate methods, we compare each method with all other methods to see if it performed equal, similarly, worse or better. Here, the performance is some external evaluation metric (e.g., adjusted Rand index), which we use to compare the predicted cluster assignments to the actual, true cluster assignments (details are presented in [Section 5.4.3](#)). In the example, a comparison of the methods (M_1, F_2, V_1) and (M_2, F_2, V_1) is demonstrated. From the labeled data, we know the true cluster assignments of the five samples/time series. The next step is to cluster the data (no labels are required) first with method (M_1, F_2, V_1) and then with method (M_2, F_2, V_1) , which both yield the predicted cluster assignments. Since we now have the predicted as well as the true clustering results, we can use any external evaluation metric to calculate a score. Here, we use the adjusted Rand index (ARI), which yields 1.0 for the first method and 0.167 for the second method. Clearly, method (M_1, F_2, V_1) performed better (higher ARI) than method (M_2, F_2, V_1) , which we enter in our so-called *diff-matrix* that

³If we skip the feature importance and filtering step, $\mathbf{F}' = \mathbf{F}$.

⁴If we skip the post-processing, there are no variants, and the method triplets thus become tuples $\mathbf{M} \times \mathbf{F}'$.

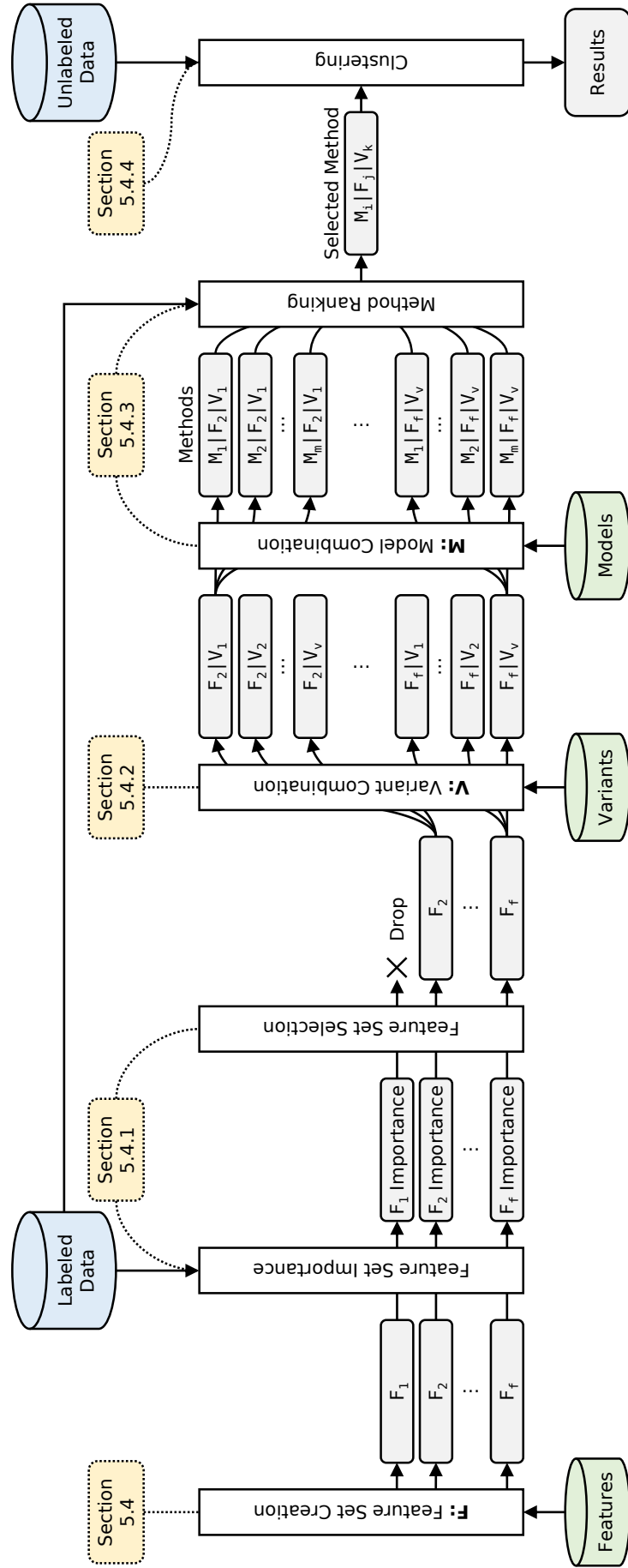


Figure 5.3: Overview of the clustering approach. The yellow annotations refer to the sections detailing the corresponding steps. The methods (M_i, F_j, V_k) are displayed as $M_i|F_j|V_k$ for optimized readability.

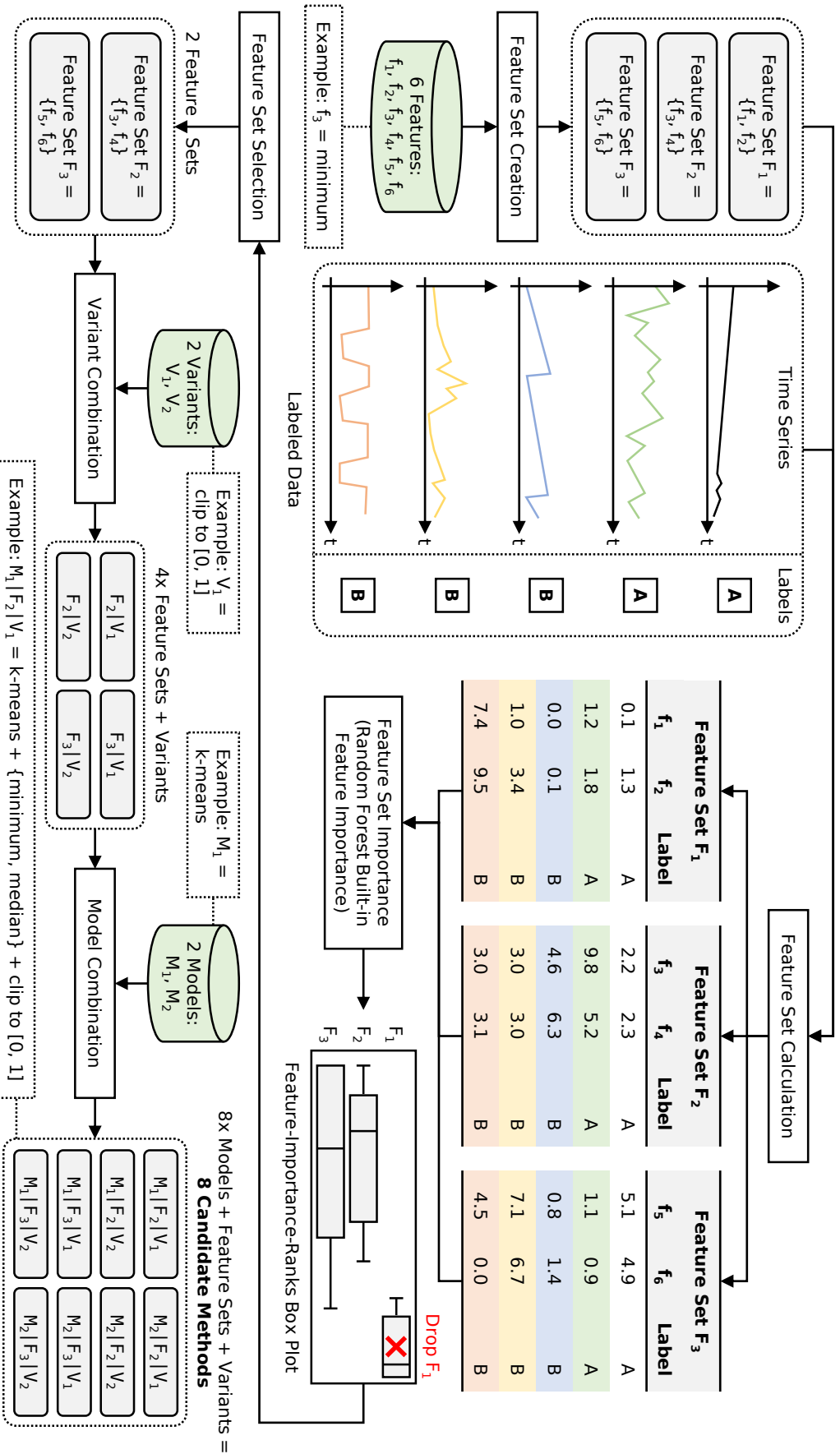


Figure 5.4: Example of creating the set of candidate methods with six features, three feature sets (two after filtering based on their importance), two variants and two models, using a labeled datasets with five time series.

contains all these method comparisons (again, details are presented in [Section 5.4.3](#)).⁵ We can sort this diff-matrix according to the “best” methods, which is user-specifiable and could, for instance, be those methods that performed better than the other methods most of the time. In the example, we can see that after sorting according to this criterion, the top-performing method is (M_1, F_2, V_1) , followed by the next best method (M_1, F_3, V_2) , etc. Users can now select one of these top-performing methods and use it for clustering the unlabeled data.

5.4.1 Determining Feature Set Importance

Using the labeled data, the first step is to investigate the importance of each feature set in \mathbf{F} by analyzing the built-in feature importance after training a random forest classifier (cf. [Section 2.4.3.4 on p. 20](#)) on *all individual* features from these feature sets. For example, if we have three feature sets $F_1 = \{f_1, f_2, f_3\}$, $F_2 = \{f_4, f_5, f_6\}$ and $F_3 = \{f_7, f_8, f_9\}$, we obtain a total of nine individual features (f_1 to f_9) that are then used to train a random forest model.

Once such a random forest is trained, we map its built-in feature importance results to scale-independent *ranks*, where rank 1 is the most important feature and the maximum rank is the least important one, i.e., lower ranks are better. In the above example, a possible feature importance ordering/ranking (cf. [Section 2.4.3.4 on p. 20](#)) of the nine features could be $f_2 \rightarrow f_1 \rightarrow f_4 \rightarrow f_3 \rightarrow f_7 \rightarrow f_9 \rightarrow f_8 \rightarrow f_5 \rightarrow f_6$ (from most important to least important), and the corresponding ranks would then be $f_2 : 1, f_1 : 2, f_4 : 3, f_3 : 4, f_7 : 5, f_9 : 6, f_8 : 7, f_5 : 8, f_6 : 9$ (sorted according to the ranks), or alternatively, $f_1 : 2, f_2 : 1, f_3 : 4, f_4 : 3, f_5 : 8, f_6 : 9, f_7 : 5, f_8 : 7, f_9 : 6$ (sorted according to the features). We can then collect the feature importance results of each feature set by combining the ranks of the corresponding features again. For feature set $F_1 = \{f_1, f_2, f_3\}$, we obtain the ranks $\{2, 1, 4\}$, for $F_2 = \{f_4, f_5, f_6\}$, we get the ranks $\{3, 8, 9\}$ and for $F_3 = \{f_7, f_8, f_9\}$, the ranks are $\{5, 7, 6\}$.

A random forest can be started with different random seeds, which might yield different results. Therefore, we repeat the training n times (ten runs) with varying seeds to take the dispersion into account, which thus yields n ranks for each feature. [Figure 5.6](#) shows all the necessary steps using the example from above. The resulting n -ranks can then be visualized with box plots, which allow us to easily inspect the performance of the feature sets and help us in deciding whether to potentially drop some irrelevant sets (resulting in the filtered feature sets \mathbf{F}'). In the example, assume that the training is repeated ten times, so each feature is ranked ten times. The individual features are put back into their corresponding feature sets, which means that F_1 , F_2 and F_3 then each consists of $3 \cdot 10 = 30$ rank results (three features, each with ten rank results, cf. [Figure 5.6](#)) that can be plotted in a feature-importance-ranks box plot as shown in [Figure 5.7](#). In the example, feature set F_1 appears to be the most important one, since it has the lowest (i.e., best) ranks on average. On the other hand, feature set F_2 seems to be significantly less important, since it has the highest (i.e., worst) ranks on average. In this case, we might consider to drop this feature set and exclude it from further processing, or more formally, $\mathbf{F}' = \{F_1, F_3\} \subset \mathbf{F}$. This step is entirely optional and can be skipped, in which case $\mathbf{F}' = \mathbf{F}$.

5.4.2 Post-Processing Feature Sets

In some cases, it might be necessary to post-process some of the feature sets, e.g., if they need to be scaled or otherwise transformed. We accomplish this by so-called *variants* that define how

⁵The diff-matrix is actually composed of multiple ARI comparisons based on the clustering results of multiple subsets of the original, labeled data. For the sake of this simple example, this step was intentionally skipped in [Figure 5.5](#).

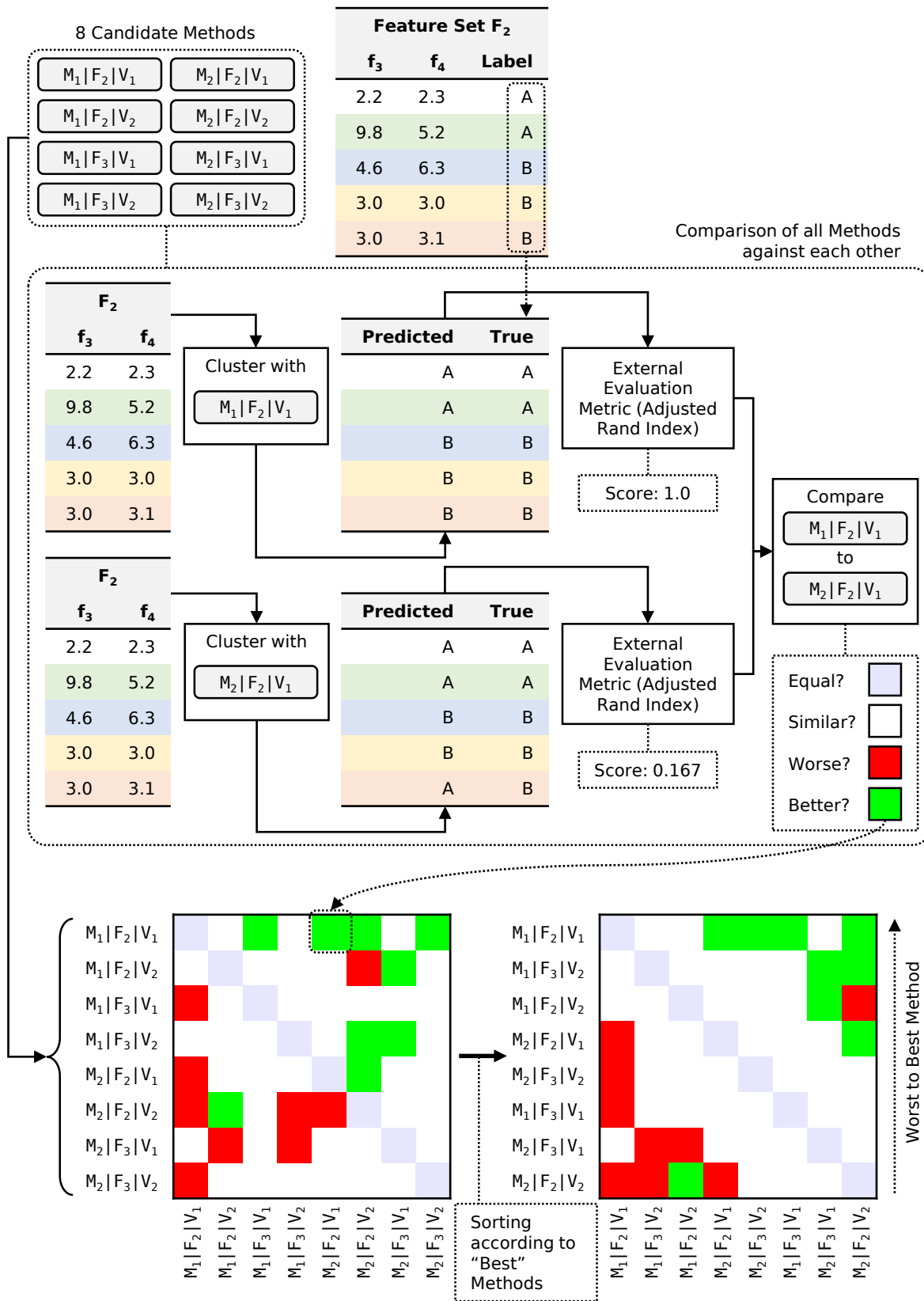


Figure 5.5: Example of the method ranking process. given the eight candidate methods and five labeled time series from the example in [Figure 5.4](#)

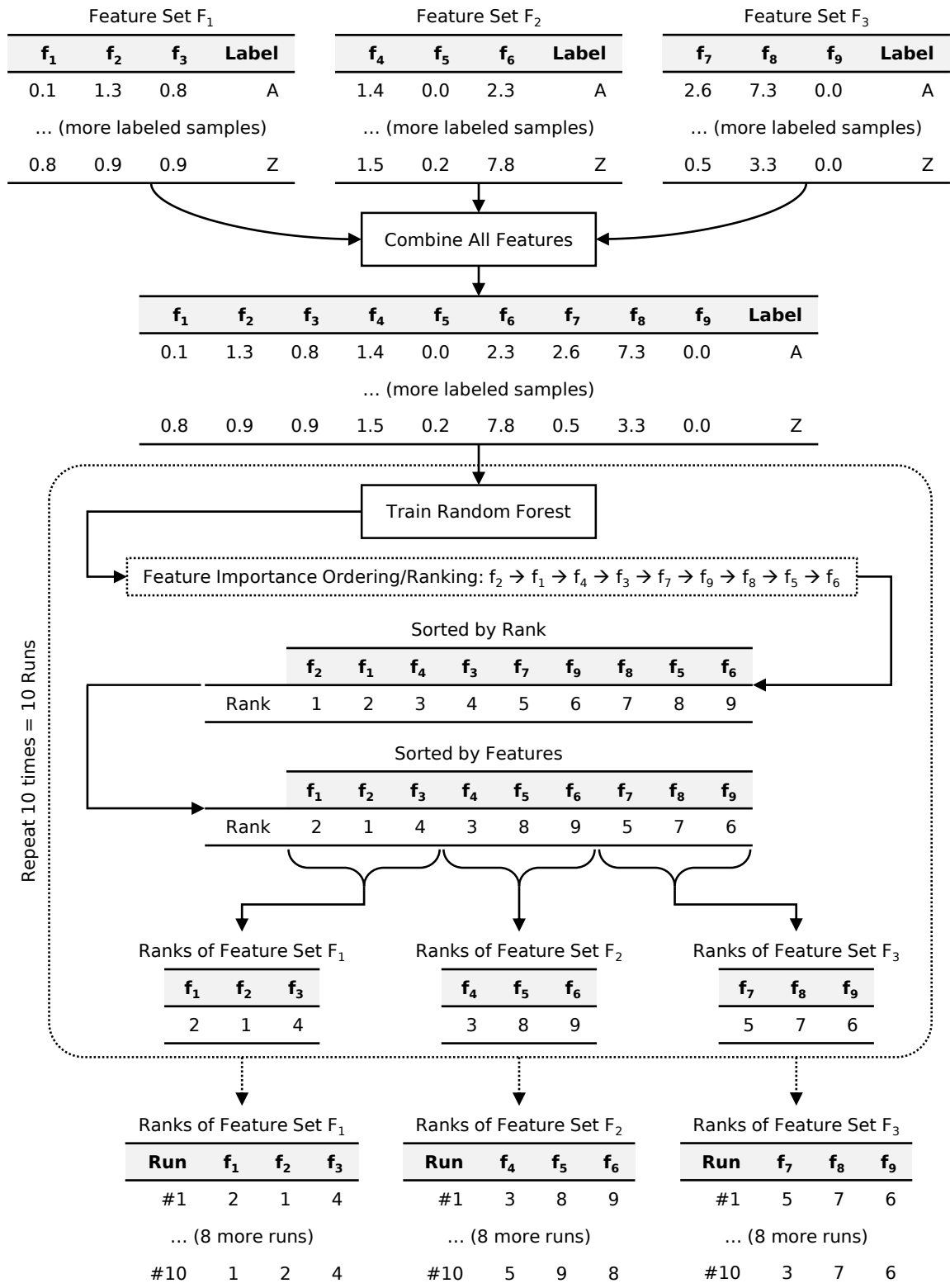


Figure 5.6: Example of the different steps to obtain the feature importance ranks for three feature sets $F_1 = \{f_1, f_2, f_3\}$, $F_2 = \{f_4, f_5, f_6\}$ and $F_3 = \{f_7, f_8, f_9\}$ with a total of nine individual features. The training of the random forest is repeated ten times (ten runs), thus resulting in ten ranks per feature at the very end.

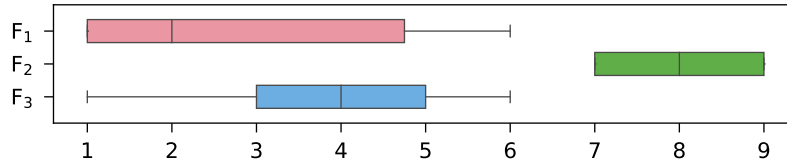


Figure 5.7: Example of a feature-importance-ranks box plot with three feature sets. The ranks range from 1 to 90, so these sets have a combined total of 90 individual features.

to post-process a set of features. For example, variants could be normalizing all the features of a feature set, scaling the feature values to a certain range, dropping correlated features or even combinations thereof (e.g. first scaling the features and then dropping correlated ones afterwards). Our approach allows us to define arbitrary variants \mathbf{V} , which can even vary between different feature sets if some of them require specific post-processing. A variant $V \in \mathbf{V}$ contains at least one post-processing option (e.g., scaling), but of course, multiple (consecutive) options are possible (e.g., scaling with dropping afterwards). This step is entirely optional and can be skipped.

5.4.3 Clustering Labeled Data

Now that we have the feature sets \mathbf{F}' and their variants \mathbf{V} , the last part to complete our method triplets is the addition of the clustering models \mathbf{M} , which yields the set of candidate methods $C = \mathbf{M} \times \mathbf{F}' \times \mathbf{V}$. Given this set and the labeled data (\mathbf{X}, \mathbf{y}) , where \mathbf{X} are *all* available time series with their respective labels \mathbf{y} (cf. the examples in [Figure 5.2](#) or [Figure 5.4](#)), we would like to know which methods perform best in terms of clustering quality. As already demonstrated in the example of [Figure 5.5](#), we have labeled data, so we know the true clustering \mathbf{y} and can easily determine the method performance with any kind of external evaluation metric such as the adjusted Rand index (cf. [Section 2.4.4.1 on p. 23](#)).

In order to make a sound comparison between the results of the different methods, we require n internal datasets \mathcal{I} based on the whole data, i.e., $\mathcal{I} = ((\mathbf{X}_1, \mathbf{y}_1), \dots, (\mathbf{X}_n, \mathbf{y}_n))$ with $(\mathbf{X}_i, \mathbf{y}_i) \subset (\mathbf{X}, \mathbf{y})$, which allow us to check whether the performance differences (if any) between the candidate methods are statistically significant. The question now is, how to obtain these internal datasets \mathcal{I} . If we are lucky, the full data \mathbf{X} might already contain such internal datasets out of the box. If not, we can easily create them ourselves by repeatedly taking random subsets of the full data. Consider the example shown in [Figure 5.8](#). The full data contains 20 labeled time series, and we simply take ten random time series to create an internal dataset. In the example, we create a total of two internal datasets, i.e., $|\mathcal{I}| = n = 2$. Some time series of these two random subsets overlap, but this does not matter since every internal dataset is subsequently processed in isolation (no influence on other datasets or results). Naturally, the sample size can be chosen arbitrarily and can be either an absolute values (like ten in our example) or percentage-based (e.g., using 5% of the full data for each internal dataset). Furthermore, we can also specify to take samples based on the classes/labels of the full data, for instance, if we want to enforce an equal class distribution in the internal datasets (which is not the case when we blindly take random samples). [Figure 5.9](#) shows the same example as above, but now the random time series are selected based on the original class. We take five samples per class, and since there are two classes (A and B), we obtain $2 \cdot 5 = 10$ random samples in total for each internal dataset. However, the class distribution is now equal, i.e., classes A and B are both represented with equally many (five) random samples per internal dataset.

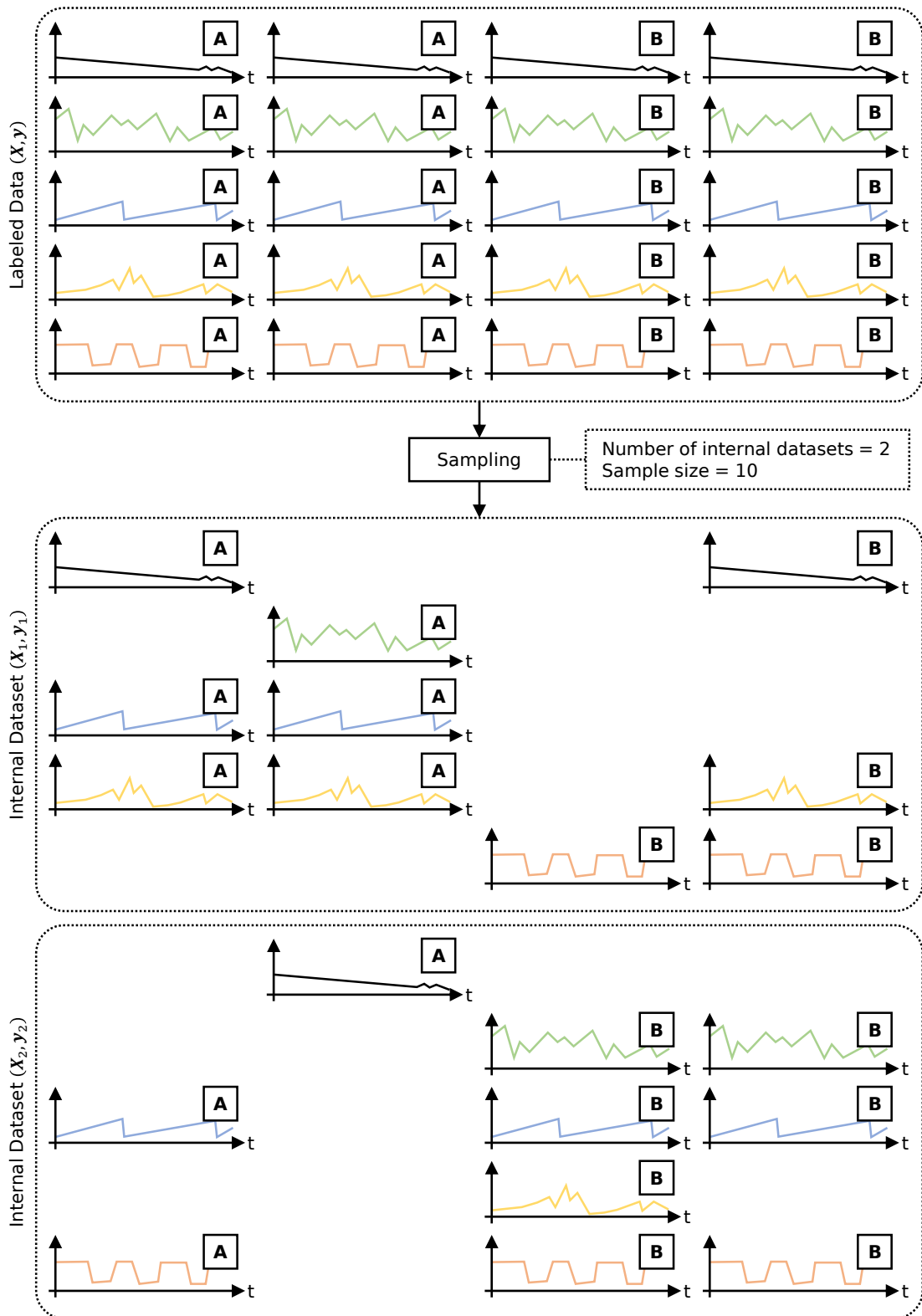


Figure 5.8: Example of creating two internal datasets (ten random samples per dataset) from the full data that contains 20 labeled time series, i.e., $\mathcal{I} = ((X_1, y_1), (X_2, y_2))$ with $(X_i, y_i) \subset (X, y)$.

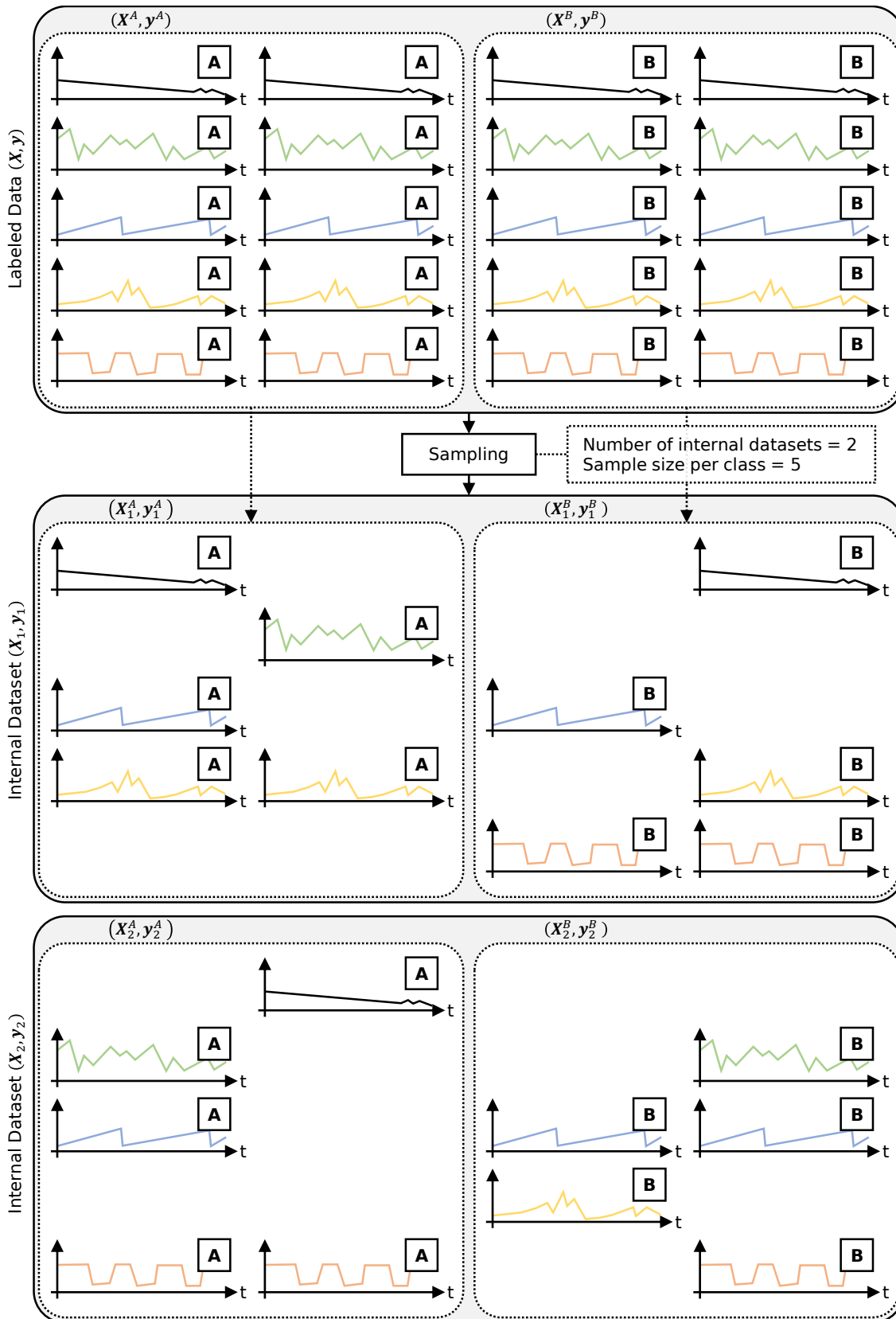


Figure 5.9: Example of creating two internal datasets (each containing five samples of the two available classes, i.e. $2 \cdot 5 = 10$ random samples per dataset) from the full data that contains 20 labeled time series, i.e., $\mathcal{I} = ((X_1, y_1), (X_2, y_2))$ with $(X_i, y_i) \subset (X, y)$. (X^A, y^A) indicates a subset that only contains the time series with labels $y = A$.

We can then run each of the candidate methods $c \in C$ on these internal datasets to get its n predicted clustering results \hat{y}_c , more formally, $\forall c \in C : \hat{y}_c = (c(\mathbf{X}_1), \dots, c(\mathbf{X}_n)) = (\hat{y}_{1c}, \dots, \hat{y}_{nc})$ with $\mathbf{X}_i \in \mathcal{I}$ and $c(\mathbf{X}_i)$ representing the clustering process with candidate method c on the internal dataset \mathbf{X}_i that results in the predicted labels \hat{y}_{ic} . A visualization of this clustering process is shown for a small example in [Figure 5.10](#), where a single method $c = (M_1, F_2, V_1)$ (feature set F_2 consists of two individual features) is used to cluster two internal datasets $(\mathbf{X}_1, \mathbf{y}_1)$ and $(\mathbf{X}_2, \mathbf{y}_2)$ with five samples each. For each of the two internal datasets, we thus get its cluster assignments predicted by this method c , i.e., \hat{y}_{1c} for the first internal dataset and \hat{y}_{2c} for the second internal dataset.

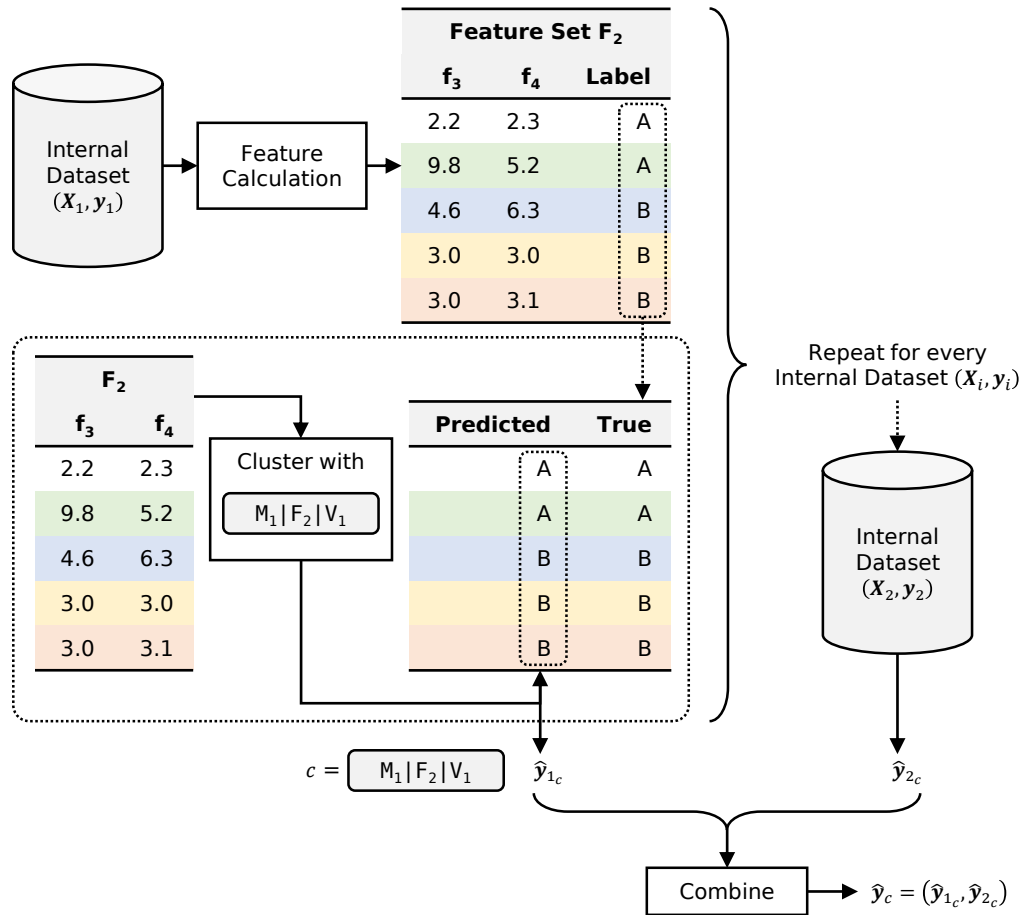


Figure 5.10: Example of clustering two internal datasets $(\mathbf{X}_i, \mathbf{y}_i)$ using a single method $c = (M_1, F_2, V_1)$ and their corresponding predicted cluster assignments \hat{y}_{ic} .

Once running all methods is completed, we compute their clustering performance by choosing the adjusted Rand index (ARI) as our external evaluation metric.⁶ For each method $c \in C$, we thus compare its n predicted clustering results \hat{y}_c to their corresponding true clustering labels \mathbf{y} and calculate n ARIs \mathbf{a}_c , more formally, $\forall c \in C : \mathbf{a}_c = (\text{ARI}(\hat{y}_{1c}, \mathbf{y}_1), \dots, \text{ARI}(\hat{y}_{nc}, \mathbf{y}_n))$ with \hat{y}_{ic} as defined above and $\mathbf{y}_i \in \mathcal{I}$. In [Figure 5.11](#), we extend the above example (cf. [Figure 5.10](#)) to also visualize this clustering performance calculation.

We can then determine whether the ARIs of the different methods are statistically significantly different and thus check which methods performed better than others. To this end, we

⁶We opted for the ARI due to its similarity to the accuracy score, which makes it an intuitive measure. Naturally, any other external evaluation metric can be selected as well.

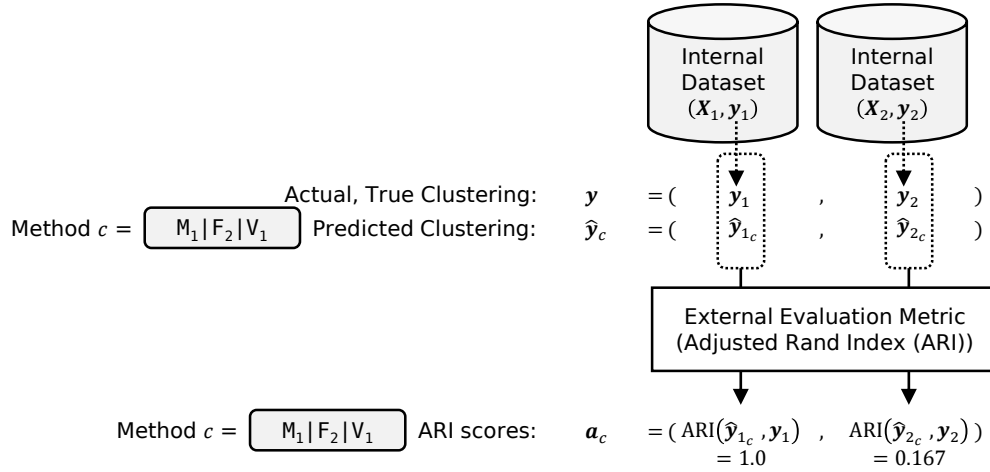


Figure 5.11: Continuing the example of [Figure 5.10](#), the clustering performance evaluation of method $c = (M_1, F_2, V_1)$ is shown using the adjusted Rand index (ARI) as external evaluation metric when comparing the true cluster assignments y_i of the two internal datasets (X_i, y_i) to the cluster assignments $\hat{y}_{i,c}$ predicted by method c .

use the Wilcoxon signed-rank test with Pratt’s method for handling zero-differences.⁷ Afterwards, we compare all possible method combinations, i.e., $\text{wilcoxon}(\mathbf{a}_c, \mathbf{a}_{c'}) \forall c, c' \in C \wedge c \neq c'$. If there are $|C| = m$ methods ($|*|$ represents the set’s cardinality), we obtain a matrix \mathbf{P} of size $m \times m$ containing the p-values of the Wilcoxon tests, where a row represents whether this method significantly differs from the methods in the columns and vice versa, given some significance level α . Since the Wilcoxon test is symmetric, i.e., $\text{wilcoxon}(\mathbf{a}_c, \mathbf{a}_{c'}) = \text{wilcoxon}(\mathbf{a}_{c'}, \mathbf{a}_c)$, \mathbf{P} is symmetric as well ($P_{ij} = P_{ji}$). In addition, we create a second matrix \mathbf{D} with the same dimensions that stores the median of all non-zero differences of the n ARIs of two methods, so we not only know if two methods are significantly different but also by how much. Since the median ARI difference is “sign-symmetric”, i.e., $\text{median}(\mathbf{a}_c - \mathbf{a}_{c'}) = -\text{median}(\mathbf{a}_{c'} - \mathbf{a}_c)$, \mathbf{D} is sign-symmetric as well ($D_{ij} = -D_{ji}$). As an extension of the above example, [Figure 5.12](#) shows how the p-values of the Wilcoxon tests as well as the median ARI differences are determined, and how the p-value matrix \mathbf{P} and the median difference matrix \mathbf{D} are created. In the example, assume that we have two models $\mathbf{M} = \{M_1, M_2\}$, two feature sets $\mathbf{F} = \{F_1, F_2\}$ and two variants $\mathbf{V} = \{V_1, V_2\}$, which results in a total of eight candidate methods $C = \{(M_i, F_j, V_k) \mid i, j, k \in [1, 2]\}$, so the two matrices \mathbf{P} and \mathbf{D} thus have a size of 8×8 . For demonstrating purposes, we compare method $c = (M_1, F_2, V_1)$ to method $c' = (M_2, F_2, V_1)$. First, we run a Wilcoxon signed-rank test based on the ARI scores of the two internal datasets, i.e., $\text{wilcoxon}(\mathbf{a}_c, \mathbf{a}_{c'})$, which results in a p-value of 0.106. Next, we calculate the differences between these ARI scores, which yields 0.001 for the first internal dataset and -0.062 for the second internal dataset. Then, we compute the median of these two differences, which results in -0.031 , indicating that method c performed slightly worse than method c' , however, this value is only considered significant if the p-value is equal to or smaller than the chosen significance level, i.e., $0.106 \leq \alpha$. We enter both the p-value and the median difference in their respective matrices \mathbf{P} and \mathbf{D} , and repeat the same procedure for all other possible method combinations. The complete matrices are listed in [Equation 5.1](#)⁸

⁷The differences are based on the ARI, and the ARI is calculated from the cluster labels, which are ordinal and hence not unlikely to be the same for two methods, resulting in equal ARIs. We thus use Pratt’s method to incorporate these zero-differences in the ranking process instead of simply discarding them.

⁸For identical clustering results of two methods, all differences are zero, so the median ARI difference is zero as well, and the p-value is NaN (not a number) because the test statistic cannot be computed.

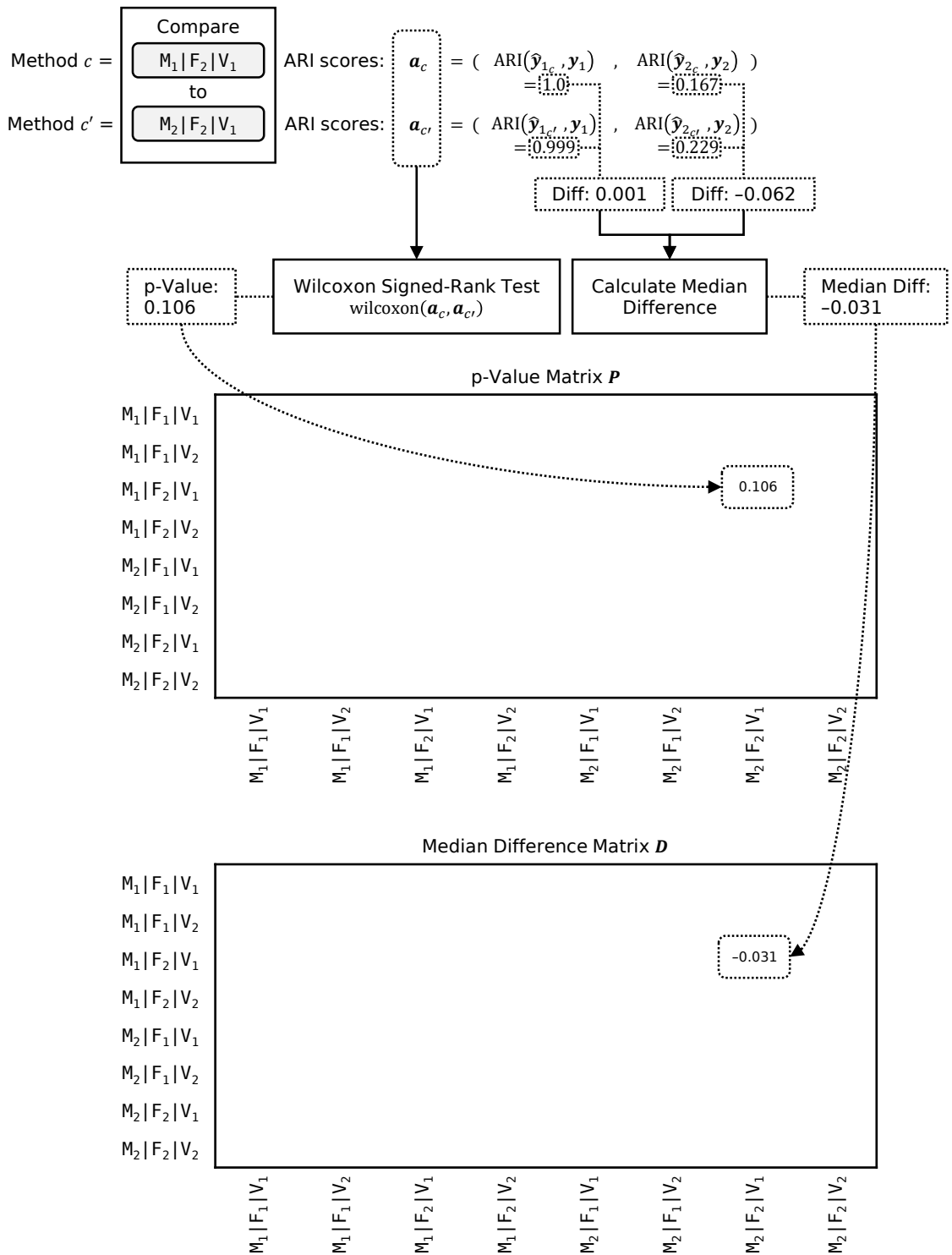






Figure 5.12: Example of comparing method $c = (M_1, F_2, V_1)$ to method $c' = (M_2, F_2, V_1)$ by calculating a Wilcoxon signed-rank test to obtain a p-value (0.106) for the p-value matrix P and by computing the ARI differences ($Diff$) of the internal datasets to obtain a median difference (-0.031) for the median difference matrix D .

$$\begin{aligned}
\mathbf{P} &= \begin{bmatrix} \text{NaN} & 0.079 & 0.275 & 0.322 & 0.324 & 0.151 & 0.203 & 0.259 \\ 0.079 & \text{NaN} & 0.012 & 0.099 & 0.020 & 0.286 & 0.124 & \text{NaN} \\ 0.275 & 0.012 & \text{NaN} & 0.119 & 0.316 & 0.073 & 0.106 & 0.306 \\ 0.322 & 0.099 & 0.119 & \text{NaN} & 0.003 & 0.249 & 0.271 & 0.025 \\ 0.324 & 0.020 & 0.316 & 0.003 & \text{NaN} & 0.082 & 0.237 & 0.108 \\ 0.151 & 0.286 & 0.073 & 0.249 & 0.082 & \text{NaN} & 0.126 & 0.202 \\ 0.203 & 0.124 & 0.106 & 0.271 & 0.237 & 0.126 & \text{NaN} & 0.051 \\ 0.259 & \text{NaN} & 0.306 & 0.025 & 0.108 & 0.202 & 0.051 & \text{NaN} \end{bmatrix} \\
\mathbf{D} &= \begin{bmatrix} 0.000 & -0.155 & -0.005 & 0.106 & 0.242 & 0.187 & -0.000 & -0.197 \\ 0.155 & 0.000 & -0.137 & 0.186 & -0.182 & -0.132 & 0.048 & 0.000 \\ 0.005 & 0.137 & 0.000 & 0.130 & -0.149 & -0.162 & -0.031 & -0.080 \\ -0.106 & -0.186 & -0.130 & 0.000 & -0.195 & -0.206 & 0.233 & -0.153 \\ -0.242 & 0.182 & 0.149 & 0.195 & 0.000 & 0.232 & -0.111 & -0.133 \\ -0.187 & 0.132 & 0.162 & 0.206 & -0.232 & 0.000 & 0.125 & -0.160 \\ 0.000 & -0.048 & 0.031 & -0.233 & 0.111 & -0.125 & 0.000 & -0.137 \\ 0.197 & 0.000 & 0.080 & 0.153 & 0.133 & 0.160 & 0.137 & 0.000 \end{bmatrix}
\end{aligned} \tag{5.1}$$

However, since analyzing the two raw data matrices is tedious, we propose a combined visualization that we call *diff-matrix*, which is shown in [Figure 5.13](#) for the above example, where we used a significance level of $\alpha = 0.1$ (for assessing the significance of the p-values obtained by the Wilcoxon signed-rank tests). To allow an easy comparison of methods, we not only display all row and column methods (can be omitted for a more compact representation⁹) but also encode the cells of this merged matrix with the following color mapping:

-  (purple, no number): This cell represents identical clustering results of the row method c_i and the column method c_j , which results in two identical ARI vectors, i.e., $\mathbf{a}_{c_i} = \mathbf{a}_{c_j}$, and, in turn, $\mathbf{P}_{ij} = \text{NaN}$ and $\mathbf{D}_{ij} = 0$. Example: row method (M_1, F_1, V_1) compared to itself.
-  (white, no number): This cell means that the row method c_i yielded different clustering results than the column method c_j , but their ARI differences are not statistically significant, i.e., $\mathbf{P}_{ij} > \alpha$ (\mathbf{D}_{ij} is irrelevant in this case because it is not significant). Example: row method (M_1, F_1, V_1) compared to column method (M_1, F_2, V_1) .
-  (red, with number x): This cell indicates that the clustering results of the row method c_i resulted in statistically significantly worse ARIs compared to those of column method c_j with a negative median ARI difference of $-0.x$ (x represents the decimal part¹⁰), i.e., $\mathbf{P}_{ij} \leq \alpha$ and $\mathbf{D}_{ij} = -0.x < 0$. Example: row method (M_1, F_1, V_1) compared to column method (M_1, F_1, V_2) .
-  (green, with number x): This cell indicates that the clustering results of the row method c_i resulted in statistically significantly better ARIs compared to those of column method c_j with a positive median ARI difference of $0.x$, i.e., $\mathbf{P}_{ij} \leq \alpha$ and $\mathbf{D}_{ij} = 0.x > 0$. Example: row method (M_1, F_1, V_2) compared to column method (M_1, F_1, V_1) .

⁹Displaying the names of the column methods c_j in addition to the row methods c_i is not necessary, since the matrix is symmetric in this regard, i.e., $c_i = c_j \forall i, j \in [1, m] \wedge i = j$, so we can safely omit the column labels, which we will later do in the evaluation (cf. [Figure 5.26](#)) to show a more compact diff-matrix.

¹⁰We use this notation to avoid a cluttered visualization. In the unlikely case that the absolute ARI difference is ≥ 1 , the text in the cell changes to $y.x$, where y represents the integer part.

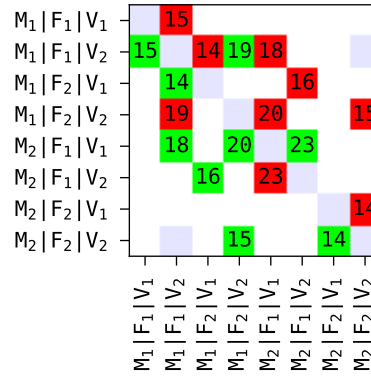


Figure 5.13: Example of a diff-matrix with eight methods. The methods (M_i, F_j, V_k) are displayed as $M_i | F_j | V_k$ for optimized readability.

Using this diff-matrix, we can now determine which methods performed best. Since “best” is not a mathematically sound term, we define the best methods as those that are statistically significantly better most times and have the highest sum of median ARI differences (of these statistically significant differences) if there is a draw.¹¹ Formally speaking, we calculate a row-based score s_i for every row method $c_i \in C$ as defined in [Equation 5.2](#):

$$s_i = \sum_{j=1}^m x_{ij} + y_{ij} \quad \text{with} \quad (5.2)$$

$$x_{ij} = \begin{cases} 0 & \text{if } P_{ij} = \text{NaN} \vee P_{ij} > \alpha, \\ 1 & \text{if } P_{ij} \leq \alpha \wedge D_{ij} > 0, \\ -1 & \text{if } P_{ij} \leq \alpha \wedge D_{ij} < 0. \end{cases} \quad \text{and} \quad y_{ij} = \begin{cases} 0 & \text{if } P_{ij} = \text{NaN} \vee P_{ij} > \alpha, \\ D_{ij} & \text{if } P_{ij} \leq \alpha. \end{cases}$$

where m is the number of methods, i.e., $m = |C|$. For example, method (M_1, F_1, V_1) has a score of $(-1 + -0.15) = -1.15$ and method (M_1, F_1, V_2) has a score of $(1 + 0.15) + (-1 + -0.14) + (1 + 0.19) + (-1 + -0.18) = 0.02$. With this score, we can then sort all rows/row methods in descending order, so the best methods are displayed at the top and the worst methods at the bottom of the diff-matrix. [Figure 5.14](#) shows this sorted diff-matrix for the above example. Here, we can clearly see that the methods (M_2, F_1, V_1) and (M_2, F_2, V_2) performed well, so we might choose one of them for the next and final step, which is clustering the unlabeled data.

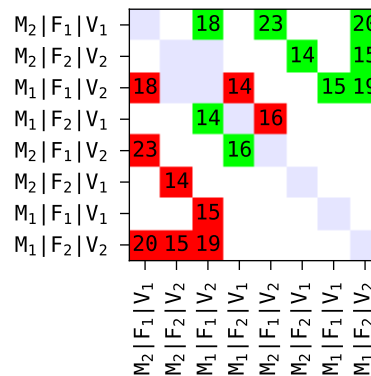


Figure 5.14: The same diff-matrix as the example in [Figure 5.13](#) but sorted by the best methods in descending order.

¹¹If the median ARI difference sum should also happen to be equal, then the methods are considered to be performing equally well.

5.4.4 Clustering Unlabeled Data

The ultimate goal is to find patterns and commonalities within the unlabeled data. Based on the results from the previous step, we know which methods performed well and thus might be suited for this task. The straightforward solution is to simply use the best method to cluster the unlabeled data, but as indicated earlier in the introduction (cf. [Section 5.1](#)), other factors could also be important to consider, such as the interpretability of the used unsupervised machine learning models or the run-time costs of the methods, which we discuss in the following section. Regardless of which method the users choose in the end, the unlabeled data is then clustered and the results can be inspected manually. In addition, we can calculate various statistics, including the number of identified clusters, their sizes and how the domain of the unlabeled data can be related to them (e.g., if we cluster multi-system data, interesting statistics are how the different systems are distributed among the clusters). All the results of clustering the unlabeled data can then be further used as input for a detailed analysis of the domain to gain more insights in general, and for the development of cluster-specific models and tools.

5.4.5 Run-Time Cost Model

While we cannot objectively assess the interpretability of clustering models, we can measure the run-time costs of the methods the users selected. We already know how to get the best methods in terms of clustering performance, and with the addition of our run-time cost model, we can then also specify how fast they are. The main goal is to measure how long it takes to calculate the feature set, to post-process this set using some variant, and how much time is required to cluster the data with the unsupervised machine learning model. Our first idea was to create a list of run-time complexity [\[170\]](#) estimates for every possible model, feature set and variant. However, there are two problems with this approach. First, it is impossible to provide a complete list, since users can select any kind of models, feature sets and variants, even their own. Second, in a real-world setting, equal run-time complexities might not always lead to actually equal run times. While this is not too surprising in general (the primary intention of complexities is to inform about the run-time behavior with increasing or very large input sizes), the differences can be quite significant.

For example, consider the four functions defined in [Figure 5.15](#) that are written in the programming language Python. Each function computes exactly the same output, namely a list containing n values in ascending order ranging from 0 to $n - 1$, but the respective implementations are different. Function 1 (cf. [Figure 5.15a](#)) represents the most straightforward way by simply appending the n individual values one after another in a `for`-loop. The code of function 2 (cf. [Figure 5.15b](#)) is identical except for the addition of the `@jit` annotation, which is part of the Numba package with the goal of compiling Python functions to optimized machine code [\[98\]](#). Function 3 (cf. [Figure 5.15c](#)) replaces the `for`-loop with a so-called *list comprehension*, which is an optimized Python-intrinsic feature. Lastly, function 4 (cf. [Figure 5.15d](#)) creates the list with the scientific computation package NumPy [\[73\]](#). If we estimate the asymptotic run-time complexity using the big-O notation [\[170\]](#), then each of these functions results in $O(n)$, which means a linear complexity.¹² However, the actual run time required to run these functions differs greatly. To get comparable results, we called each function 10000 times with $n = 1000$ and measured the total execution time.¹³ [Table 5.3](#) lists all measurements. Clearly,

¹² $O(n)$ thus means that the actual run time is expected to scale linearly with n , e.g., for $2n$, we expect all functions to take twice as long.

¹³Executed on a machine with an Intel Xeon E3-1245 v3 3.4GHz processor with four physical cores and eight threads, and 16GB of main memory.

Function 4 is by far the fastest, reducing the actual run time by approximately 79% compared to Function 1.

```
def function1(n):
    x = []
    for i in range(n):
        x.append(i)
    return x
```

(a) Function 1: standard for-loop.

```
from numba import jit

@jit
def function2(n):
    x = []
    for i in range(n):
        x.append(i)
    return x
```

(b) Function 2: compilation with Numba.

```
def function3(n):
    return [i for i in range(n)]
```

(c) Function 3: list comprehension.

```
import numpy

def function4(n):
    return numpy.arange(n).tolist()
```

(d) Function 4: implemented with NumPy.

Figure 5.15: Different Python-based implementations for creating a list of n ascending values.

Function	Run-Time Complexity	Measured Run Time
Function 1	$O(n)$	$\sim 675\text{ms}$
Function 2	$O(n)$	$\sim 380\text{ms}$
Function 3	$O(n)$	$\sim 285\text{ms}$
Function 4	$O(n)$	$\sim 145\text{ms}$

Table 5.3: The actual run time in milliseconds required for 10000 executions of the four functions introduced in [Figure 5.15](#) with an input size of $n = 1000$.

This is only one example, and other potential issues such as compiler optimizations, different language-intrinsic features and language mixtures exist. If we want to support users in choosing methods based on their run-time costs, complexity estimates are therefore not ideal. Instead, measuring the actual run times seems to be the better and more accurate option. In our approach, we thus measure how long it takes to compute the feature sets and variants, and to fit the machine learning models on a concrete machine, which should be the one used for clustering future data, since otherwise, determining the absolute run time would not make much sense. Time measurements are often unstable and can vary between executions, so we robustly measure the average run time by collecting r runs, which results in a set of measurements R . Given a lower and an upper percentile-based threshold p_l and p_u , we extract only the measurements in between and calculate the average thereof as our robust estimate of the actual run-time cost \bar{r} , which is defined in [Equation 5.3](#):

$$\bar{r} = \frac{1}{|R'|} \sum_{r' \in R'} r' \quad \text{with} \quad R' = \{r \in R \mid p_l(R) \leq r \leq p_u(R)\} \quad (5.3)$$

where $|*|$ represents the set's cardinality and p_i is the $i\%$ percentile. Now we can measure the three parts of our method triplet as described in the following:

- **Feature sets:** The run-time cost \bar{r}_F of a feature set $F \in \mathbf{F}$ is the sum of the run time costs for computing the individual features $f \in F$, given n time series of length t . In this step, we can choose to enable multi-processing using a specified number of processes, where the n time series are then distributed accordingly.
- **Variants:** For each variant $V \in \mathbf{V}$ and feature set $F \in \mathbf{F}$, we determine the run-time cost \bar{r}_V by measuring how long it takes to post-process the n feature vectors that were computed with F in the previous step.
- **Models:** For each machine learning model $M \in \mathbf{M}$, feature set $F \in \mathbf{F}$ and variant $V \in \mathbf{V}$, we obtain the run-time cost \bar{r}_M by measuring how long it takes to fit the model on the n post-processed feature vectors that were computed in the previous two steps.

Given a set of candidate methods $c \in C$ and labeled data, we can now calculate both the methods' clustering performance quality with any kind of external evaluation metric and their actual run-time costs with $\bar{r}_c = \bar{r}_M + \bar{r}_F + \bar{r}_V$. The results of all evaluated methods can be visualized with a quality-cost trade-off graph, whose x-axis represents the clustering quality and the y-axis represents the run-time costs. In this graph, we can also enable a user-specifiable lower quality threshold¹⁴ qt as well as an upper cost threshold ct , allowing us to extract only those methods that are relevant considering these thresholds, i.e., those that performed well and fast enough, more formally, the relevant methods must fulfill $e_c \geq qt \wedge \bar{r}_c \leq ct$, where e_c is the evaluation metric obtained with method c . Using the same example as introduced in [Section 5.4.3](#), i.e., eight candidate methods $C = \{(M_i, F_j, V_k) \mid i, j, k \in [1, 2]\}$, we present such a quality-cost trade-off graph in [Figure 5.16](#), where we chose the adjusted Rank index (ARI) as our external evaluation metric. For demonstration purposes, the actual run-time costs \bar{r} are assumed to be in the range of seconds (cf. *Run Time [s]*). When specifying the two thresholds of $qt = 0.1$ (ARI) and $ct = 7$ (seconds), four out of the eight candidate methods remain as relevant, of which we can calculate the Pareto front¹⁵ that leads to the final set of three relevant methods as shown in the table next to the graph. We can see that method (M_2, F_2, V_2) performed best in terms of clustering quality (the best method overall was too slow and thus filtered out), but when accepting a slight decrease in the ARI score (-0.03), method (M_1, F_2, V_1) manages to outperform this method by over 21% in run time. Ultimately, it is then up to the users which of these methods they choose for clustering.

5.5 Data for Evaluation

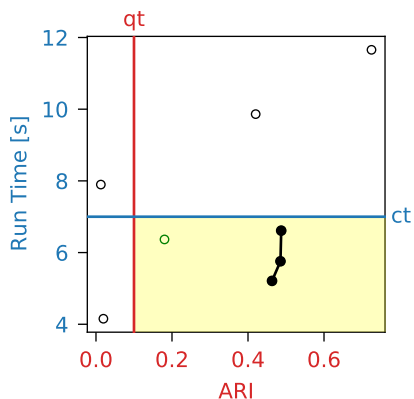
In the evaluation, we use data from two different sources. The first source is the UCR time series archive [\[45, 44\]](#), a collection of time series from various domains. The second source are two independent multi-system infrastructure monitoring time series (IMTS) datasets from our industry partner, whose structure we already described in [Section 2.3.4 on p. 11](#).

5.5.1 UCR Archive

The UCR time series archive covers a wide array of domains and different types of time series, ranging from electrical device measurements, image classification, ECG and motion data to sensor data as well as simulated data from various areas. The archive consists of 128 labeled datasets, each of which is split into a training and test set that share the same number of

¹⁴In case the external evaluation metric is a lower-is-better value, an upper quality threshold must be used.

¹⁵The Pareto front is the set of best possible methods considering both the quality and run-time cost, where neither property can be improved any more without worsening at least one of them.



Model	Feature Set	Variant	ARI	Run Time [s]
M_2	F_2	V_2	0.487	6.612
M_2	F_1	V_2	0.485	5.757
M_1	F_2	V_1	0.463	5.211

Figure 5.16: Quality-cost trade-off graph example with eight methods. The lower quality threshold (qt) of $ARI \geq 0.1$ and upper run-time cost threshold (ct) of $\bar{r} \leq 7s$ result in four relevant methods (highlighted with yellow background), whose Pareto front is listed in the table.

classes/labels and time series lengths.¹⁶ Between the different datasets, the number of samples, classes and time series lengths differ significantly, where an overview is shown in [Table 5.4](#) (the complete information can be found in the appendix in [Section D.2 on p. 252](#)).

Statistic	μ	σ	min	p_{10}	p_{25}	p_{50}	p_{75}	p_{90}	max
#Train	473.09	1107.39	16	23	53.75	190.50	400	896	8926
#Test	1020.34	2001.83	20	70.30	139	316	870.75	2850	16800
#Classes	8.73	12.03	2	2	2	4	10	24.30	60
Length	534.54	563.05	15	80	144	344	657.75	1378.80	2844

Table 5.4: Various statistics of the 128 UCR datasets, where the # character represents the number of train/test samples and unique classes, respectively, and *Length* indicates the time series length. μ = average, σ = standard deviation, $p_i = i\%$ percentile, min = minimum, max = maximum.

5.5.2 IMTS Archive

Our own data archive contains two multi-system IMTS datasets: $IMTS_1$ and $IMTS_2$. The first one is the same dataset we used in the event prediction approach (cf. [Section 4.5 on p. 82](#)), i.e., 20 days' worth of exported data from 705 software systems, however, only the time series are of interest. Since entire time series are now the input for our approach (rather than the previous observation window subsequences), we wanted them to represent full cycles of a working week to capture any weekly patterns, i.e., each time series should record entire weeks. Within the 20 export days, there are two such full cycles, resulting in a dataset of 14 days that range from 22.01.2018 00:00 UTC (Coordinated Universal Time) to 04.02.2018 23:59 UTC. Given the resolution of one minute, each time series of $IMTS_1$ thus has 20160 data points. $IMTS_2$ is similar, as the same data was collected but for different systems (eight Dynatrace-internal systems) and for a different observation period (28 days, ranging from 15.07.2019 00:00 UTC to 11.08.2019 23:59 UTC). Given the same one-minute resolution, each time series of $IMTS_2$ thus has 40320 data points. Unfortunately, both IMTS datasets are unlabeled, so we had

¹⁶There is also a version where some datasets have varying lengths. However, we plan to run the clustering models on the raw data as well, so we use the length-adjusted data.

to fall back to the first alternative as specified in our data requirements and assumptions in [Section 5.2](#), which states that for unlabeled data, we should provide sufficiently different time series “sources”. This can easily be achieved with our different metrics that are listed in [Table 2.2 on p. 12](#). Each such metric can be considered a time series source as long as we do not choose metrics that are too similar, such as the CPU system utilization metric together with the CPU user utilization. We thus decided to only use a specific subset of our 34 time series metrics, namely the following seven: CPU Idle (H-01), CPU IO Wait (H-05), Page Faults (H-06), Memory Available % (H-07), Disk Available % (D-03), Read Bytes (D-04) and Bytes Received (N-01). Since we already knew from the data exploration in the previous chapter that many data points of the time series are missing (cf. [Figure C.6 on p. 228](#)), we excluded all those series that had less than 99.9% data points available and linearly interpolated the remaining missing values. As practically constant and stagnant time series are not of interest to us¹⁷ and such series would only reduce the differences between our time series metric sources,¹⁸ we additionally dropped all those series with $\geq 99\%$ equal values. If too few time series for a specific metric remain, the entire metric is dropped.

Ultimately, IMTS_1 contains 32867 time series for five (out of the selected seven) metrics from 615 systems, and IMTS_2 contains 8216 time series for six metrics from eight systems, where a breakdown is presented in [Table 5.5](#), and detailed statistics on a per-system basis are shown in [Figure 5.17](#) ([Table 5.6](#)) and [Figure 5.18](#) ([Table 5.7](#)), respectively. Especially for IMTS_1 , the statistics reveal that the majority of the systems only provide around ten time series per metric on average and that there are a few large systems with hundreds of series. While this imbalance was an issue in the event prediction approach (system balancing necessary), the system distribution is irrelevant here, since our clustering approach solely relies on time series and the system sizes thus no longer have any impact.

ID: Metric	IMTS ₁		IMTS ₂	
	#TS	#Sys.	#TS	#Sys.
H-01: CPU Idle	7113	606	1274	8
H-05: CPU IO Wait	-	-	680	7
H-06: Page Faults	3361	437	680	6
H-07: Memory Available %	7176	608	1356	8
D-03: Disk Available %	8220	503	2259	7
D-04: Read Bytes	-	-	1967	7
N-01: Bytes Received	6997	568	-	-
Total	32867	615	8216	8

Table 5.5: Number of individual time series ($\#TS$) and number of systems ($\#Sys.$) for each collected metric and both IMTS datasets. The - character means that no time series of this particular metric are present (due to data requirements and filtering).

5.5.3 UCR and IMTS Datasets

Now that we have established the raw data sources, the last step is to create the actual datasets that are going to be the input for our clustering method selection approach in the following evaluation. As described in [Section 5.4.3](#), we need n internal, labeled datasets \mathcal{I}

¹⁷They can easily be detected and filtered out without the need for clustering.

¹⁸For example, it can happen that a time series of the available disk space metric is identical to a time series of the disk read metric in case the disk is not used throughout the observation period.

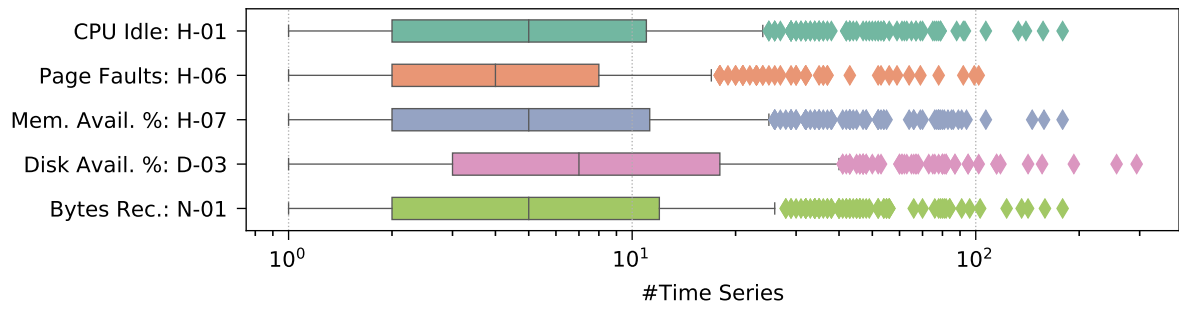


Figure 5.17: IMTS₁: System-based time series count statistics, visualized with a box plot on a logarithmic scale. Detailed information is available in [Table 5.6](#).

ID	#Sys.	μ	σ	min	p_{10}	p_{25}	p_{50}	p_{75}	p_{90}	max
H-01	606	11.74	19.49	1	1	2	5	11	29	179
H-06	437	7.69	12.54	1	1	2	4	8	17.4	102
H-07	608	11.80	19.53	1	1	2	5	11.25	29	179
D-03	503	16.34	28.32	1	1	3	7	18	41	294
N-01	568	12.32	20.43	1	1	2	5	12	33.3	179

Table 5.6: IMTS₁: System-based time series count statistics. #Sys. represents the number of systems. μ = average, σ = standard deviation, $p_i = i\%$ percentile, min = minimum, max = maximum.

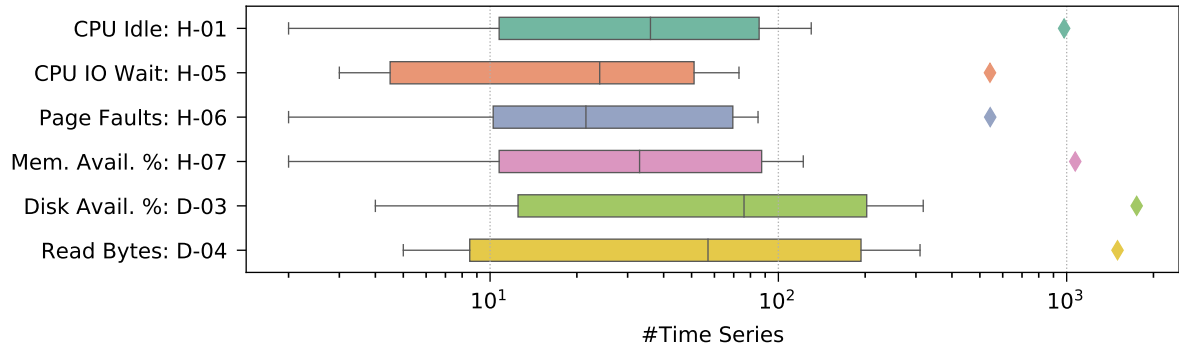


Figure 5.18: IMTS₂: System-based time series count statistics, visualized with a box plot on a logarithmic scale. Detailed information is available in [Table 5.7](#).

ID	#Sys.	μ	σ	min	p_{10}	p_{25}	p_{50}	p_{75}	p_{90}	max
H-01	8	159.25	334.52	2	5.5	10.75	36	85.75	385	980
H-05	7	97.14	197.70	3	3.6	4.50	24	51	260.6	542
H-06	6	113.33	212.60	2	4.5	10.25	21.5	69.50	314	543
H-07	8	169.50	366.65	2	5.5	10.75	33	87.50	406.7	1071
D-03	7	322.71	638.33	4	7.6	12.50	76	202.50	890.4	1749
D-04	7	281	548.59	5	5.6	8.50	57	193.50	786.4	1501

Table 5.7: IMTS₂: System-based time series count statistics. #Sys. represents the number of systems. μ = average, σ = standard deviation, $p_i = i\%$ percentile, min = minimum, max = maximum.

to compute the diff-matrices. For the UCR data, this is straightforward, since the 128 UCR datasets already fulfill this requirement ($n = 128$). We can even combine both the train and test samples because clustering is unsupervised, and hence, no separate test set is needed. The IMTS data, on the other hand, is unlabeled and does not have such internal datasets out of the box. However, we can now use the fact that we have sufficiently different time series sources given by the monitoring metrics, which allows us to create labeled data ourselves. To this end, for all time series $\mathbf{x}^k \in \mathbf{X}^k$ of a metric/time series kind k , we assign the metric itself as class label, i.e., $\forall \mathbf{x}^k \in \mathbf{X}^k : y = k$, which yields the metric class label vector \mathbf{y}^k for all time series \mathbf{X}^k (this procedure is exactly the same as shown in the example of [Figure 5.2](#) for time series kinds *CPU* and *disk*). This procedure is repeated for all m metrics $K = \{k_j \mid j \in [1, m]\}$ (e.g., for the example of [Figure 5.2](#), $K = \{CPU, disk\}$), and then we merge everything into a single dataset, more formally, $\mathbf{X} = (\mathbf{X}^k \mid k \in K)$ and $\mathbf{y} = (\mathbf{y}^k \mid k \in K)$. We now have the labeled data (\mathbf{X}, \mathbf{y}) , from which we can easily extract the required internal datasets via sampling as already discussed in [Section 5.4.3](#). We accomplish this by randomly taking r time series from each metric k (if r is equal across all $k \in K$, we obtain a balanced dataset; cf. [Figure 5.9](#) which exactly shows this procedure for two classes/metrics A and B , i.e., $K = \{A, B\}$), or alternatively, we can also use a sampling fraction $s \in (0, 1]$ to retain the original metric distribution (thereby obtaining an equally unbalanced dataset if the original data was unbalanced). To sufficiently represent the original data, we repeat this sampling step n times, which yields n internal datasets \mathcal{I} , more formally, $\mathcal{I} = (\text{sample}_i(\mathbf{X}, \mathbf{y}) \mid i \in [1, n])$ with $\text{sample}(\mathbf{X}, \mathbf{y}) = (\text{rnd}(\mathbf{X}^k, \mathbf{y}^k, r) \mid \forall k \in K)$, where the function rnd represents taking r random samples from $(\mathbf{X}^k, \mathbf{y}^k)$ (cf. [Figure 5.9](#) with $n = \text{Number of internal datasets} = 2$ and $r = \text{Sample size per class} = 5$). [Figure 5.19](#) illustrates this procedure of creating such internal, labeled datasets. The idea behind this approach is the hypothesis that the clustering method candidates should at least be able to separate the different metrics again.¹⁹ If they fail to do so, we expect a bad performance when trying to cluster the unlabeled data as well.

For our IMTS data, we set the sampling quantity r to 100 and the number of internal datasets n to 30.²⁰ This resulted in 30 internal IMTS₁ datasets, where each dataset contains $m \cdot r = 5 \cdot 100 = 500$ time series, since IMTS₁ consists of five metrics and we sample each metric 100 times. Analogously, we also created 30 internal IMTS₂ datasets, each of which has $m \cdot r = 6 \cdot 100 = 600$ samples, since IMTS₂ consists of six metrics. To make even more use of the UCR data, we can apply the same procedure there as well. If we consider the 128 individual UCR datasets as our time series sources and drop all internal labels (each UCR dataset thus becoming comparable to our unlabeled metrics), we can again create a merged, labeled dataset, where we now assign the dataset *names* as class labels, analogous to using the metrics as class labels as we did when processing the IMTS data. For creating the internal datasets, we used a sampling fraction of $s = 0.05 = 5\%$ and set the number of internal datasets n again to 30. Our new, merged UCR dataset thus has 30 internal datasets (an example of how such an internal dataset could look like is shown in [Figure 5.20](#)), each of which has $\sum_{u \in U} s \cdot |\mathbf{X}^u| = \sum_{u \in U} 0.05 \cdot |\mathbf{X}^u| = 9555$ samples (i.e., 5% of each of the 128 UCR datasets), where U is the set set of all 128 UCR datasets and $|\mathbf{X}^u|$ represents the number of time series in \mathbf{X}^u .²¹ [Table 5.8](#) summarizes all four datasets for the evaluation, and [Figure 5.20](#) shows examples of how the first of the n internal datasets could look like for each of these four datasets.

¹⁹This is precisely the reason why we added the requirement of sufficiently different time series sources, since otherwise, telling them apart after having merged the data becomes significantly more difficult, thereby possibly limiting the validity of the labeled clustering results (diff-matrices).

²⁰Different parameter settings might be necessary to sufficiently represent other data.

²¹This is the result after joining both the train $\mathbf{X}_{\text{train}}^u$ and test samples $\mathbf{X}_{\text{test}}^u$ into a combined \mathbf{X}^u , and then calculating the sample fraction based on this combined data for each UCR dataset $u \in U$. The exact number of train and test samples can be found in the appendix in [Table D.1 on p. 255](#).

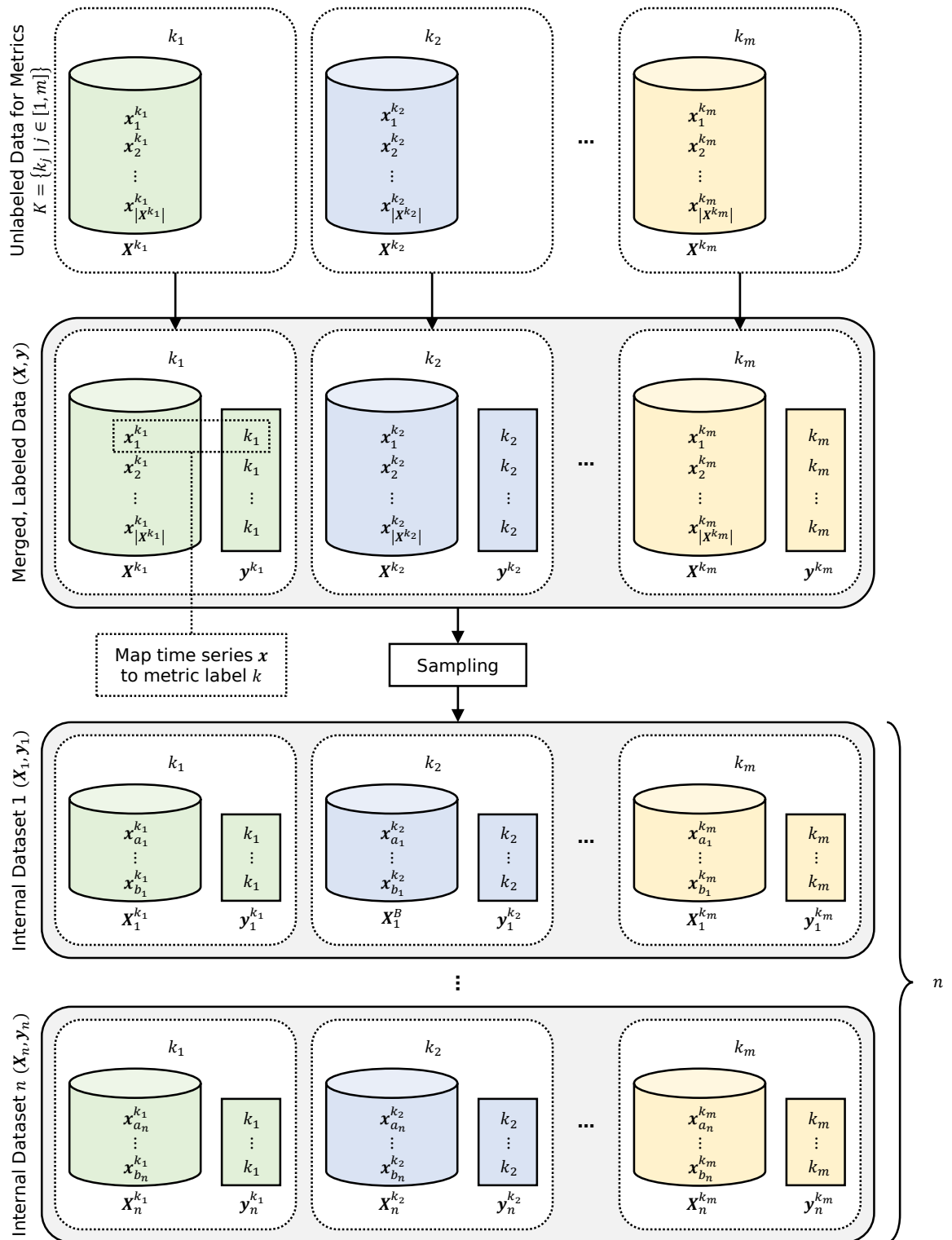


Figure 5.19: Illustration of creating merged, labeled data (\mathbf{X}, \mathbf{y}) from a set of unlabeled time series metrics K , followed by extracting n internal, labeled datasets $(\mathbf{X}_i, \mathbf{y}_i)$ via repeatedly taking random samples. \mathbf{X}^k represents all time series of metric k (and \mathbf{X}_i^k a random subset thereof), $|\mathbf{X}^k|$ its number of time series and \mathbf{x}_l^k its l -th (single/individual) time series. For the internal datasets $(\mathbf{X}_i, \mathbf{y}_i)$, a_i and b_i indicate random indices. Depending on whether an absolute number of samples r or a relative sampling fraction s was chosen in the sampling step, there are a total of r or $s \cdot |\mathbf{X}_i^k|$ such indices per metric k .

Dataset	n	#Samples	#Classes
UCR	128	varying	varying
UCR-merged	30	9555	128
IMTS ₁	30	500	5
IMTS ₂	30	600	6

Table 5.8: UCR and IMTS datasets for the evaluation, where n is the number of the internal datasets \mathcal{I} , and #Samples and #Classes represent the number of samples (time series) and classes in each such internal dataset, respectively. The varying number of samples and classes of the 128 internal UCR datasets can be looked up in the appendix in [Table D.1 on p. 255](#).

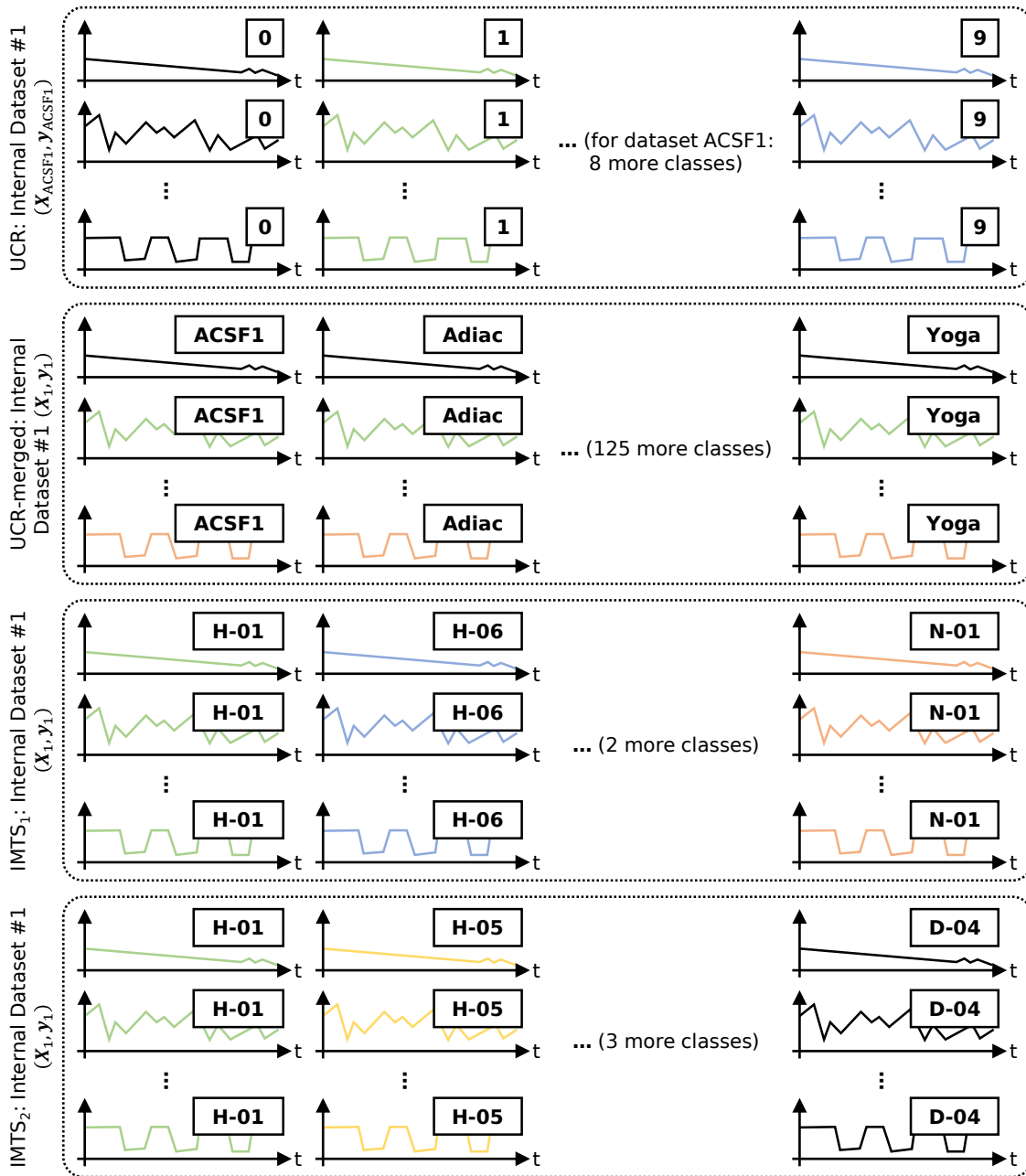


Figure 5.20: Examples of how the first of the n internal datasets could look like for each of the four datasets listed in [Table 5.8](#). The time series are just for demonstrating purposes and do not look like this in the actual dataset.

5.6 Evaluation

Before evaluating our clustering method selection approach on the above datasets, we first present results regarding our time series characteristics to show their usefulness and ability to extract meaningful properties, i.e., that they can indeed be used for clustering and also classification tasks.

5.6.1 Time Series Characteristics

We conducted three experiments to show that our time series characteristics (TSC) are not just a random and arbitrary selection of time series features. Specifically, we tested their capability to classify time series, i.e., to train a classical supervised machine learning model on a training set and evaluate it on the corresponding test set, and to cluster time series using an unsupervised machine learning model. In both cases, we used the raw time series data as well as the feature set *catch22* [110] (containing 22 selected features) to compare to our TSC. *catch22* was already shown to perform well on the UCR datasets, so we decided to evaluate the TSC on the same data. Lastly, we present how our TSC groups can be useful to inspect the data of the classes/clusters. The results of these experiments are summarized in the following:

- **Classification:** The UCR datasets are already split into training and test sets, so we can directly train a supervised machine learning model, where we opted for the default scikit-learn implementation [131] of the random forest classifier with 100 trees and no depth limit. As input, we used the raw time series data, the features calculated with *catch22* and the features calculated with our TSC, and we did not perform any post-processing (i.e., no variants), which resulted in the three methods *rf|raw*, *rf|catch22* and *rf|tsc* (*rf* is short for random forest). Figure 5.21 shows their accuracy (ACC) scores for the 128 UCR datasets, where the method on the y-axis is compared to the method on the x-axis. The diagonal line from bottom left to top right serves as a visual guide to check whether the y-axis method performed better (data point is above the diagonal) or worse than the x-axis method (data point is below the diagonal). If one method is statistically significantly better across all 128 datasets (Wilcoxon signed-rank test, significance level $\alpha = 0.01$), the corresponding half of the diagonal is highlighted with a yellow background.

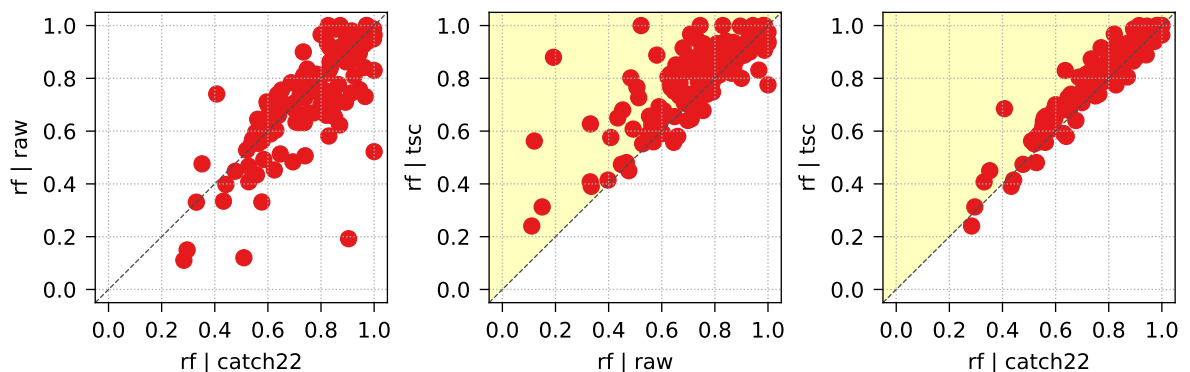


Figure 5.21: Accuracy (ACC) score comparisons when classifying the 128 UCR datasets using three different methods (random forest with raw time series data, *catch22* features and TSC features). The yellow highlighted background in the upper left part indicates that the method on the y-axis performed statistically significantly better than the method on the x-axis.

The results clearly show that our selected TSC set work well when classifying the UCR datasets, outperforming both the methods that are based on the raw time series data as well as the catch22 features. However, it must be noted that the primary goal of catch22 was not to get the best classification results, but rather to find a small set of merely 22 features that still provide good performance while reducing the required computation time drastically [110].

- Clustering: This task is arguably more difficult than classification since we now no longer know the true classes of the datasets, and there are no separate training and test sets any more. As mentioned in Section 5.5.3, we can thus combine the UCR train and test data, allowing us to leverage a larger amount of data. Using this combined data, we evaluated the clustering performance with the default scikit-learn implementation [131] of the k-means algorithm, where we set its number of classes/clusters k to the number of classes of the corresponding UCR datasets (cf. column $\#C$ in Table D.1 on p. 255). As input, we again used the raw time series data, the features calculated with catch22 and the features calculated with our TSC, and we did not perform any post-processing (i.e., no variants), which resulted in the three methods `kmeans|raw`, `kmeans|catch22` and `kmeans|tsc`. Figure 5.22 shows their adjusted Rand index (ARI) scores for the 128 UCR datasets, where the plot information is exactly the same as in the above classification case.

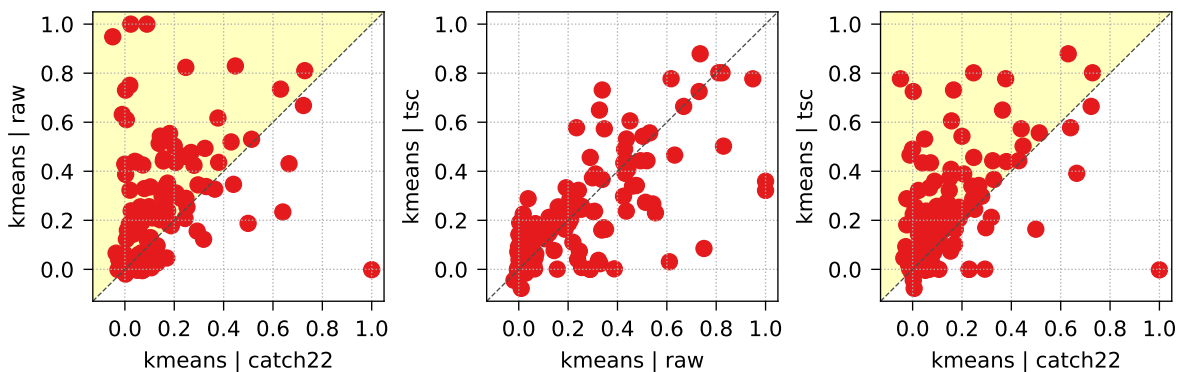


Figure 5.22: ARI comparisons when clustering the 128 UCR datasets using three different methods (k-means with raw time series data, catch22 features and TSC features). The yellow highlighted background in the upper left part indicates that the method on the y-axis performed statistically significantly better than the method on the x-axis.

Again, our TSC work well when clustering the UCR datasets, where we outperform catch22 once more and obtain equally good results compared to clustering based on the raw time series data. In contrast to the raw data, however, the TSC provide the benefit that we can inspect various properties of the different time series classes/clusters, which we present in the last experiment.

- Class/Cluster inspection: We designed our TSC in a way that users can analyze certain time series properties by inspecting the feature values of the different groups and subgroups. If we have classes for a given dataset (either the true classes or the ones we obtained via our clustering methods), we can visualize their (sub)group feature values and easily compare them to each other. Since the TSC features are all robustly scaled (90% of the feature values are within the interval $[0, 1]$), we can also see which (sub)groups are more pronounced. In Figure 5.23, the feature values of all TSC subgroups (including the *test* group because this group does not have any subgroups) for the two-class UCR dataset *BirdChicken* are shown (the TSC group names on the right of the plot are

abbreviations of the form $g_subgroup_b$, where g is the first letter of the main group and b represents the blockwise subgroups). In this dataset, for instance, we can observe that the *frequency*, *flatness* and *test* groups yield rather low values for all time series, whereas the *similarity* and *miscellaneous complexity* groups result in near maximum values. The class differences are mainly between the *distributional dispersion*, *blockwise temporal dispersion* and *entropy* groups.

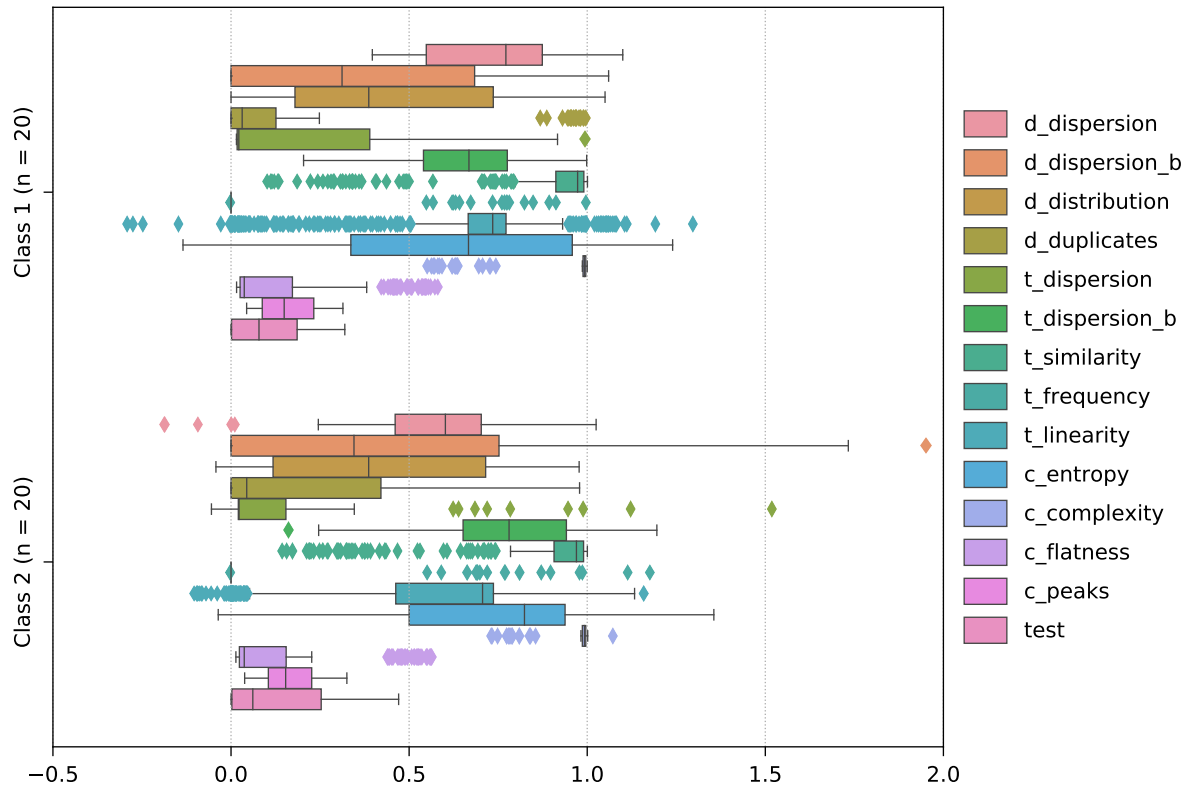


Figure 5.23: The feature values of the TSC (sub)groups for the two classes 1 and 2 of the UCR dataset *BirdChicken*. The respective class sizes are denoted by n . Abbreviations: d = distributional, t = temporal, c = complexity, b = blockwise.

Using such a visualization, we can gain more insights into the feature value distribution across the different classes/clusters and can also easily compare both the classes as well as the individual TSC groups themselves to each other.

5.6.2 Clustering Method Selection

The evaluation of our clustering method selection with the two UCR and two IMTS datasets follows the same steps presented in the approach, i.e., we first determine the importance of the feature sets we selected, continue with various post-processing options, create our set of candidate methods and then cluster the labeled data to get the ranked list of those methods that performed best. Finally, we use one of these methods to cluster our unlabeled multi-system infrastructure monitoring data. Regarding the selected feature sets \mathbf{F} , we decided to use the TSC subgroups, the main groups and all groups combined (the set of all TSC features), i.e., a total of $13 + 4 + 1 = 18$ TSC feature sets. To compare our TSC to an established feature set from related work, we also integrated catch22 [110] (as already introduced in the previous section) into our clustering method selection approach. In the last part of the evaluation, we also show the run-time cost model when applied to the UCR datasets.

5.6.2.1 Determining Feature Set Importance

The first step is the feature set importance evaluation of our TSC (sub)groups²² and the catch22 feature set. To this end, we trained a random forest with 100 trees and no depth limit (default scikit-learn implementation [131]), and we repeated this training $n = 10$ times. There are 167 TSC features (distributed among four main groups and 13 subgroups) and 22 features from catch22, which means that there are a total of 189 features and, in turn, 189 possible ranks. Figure 5.24 shows the feature-importance-ranks box plot, where we can infer the importance of our TSC groups (the abbreviations are the same as discussed above in Section 5.6.1) and the catch22 feature set for the respective datasets.

We can see that none of the groups truly outshines another. Some appear to be slightly more/less important but often also contain the worst/best ranks, and there are differences among the datasets as well, so we might not want to drop any of them here. However, we might also consider to only take the two IMTS datasets into account if we want our approach to focus more on the IMTS data rather than the IMTS *and* the UCR data, in which case the *test* group could be dropped due to its lowest importance. Since dropping feature sets is mainly beneficial with respect to computational costs (fewer feature sets lead to fewer method candidates) and our main goal is finding clusters in our multi-system data with the computational costs being less relevant, we decided not to drop any feature sets at this point and to use all of them in the next steps, i.e., $\mathbf{F}' = \mathbf{F}$.

5.6.2.2 Post-Processing Feature Sets

As mentioned in Section 5.3, we robustly scale some TSC features in the normalization process, which can still occasionally lead to potentially large outliers outside the range of $[0, 1]$. Since many clustering algorithms rely on the distance between the different feature values and are thus affected by scale (and, in turn, affected by outliers), we post-process the values of each individual feature with the following three variants, whose effects are also visualized in Figure 5.25:

- *clip01*: Clips the values to $[0, 1]$.
- *clipTan*: Scales each value v with $\frac{\tanh(2 \cdot v - 1)}{2 \cdot (\tanh(1) + 1)}$. This is a non-linear transformation which continuously reduces the spacing between values the larger those values become. Outlier values with a magnitude of 2 or more (i.e., < -2 or > 3) will effectively be trimmed to the interval $[-0.1565, 1.1565]$.
- *clipLog*: Scales each value v above 1 with $1 + \log_{10}(v)$ and each value v below 0 with $-\log_{10}(|v| + 1)$. The logarithm is not bound, so the scaled values are additionally clipped to $[-3, 4]$, which occurs for $|v| \geq 10000$, i.e., extreme outliers. In contrast to clipTan, values between $[0, 1]$ stay exactly the same, i.e., the spacing is non-linearly reduced only for values outside this range.

The features from catch22 are not normalized, so we scale the values of each individual feature as follows:

- *minmax*: Scales the values to $[0, 1]$ (normalization).
- *robust*: Robustly scales the values using the 5% percentile as lower bound and the 95% percentile as upper bound.

²²With the exception of the group *test* which does not have any subgroups, the subgroups are sufficient to get an overview, since the main groups and the entire TSC set are just aggregations thereof.

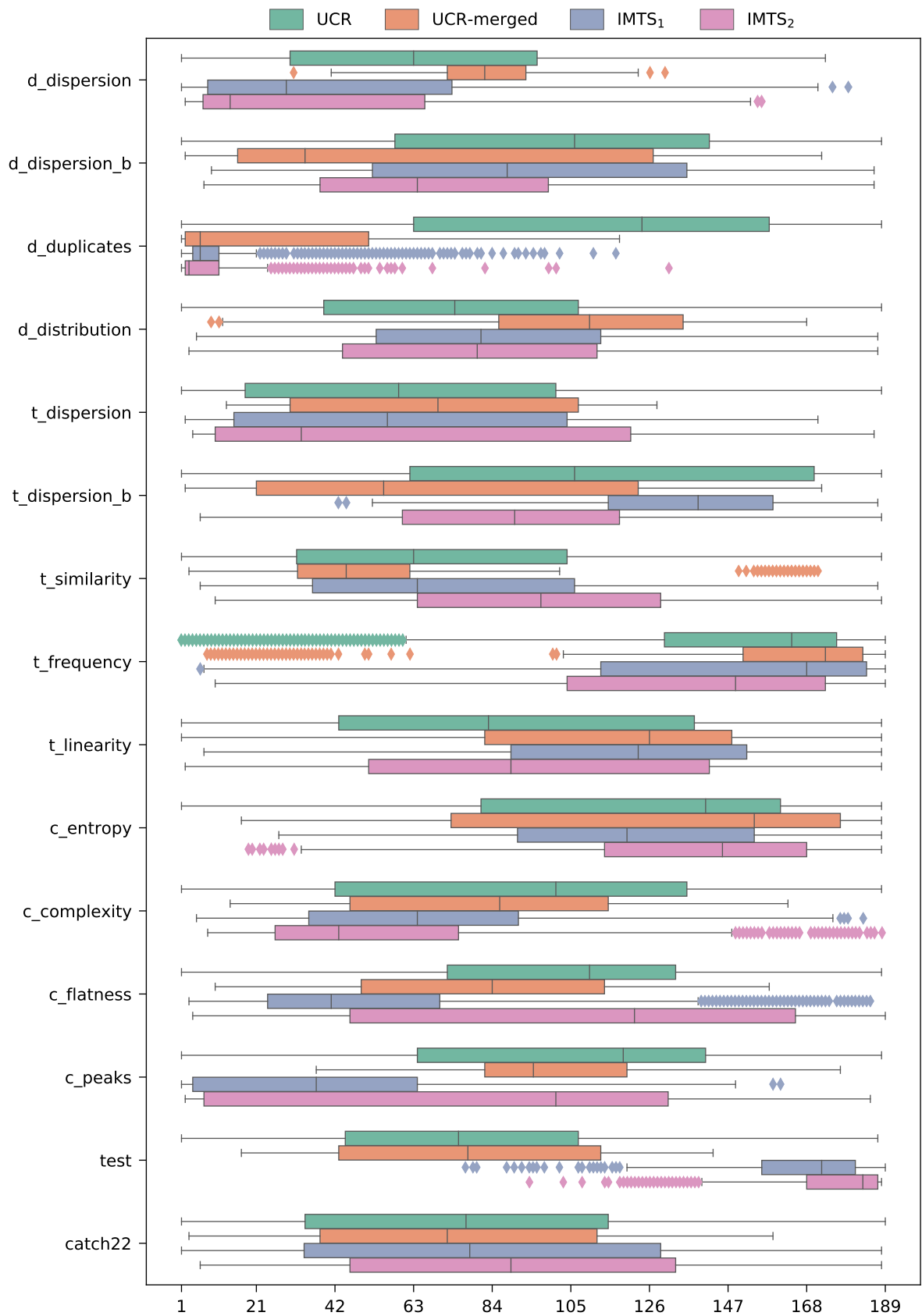


Figure 5.24: Feature-importance-ranks box plot of the TSC (sub)groups and catch22 feature set with a total of 189 features and thus 189 ranks, grouped by the four datasets. Abbreviations: *d* = distributional, *t* = temporal, *c* = complexity, *b* = blockwise.

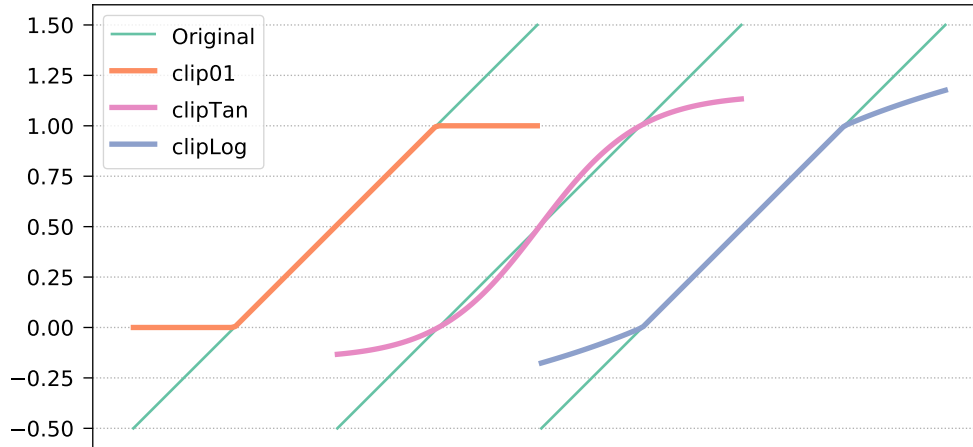


Figure 5.25: Example of the effect of the three TSC clipping variants clip01, clipTan and clipLog on the *Original* values in the range $[-0.5, 1.5]$.

Finally, the post-processing function *drop* omits features with only one unique value and features with an absolute Pearson correlation coefficient ≥ 0.95 .

For the TSC groups, we now create eight variants V_{tsc} : no post-processing, dropping, all three TSC clipping options and all three clipping options with additional dropping afterwards. For the catch22 feature set, we create six variants V_{catch22} : no post-processing, dropping, both catch22 scaling options and both scaling options with additional dropping afterwards. We have 18 TSC feature sets and the catch22 feature set, which results in a total of $18 \cdot 8 + 1 \cdot 6 = 150$ feature-set-variant combinations as input for the next step.

5.6.2.3 Clustering Labeled Data

This is the main step of the clustering method selection approach, where we obtain the ranked list of all candidate methods we want to evaluate. With the completed feature sets F and variants V from the previous steps, the last part of the method triplets are the unsupervised clustering models M , where we decided to use the following five models due to their ability to scale well with large datasets and because they have already been successfully applied in numerous domains:

- k-means: Default scikit-learn implementation [131] with a fixed number of clusters and a fixed random state for reproducibility.
- BIRCH: Default scikit-learn implementation [131] with a fixed number of clusters.
- Linkage: Agglomerative hierarchical clustering using the default SciPy implementation *linkage* [185] with a fixed number of clusters, Euclidean distance metric and Ward’s linkage criteria for the distance calculation between clusters.
- Linkage weighted: Same as the linkage model above but with the weighted average linkage criteria instead of Ward’s criteria.
- Linkage weighted cosine: Same as the linkage weighted model above but with the Cosine distance metric instead of the Euclidean distance.

We set the number of clusters to the number of classes of the corresponding datasets, which are listed in [Table 5.8](#) under *#Classes*. Now we can complete our candidate methods. First, we

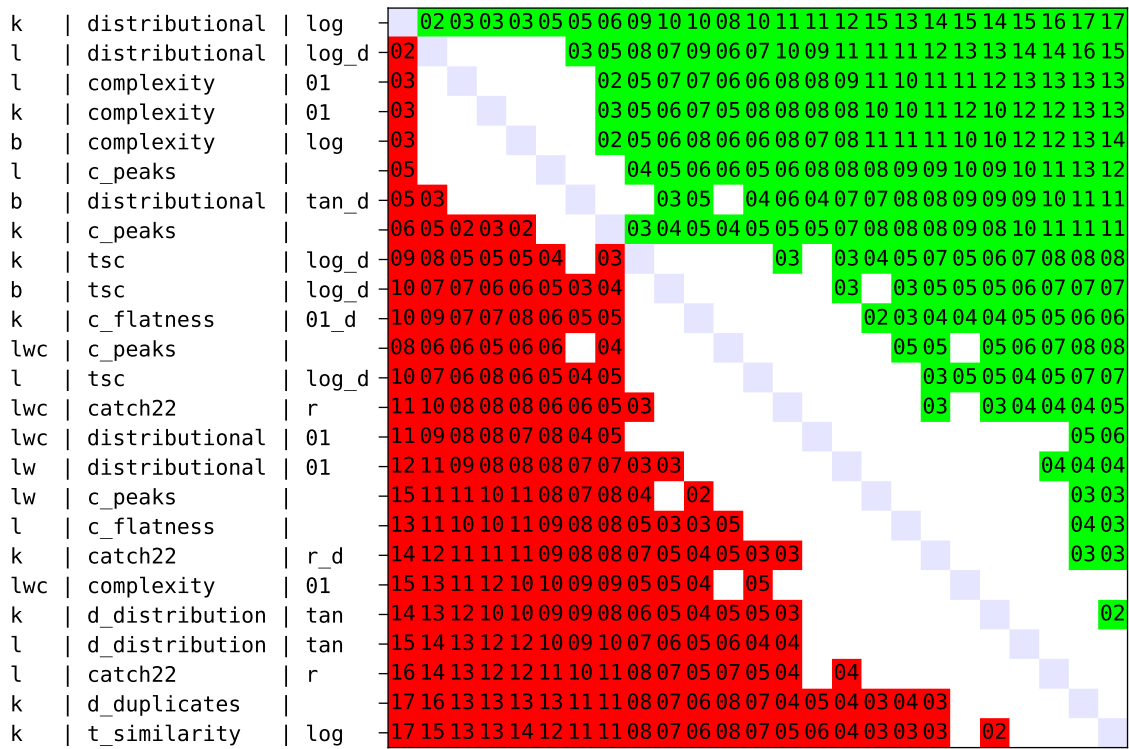
create all TSC-based methods with $\mathbf{M} \times \mathbf{F}_{\text{tsc}} \times \mathbf{V}_{\text{tsc}}$, which results in $5 \cdot 18 \cdot 8 = 720$ methods. Second, we create all catch22-based methods with $\mathbf{M} \times \mathbf{F}_{\text{catch22}} \times \mathbf{V}_{\text{catch22}}$, which results in $5 \cdot 1 \cdot 6 = 30$ methods. Third, we also create raw-based methods to see how well clustering based on the raw time series data fares against the feature-based methods. To this end, we added five additional methods (each of the five models with the raw data as input). In total, the evaluation of the four labeled datasets thus comprises a total of $720 + 30 + 5 = 755$ candidate methods, and we set the significance level α to 0.01 when calculating the diff-matrices.

Due to this large number of methods, we cannot visualize the entire 755×755 -sized diff-matrices. Therefore, we first filter the matrices to only keep the best variant of each method, where “best” is defined by the row-based score introduced in Equation 5.2. This reduces the size from 755 to 95, of which we then display the top 25 methods. Figure 5.26 shows the diff-matrices for our four datasets, each containing the best 25 methods in descending order, i.e., the best-performing method is the first row. To get more compact diff-matrix representations, the column method names are omitted (they are identical to the corresponding row method names when mirrored along the main diagonal of the matrices).

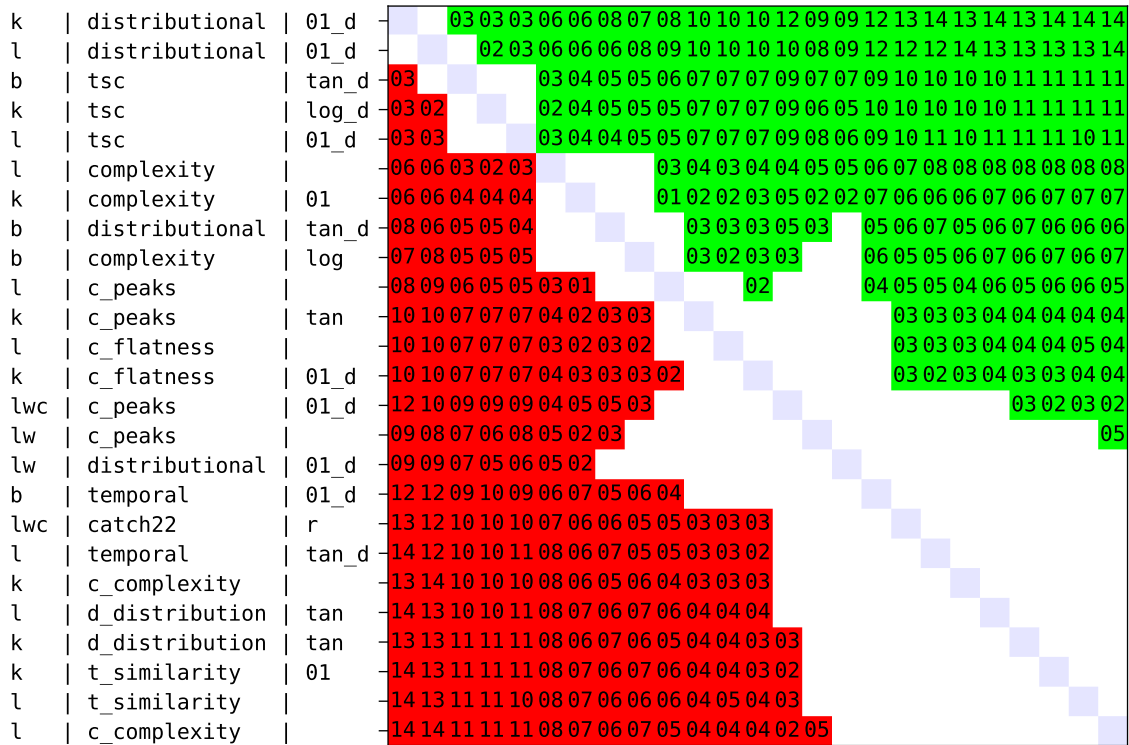
The results reveal several interesting aspects. For example, all five clustering models appear among the top 25 methods in every dataset, and the default linkage model generally performed better than its weighted and weighted cosine counterparts. The catch22 feature set managed to get into the top 25 methods, but it was outperformed by various TSC (sub)groups, most notably by the entire TSC feature set (*tsc*), which generally performed best (top-ranked feature set in UCR and UCR-merged, sixth best in IMTS₁ and second best in IMTS₁). However, we can also see that some TSC groups yielded comparable or even better results than the entire TSC feature set despite (significantly) fewer features. For instance, the *distributional* group (34 features) was the top-ranked feature set in both IMTS datasets and thus beat the full *tsc* set (167 features), which can be a determining factor if computational costs are relevant. Raw-based clustering evidently did not work when applied on our IMTS datasets,²³ in contrast to the UCR datasets, where the methods utilizing raw time series managed to perform nearly as well as the TSC-based methods. Some readers might conclude that the variants of all methods seem arbitrary and mixed. This is because the differences in the variants are actually rather small in most cases, which is not visible in these reduced matrices. As an example of such a case, Figure 5.27 shows the variant differences of the three selected models k-means, BIRCH and (standard) linkage within the *distributional* feature set for all four datasets (again, the column variants are omitted to get more compact matrix representations). In all datasets but UCR-merged, we can clearly see that the no-post-processing variant and the drop variant generally performed worse than the other variants. However, among these other variants, the differences are often either rather small (absolute median ARI-difference ≤ 0.05 in the majority of cases) or entirely insignificant. With a few exceptions, this observation can be made throughout all different datasets, models and feature sets, where the full list of all possible variant differences can be found in the appendix in Section D.3.1 on p. 255.

Finally, we must choose a method that we will use to cluster the unlabeled data. Just like in the feature set importance part, we must decide which of the diff-matrices we take into consideration for choosing this method, although the choice here has a much greater impact, since it now is not only related to the computational costs but also to the actual clustering performance. The type of both the evaluated datasets and the future data, which we intend to cluster, heavily influences this selection decision. We have two general datasets (UCR and UCR-merged) and two domain-specific datasets (IMTS₁ and IMTS₂). If we seek a more general solution, we might consider all four datasets, for example, if we expect the future data and its

²³In the full, unfiltered diff-matrices, the highest-ranked raw-based method only lies at position 459/755 for IMTS₁ and 360/755 for IMTS₂.

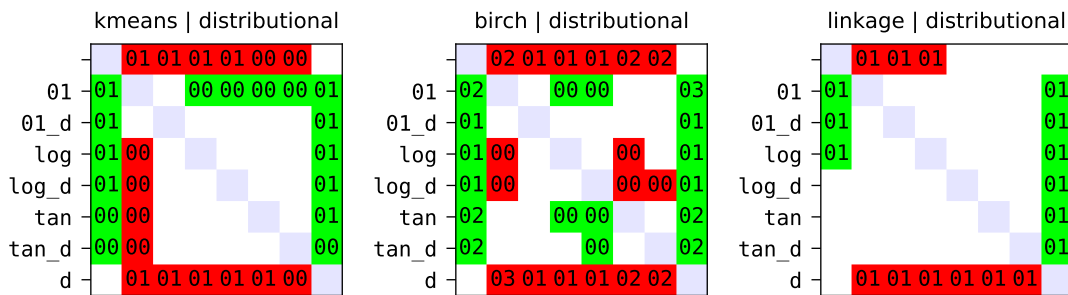


(c) Diff-matrix for dataset IMTS₁.

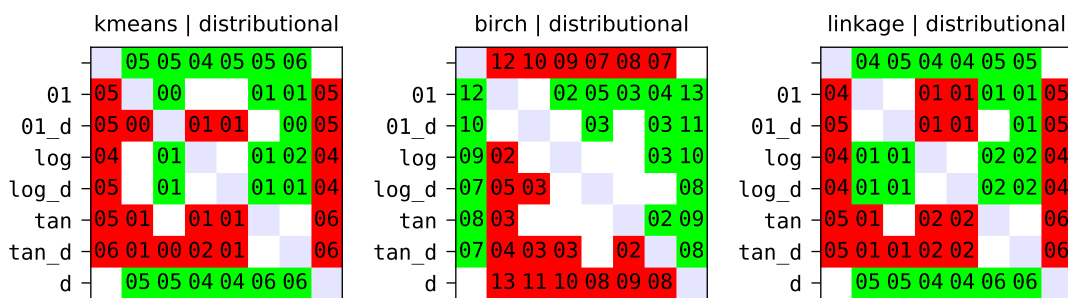


(d) Diff-matrix for dataset IMTS₂.

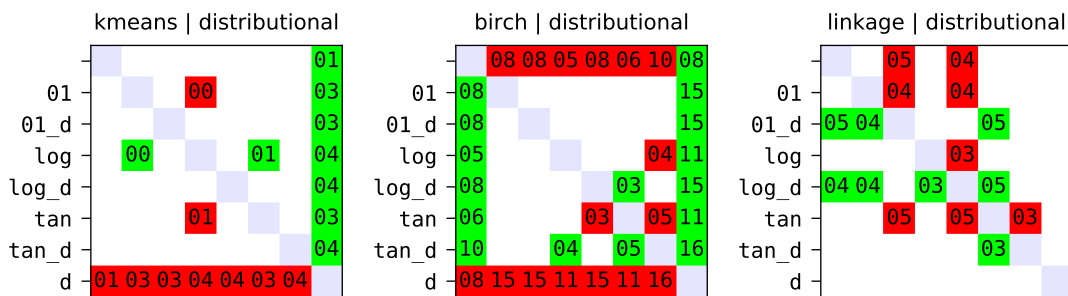
Figure 5.26: Diff-matrices for all datasets, sorted by the 25 best methods. Abbreviations: models: k = k-means, b = BIRCH, $l(w)(c)$ = linkage (weighted) (cosine); feature sets (cf. [Section 5.6.1](#)): d = distributional, t = temporal, c = complexity, b = blockwise; variants: empty = no post-processing, 01 = clip01, tan = clipTan, log = clipLog, m = minmax, r = robust, d = drop, v_d = variant with drop. Omitted column methods = row methods.



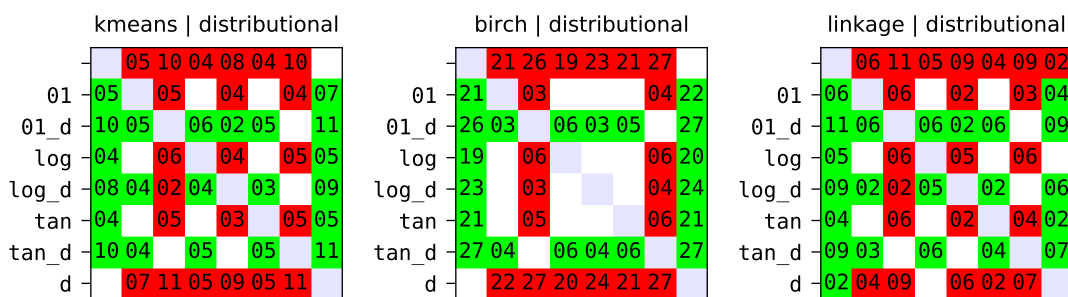
(a) Variant differences for dataset UCR and feature set *distributional*.



(b) Variant differences for dataset UCR-merged and feature set *distributional*.



(c) Variant differences for dataset IMTS₁ and feature set *distributional*.



(d) Variant differences for dataset IMTS₂ and feature set *distributional*.

Figure 5.27: Variant differences for all datasets of the three models k-means (left), BIRCH (middle) and linkage (right) in combination with the *distributional* feature set. Abbreviations: empty = no post-processing, *01* = clip01, *tan* = clipTan, *log* = clipLog, *d* = drop, *v_d* = variant with drop. Omitted column variants = row variants.

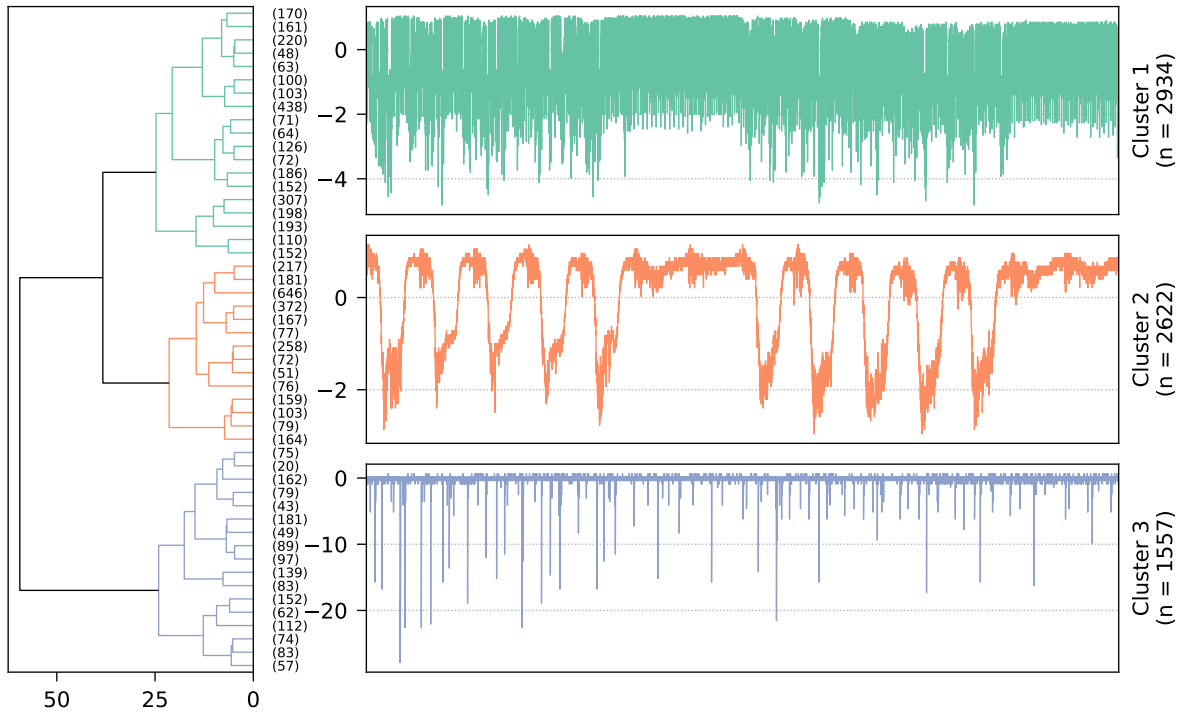
domain to be different or more general as well. In our case, however, the goal is to find clusters within the same infrastructure monitoring data that we used to create the two IMTS datasets, so we decided to choose a method that particularly performed well on IMTS₁ and IMTS₂, i.e., one of the top-ranked methods in [Figure 5.26c](#) and [Figure 5.26d](#), respectively. As already discussed above, the *distributional* feature set achieved the highest scores, and regarding the clustering model, we opted for the linkage model because it performed nearly equally well as the k-means model (the differences are negligible), but it comes with the advantage that we can use a dendrogram for visualizing the cluster hierarchy, which is immensely helpful when determining the number of clusters. For choosing the variant, we can look at the variant differences of the linkage model in combination with the *distributional* feature set, which we already presented in [Figure 5.27c](#) for IMTS₁ and [Figure 5.27d](#) for IMTS₂. Clearly, the no-post-processing variant and the drop variant performed worse, followed by the three clipping variants, and the best results were achieved with the three clipping options when additionally combined with the drop variant. Among the latter three, the differences are minimal, so we selected the variant *01_d* (clipping to $[0, 1]$ and dropping correlated features afterwards) due to its simplicity compared to its logarithm-based and tangent-based counterparts. Our final method for clustering the unlabeled data is thus `linkage|distributional|clip01_drop`.

5.6.2.4 Clustering Unlabeled Data

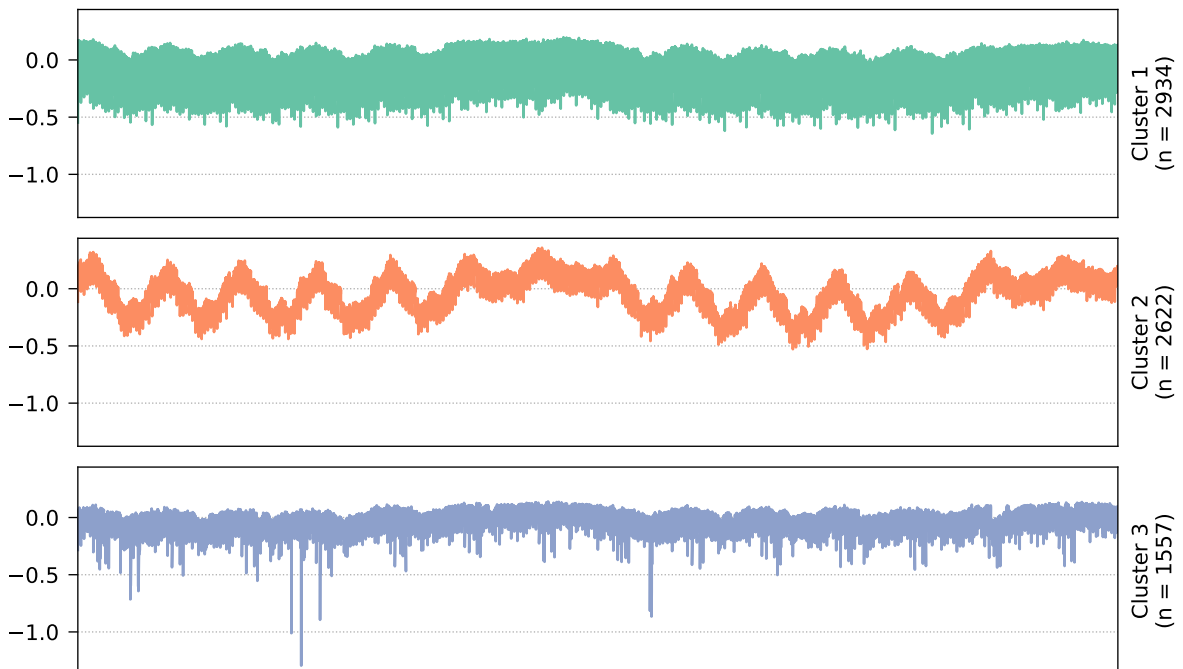
We can now finally cluster the unlabeled data, which is our ultimate goal. For instance, we might be interested in the clusters within the CPU Idle (H-01) metric. We thus applied our selected method on the 7113 time series of the IMTS₁ dataset, which we collected from 606 different systems. In [Figure 5.28a](#), the truncated dendrogram (50 splits) is shown, with which we could identify three main clusters,²⁴ where a representative time series sample is plotted next to the dendrogram for each cluster. Moreover, the cluster sizes n are shown next to these time series. We also provide the cluster averages, i.e., for each cluster, the average μ of all n time series, which are displayed in [Figure 5.28b](#). The results indicate that the three clusters comprise time series that are particularly active (cluster 1), those that have distinct patterns and shapes such as trends and seasonality (cluster 2), and those that appear to have many (potentially significant) spikes (cluster 3). The authors of [\[204\]](#) and [\[142\]](#) manually observed such a threefold classification as well, but in our case, this distinction is within a single metric. The cluster sizes are different but without any major outliers: The largest cluster 1 has 2934 time series (41%), cluster 2 has 2622 time series (37%) and the smallest cluster 3 has 1557 time series (22%). More interesting are the multi-system statistics: In cluster 1, there are time series from 443 systems (73%), cluster 2 contains 422 systems (70%) and cluster 3 contains 352 systems (58%). We can also calculate the cluster-overlap distribution: 194 systems (32%) appear in only a single cluster, whereas the time series of 213 systems (35%) are distributed among two clusters, and the remaining 199 systems span all three clusters. The Venn diagram [\[184\]](#) in [Figure 5.28c](#) shows these statistics in more detail. Overall, these are interesting findings since we successfully identified three main CPU clusters and obtained their system distributions, which we could now use for developing cluster-based tools and models that could then be applied in the multi-system environment.

We present another example, where we clustered the 7176 time series of the Memory Available % (H-07) metric of the IMTS₁ dataset. Again, we could identify three main clusters, albeit of different types. [Figure 5.29](#) shows that cluster 3 still represents spiked data, but clusters 1 and 2 both contain distinctly shaped time series, where those of cluster 2 appear to

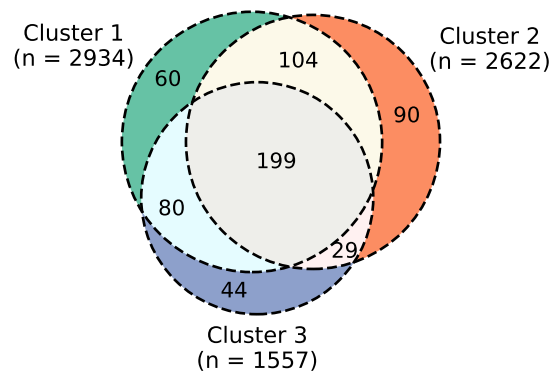
²⁴More (detailed) clusters can be obtained by cutting the dendrogram at a lower distance threshold (x-axis). Our primary focus was to get a general overview, so we deliberately tried to extract (few) main clusters.



(a) Dendrogram (left) and representative time series (right) for the three identified clusters.



(b) Cluster time series averages.



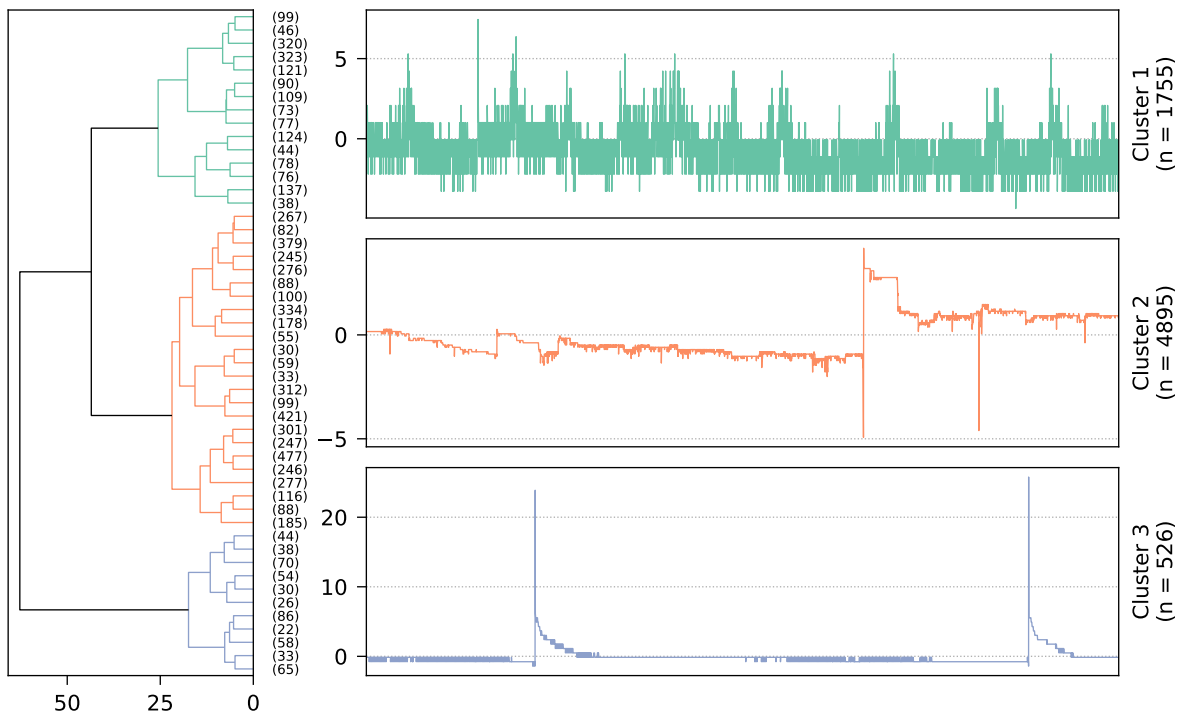
(c) Venn diagram showing the distribution of the 606 systems.

Figure 5.28: Various results obtained when clustering the 7113 CPU Idle (H-01) series of the IMTS₁ dataset into three clusters. The respective cluster sizes are denoted by n , and all time series contain 20160 data points (two weeks in one-minute resolution, ranging from 22.01.2018 00:00 UTC to 04.02.2018 23:59 UTC).

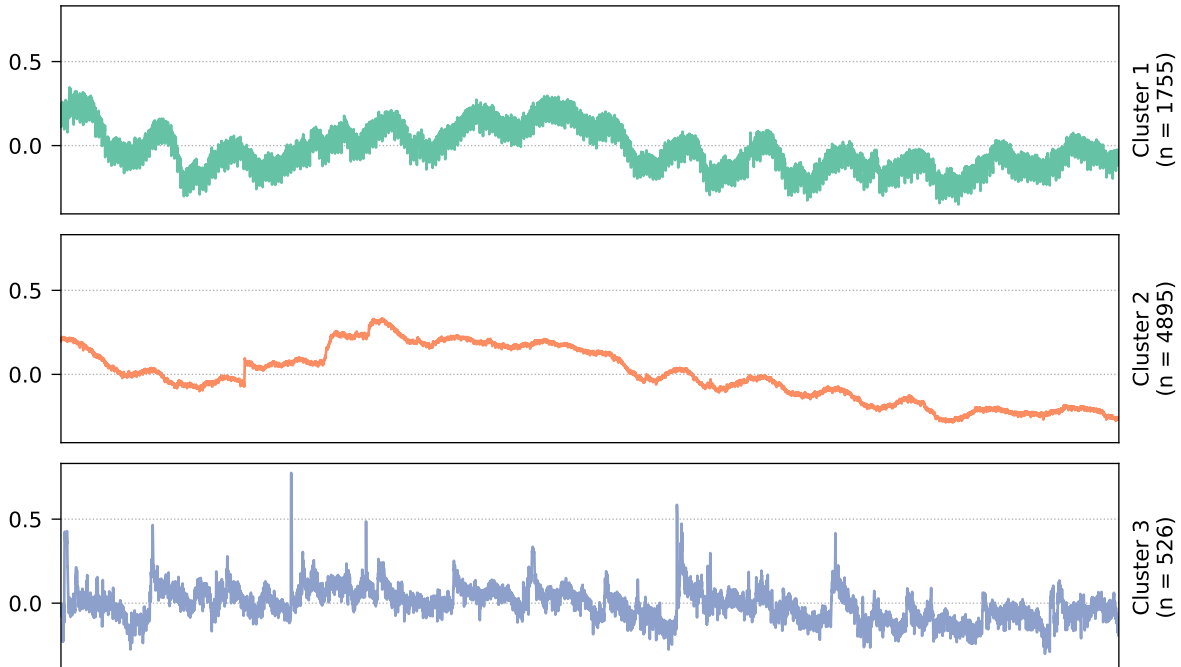
be much smoother. Furthermore, the cluster sizes are now significantly different: Cluster 1 consists of 1755 series (25%), in contrast to clusters 2 and 3 that contain 4895 (68%) and only 526 time series (7%), respectively. We thus decided to split the clusters further by cutting the dendrogram at a lower distance threshold, which ultimately resulted in a new clustering with a total of six clusters.²⁵ In [Figure 5.30](#), the corresponding dendrogram (with representative time series) and the six cluster time series averages are shown. The new cluster 6 represents the same 526 spiked time series as previously cluster 3, but the five new clusters covering the other 6650 series allow more insights into the previously rather similarly looking clusters 1 and 2. For instance, we can see that the new clusters 4 and 5 contain time series with a downward slope, clusters 1 and 3 appear to have more pronounced seasonal or periodic patterns, and cluster 2 can be interpreted as the remainder without any significant characteristics (white noise signal), which is comparable to cluster 6 but without the spikes.

Increasing the number of clusters does not always help, as the following example shows. We first separated the 2259 time series of the Disk Available % (D-03) metric of the IMTS₂ dataset into two clusters, whose time series averages are displayed in [Figure 5.31](#). Clearly, the two cluster do not seem to be much different, so we again checked different numbers of clusters up to six, where the final dendrogram (with representative time series) and the six cluster averages are shown in [Figure 5.32](#). Clusters 2, 4 and 6 are distinct (to a certain degree), but the other three clusters still exhibit hardly any differences. Using other features than the *distributional* ones, for example, all of our TSC features, can yield better results, as already the first three main clusters look significantly different, which is shown in [Figure 5.33](#). Possible reasons behind this is the fact that we did not have perfect historical data available when selecting our method (we created the labeled data ourselves), so deviations and cluster differences are to be expected. Furthermore, just like the *distributional* features, the TSC group was among the top-performing feature sets as well, so its performance here is not too surprising. Lastly, since clustering is highly dependent on the data, it could easily be the case that some distinctive characteristics of this particular unlabeled dataset can only be captured using the entire set of TSC features.

²⁵We consecutively tried lower distance threshold to create four, five and lastly six clusters, where the results of the latter looked most promising.

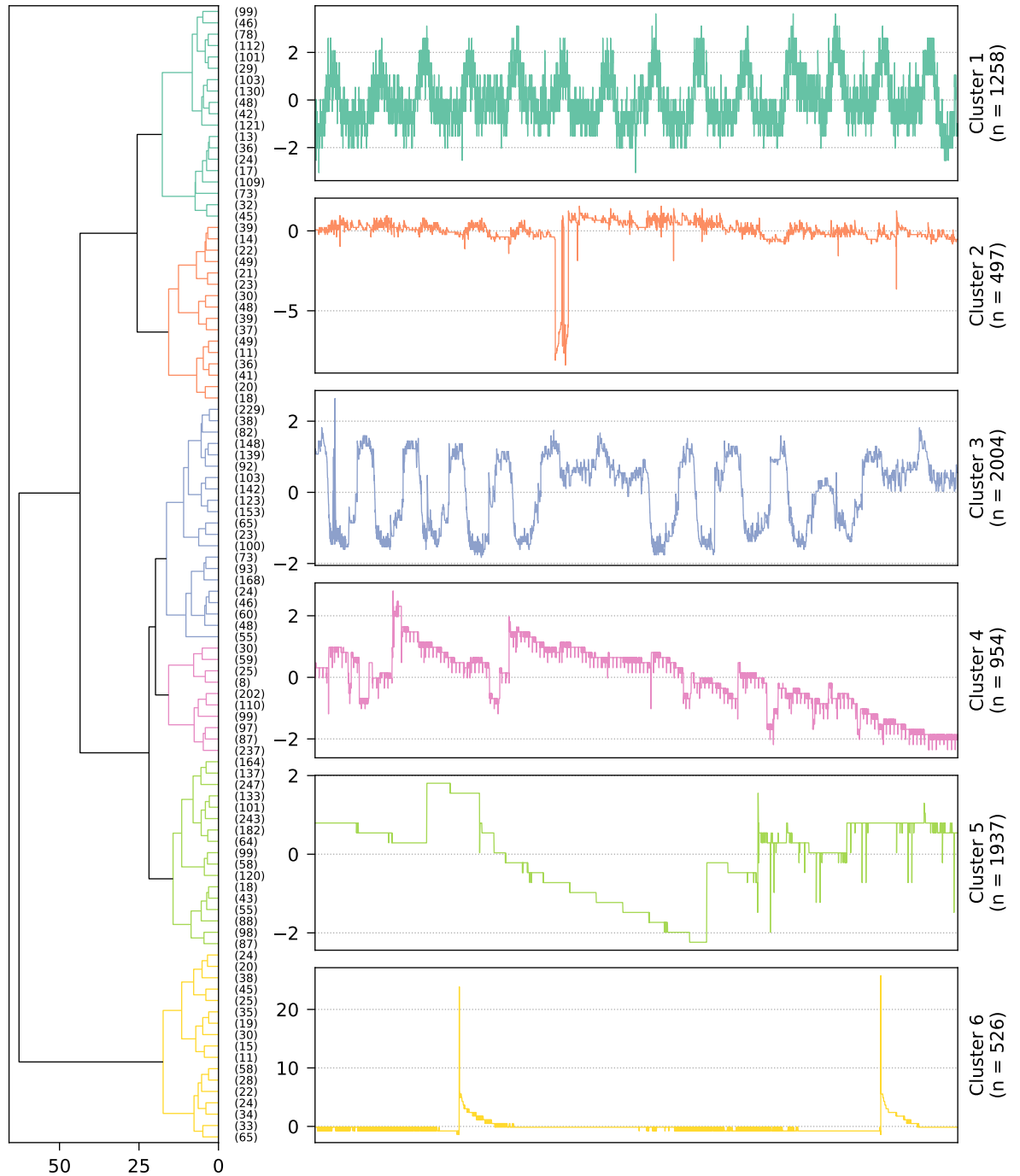


(a) Dendrogram (left) and representative time series (right) for the three identified clusters.

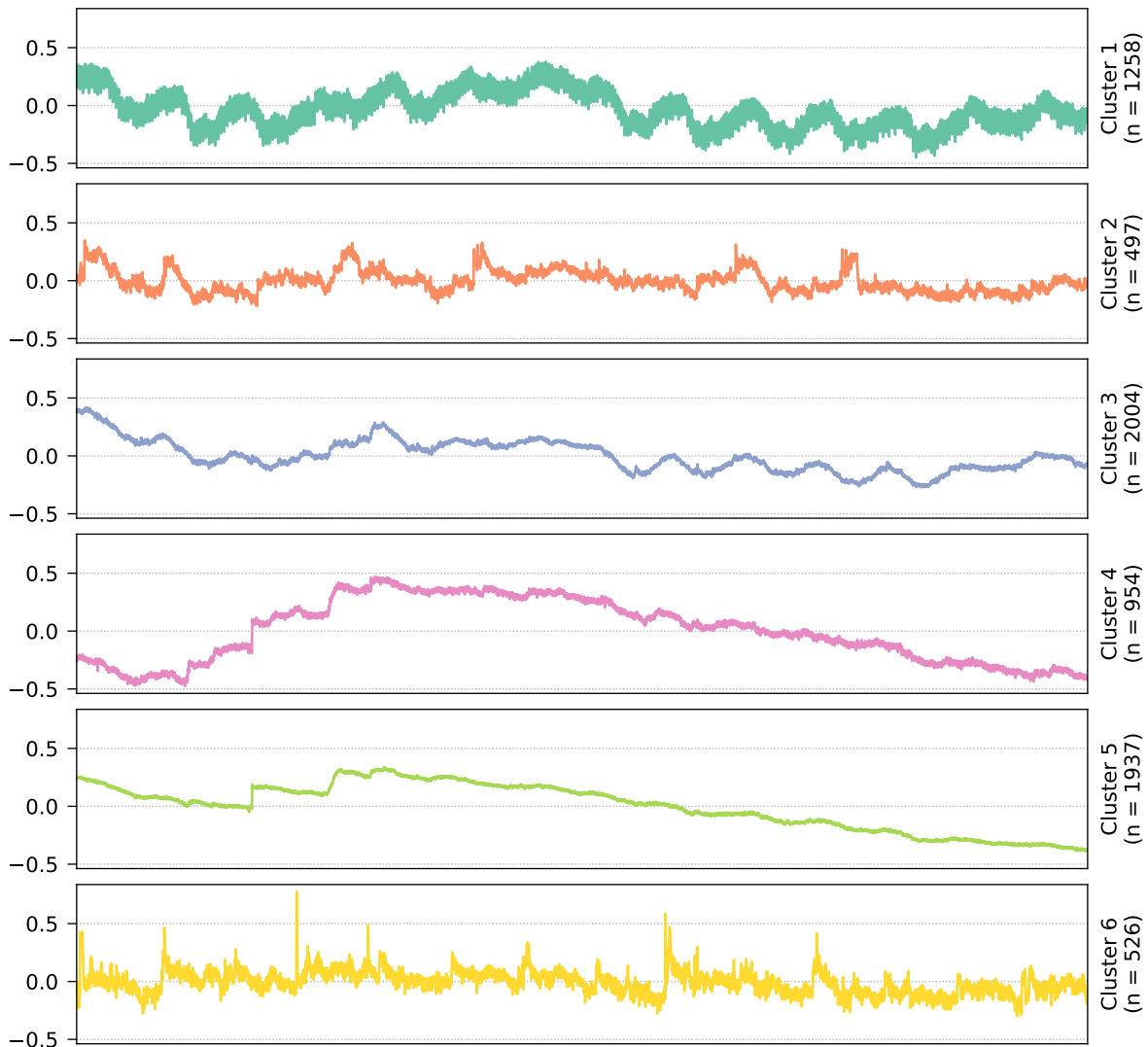


(b) Cluster time series averages.

Figure 5.29: Various results obtained when clustering the 7176 Memory Available % (H-07) series of the IMTS₁ dataset into three clusters. The respective cluster sizes are denoted by n , and all time series contain 20160 data points (two weeks in one-minute resolution, ranging from 22.01.2018 00:00 UTC to 04.02.2018 23:59 UTC).



(a) Dendrogram (left) and representative time series (right) for the six newly identified clusters.



(b) Cluster time series averages.

Figure 5.30: Various results obtained when clustering the 7176 Memory Available % (H-07) series of the IMTS_1 dataset into six new clusters. The respective cluster sizes are denoted by n , and all time series contain 20160 data points (two weeks in one-minute resolution, ranging from 22.01.2018 00:00 UTC to 04.02.2018 23:59 UTC).

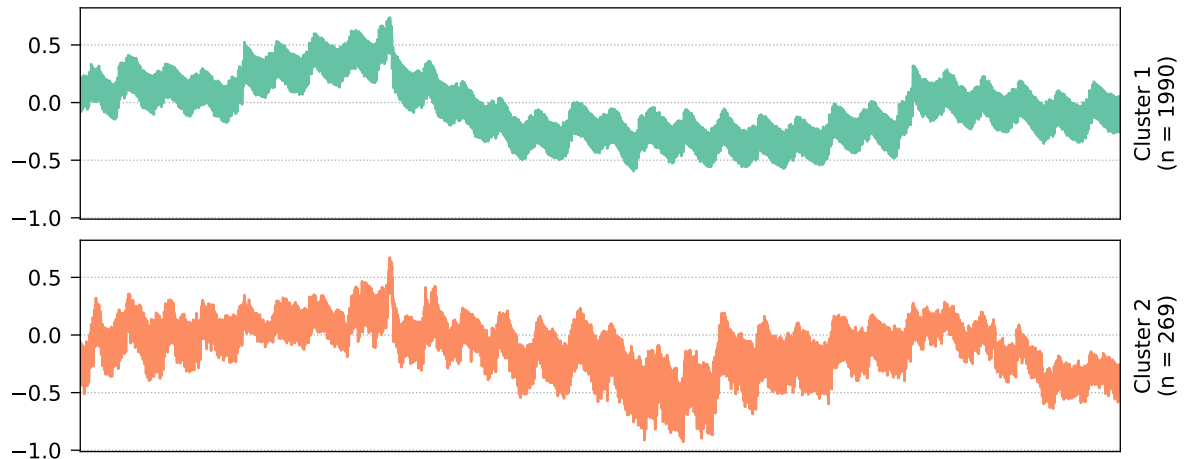


Figure 5.31: Cluster time series averages for the two identified clusters within the 2259 Disk Available % (D-03) series of the IMTS₂ dataset. The respective cluster sizes are denoted by n , and all time series contain 40320 data points (four weeks in one-minute resolution, ranging from 15.07.2019 00:00 UTC to 11.08.2019 23:59 UTC).

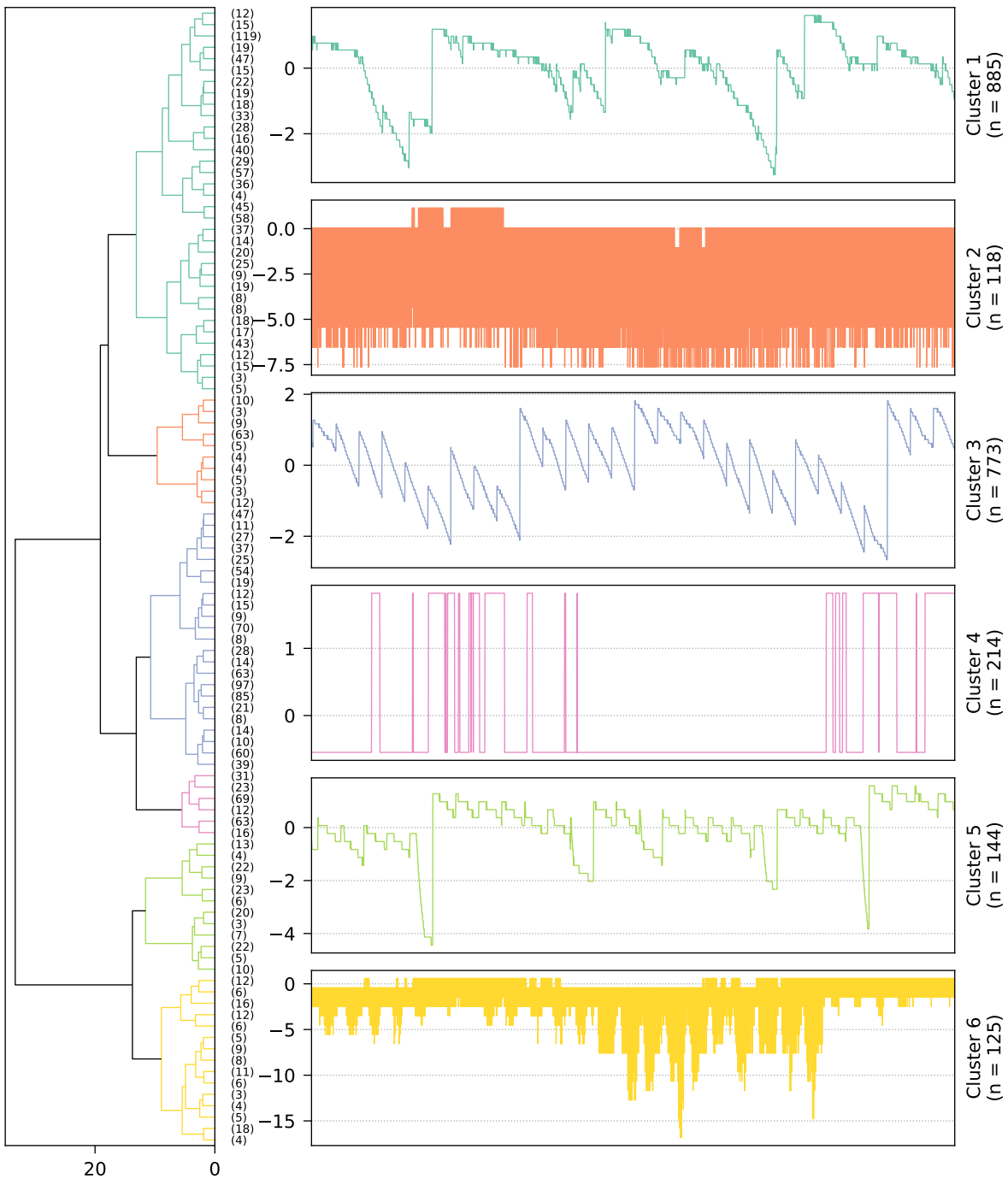
In conclusion, we showed that clustering using one of the top-performing methods from the labeled data evaluation yielded promising results and interesting insights for various unlabeled datasets (the results for all remaining datasets (cf. [Table 5.5](#)) can be found in the appendix in [Section D.3.2 on p. 255](#)). Our approach is based on a full model selection [\[132\]](#) with machine learning models, feature sets and post-processing variants, and is then applied to the problem of clustering, i.e., unsupervised machine learning, which has not yet been a strong focus in research. Furthermore, we presented clustering results of infrastructure monitoring data from a real-world, multi-system environment, where, to the best of our knowledge, we are among the first to present such detailed insights. These results could then be the basis for the development of cluster-specific tools, thereby leveraging our multi-system environment, where we could not only benefit from (sufficient) data of different systems but also from the fact that such tools could potentially be applied to all systems that are part of the corresponding clusters.

5.6.2.5 Run-Time Cost Model

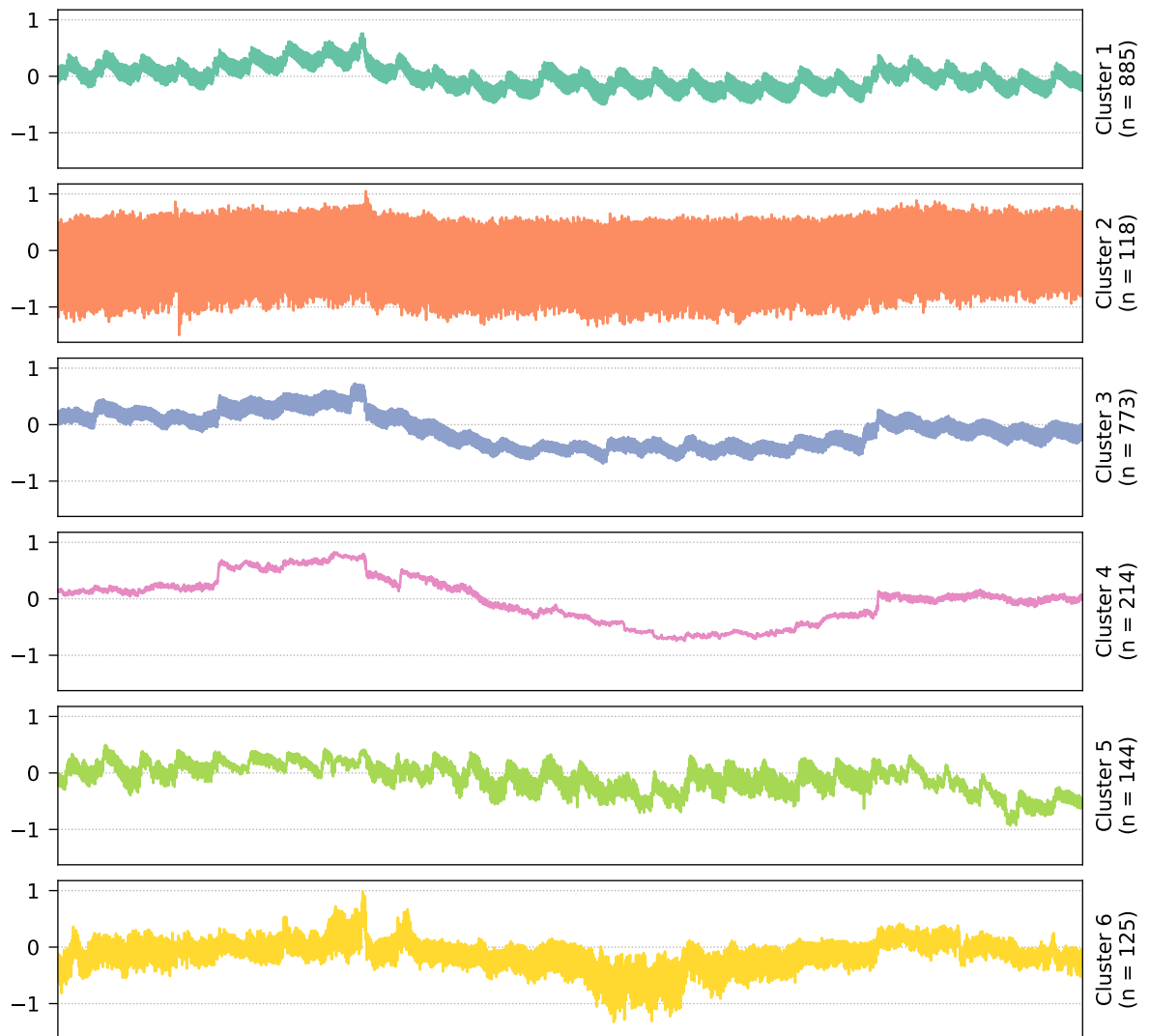
If users are not only interested in the clustering quality but also in the computational costs of the evaluated candidate methods, they can additionally utilize our run-time cost model. In our approach, we measure the absolute run-time costs, so all following results are highly dependent on the concrete machine used for clustering. We decided to only present the UCR dataset here, which suffices to show that our run-time cost model can successfully be used to investigate both the clustering quality as well as the computational costs. For our candidate methods, we opted for the same five clustering models introduced in [Section 5.6.2.3](#), which we evaluated on both the raw time series data as well as on feature-based representations. As feature sets, we decided to run all of our TSC groups with their eight variants listed in [Section 5.6.2.2](#), i.e., we created $5 \cdot 18 \cdot 8 = 720$ feature-based methods.²⁶ In combination with the five raw-based methods, we thus evaluated a total of 725 methods.

Our approach requires to specify the number of repeated runs r as well as both the lower and upper percentile-based thresholds p_l and p_u to determine the robust run-time measurement \bar{r} for each method. To get reliable results, we set them to $r = 30$, $q_l = 10\%$ and $q_u = 90\%$

²⁶The feature set catch22 from the previous evaluation only works on Unix-based systems. Unfortunately, no such system was available at the time of testing, which is the reason why catch22 is not evaluated here.

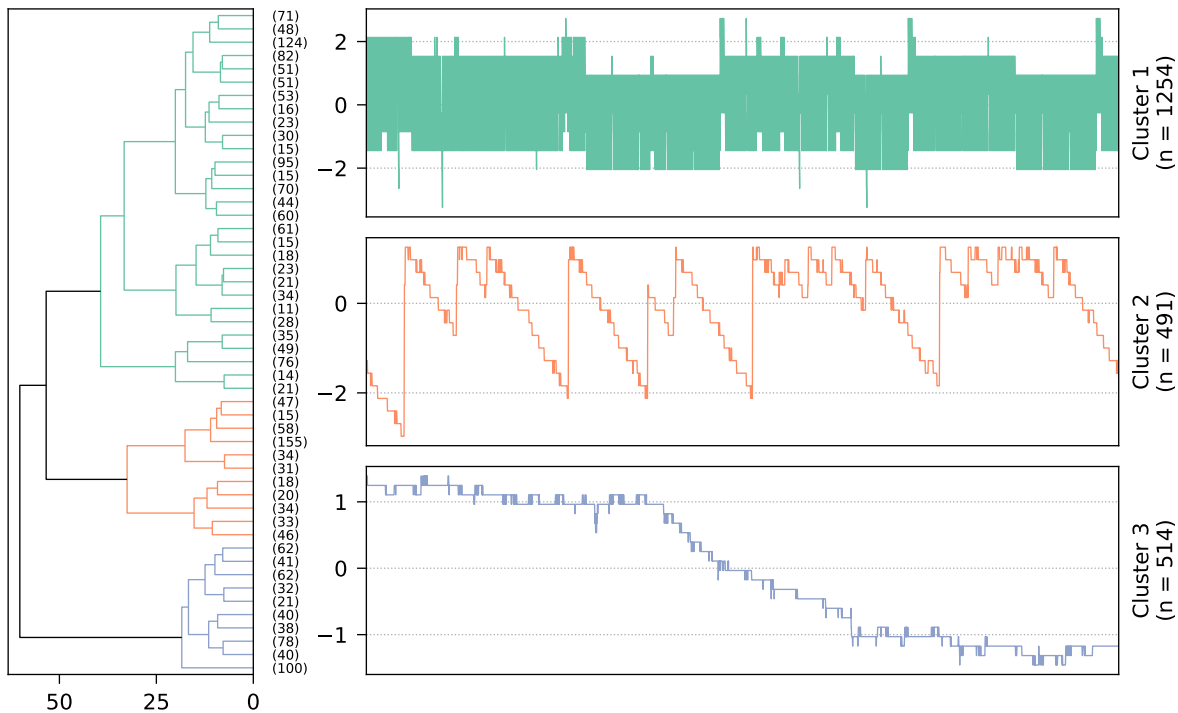


(a) Dendrogram (left) and representative time series (right) for the six newly identified clusters.

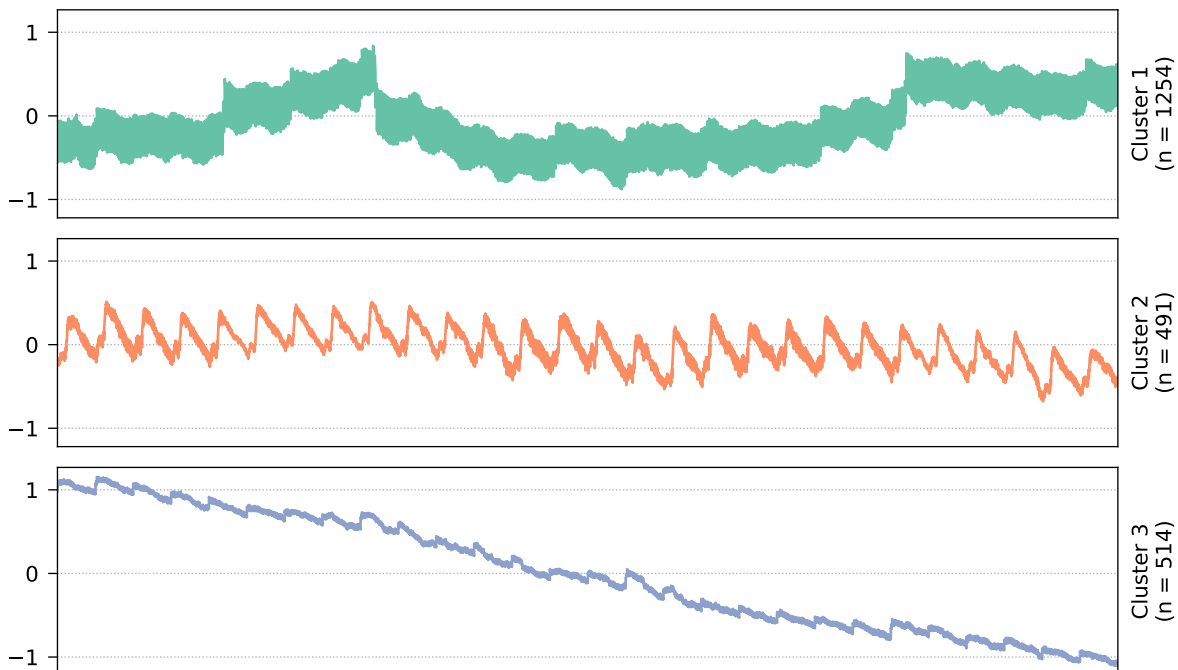


(b) Cluster time series averages.

Figure 5.32: Various results obtained when clustering the 2259 Disk Available % (D-03) series of the IMTS₂ dataset into six new clusters. The respective cluster sizes are denoted by n , and all time series contain 40320 data points (four weeks in one-minute resolution, ranging from 15.07.2019 00:00 UTC to 11.08.2019 23:59 UTC).



(a) Dendrogram (left) and representative time series (right) for the three identified clusters.



(b) Cluster time series averages.

Figure 5.33: Various results obtained when clustering the 2259 Disk Available % (D-03) series of the IMTS₂ dataset into three clusters using the entire set of TSC features instead of the *distributional* feature set. The respective cluster sizes are denoted by n , and all time series contain 40320 data points (four weeks in one-minute resolution, ranging from 15.07.2019 00:00 UTC to 11.08.2019 23:59 UTC).

(average of the middle 80% of 30 measurements). As the clustering quality evaluation metric, we once again used the adjusted Rand index (ARI). All evaluation runs were executed on a machine with an Intel Xeon E3-1245 v3 3.4GHz processor with four physical cores and eight threads, and 16GB of main memory. The implementations were written in Python 3.6.10, where the required libraries had the following versions: SciPy 1.4.1 [185] for the linkage model, scikit-learn 0.22.1 [131] for the implementations of k-means, BIRCH and the ARI metric, pandas 1.0.3 [140, 117] and NumPy 1.18.1 [73] for general data handling, joblib 1.14.1 [176] for parallel execution contexts, and tsfresh 0.15.1 [35], nolds 0.5.2 [151] and arch 4.13 [167] for our TSC implementation.²⁷

Figure 5.34 shows the results of all 725 methods for four selected UCR datasets via the quality-cost trade-off graphs, where all run times are measured in seconds. Each dataset is specified with the number of samples n it contains, i.e., the number of individual time series, and how many data points t such a time series consists of. This allows us to quickly see how fast the different methods executed with respect to the input data size, which can be seen in both the graph and its corresponding table that contains the Pareto front methods. For demonstration purposes, we set up some quality and cost thresholds²⁸ to focus our search for the best methods only on the relevant methods. For example, we can drastically reduce the methods down to 20 for dataset *ElectricDevices* (cf. Figure 5.34a), and the Pareto front further reduces them to eleven. These eleven methods are then the methods of interest, and we can now analyze how well they performed in terms of clustering quality and computational costs. If we select the *complexity* feature set, we get the best ARI but at the cost of the highest run time. If we can live with a small decrease in clustering performance, the *entropy* group yields much lower run times, especially if we opt for the k-means model. We might also choose the *blockwise distributional dispersion* feature set in combination with the k-means model, which has the lowest ARI but is significantly faster in return. Regarding run-time performance, the variants do not matter much (differences around 0.02 seconds). The other three UCR datasets yield comparable results (with different relevant methods) and can be interpreted analogously. Ultimately, it is up to the users whether quality or run-time cost is more important and which methods they choose in the end.

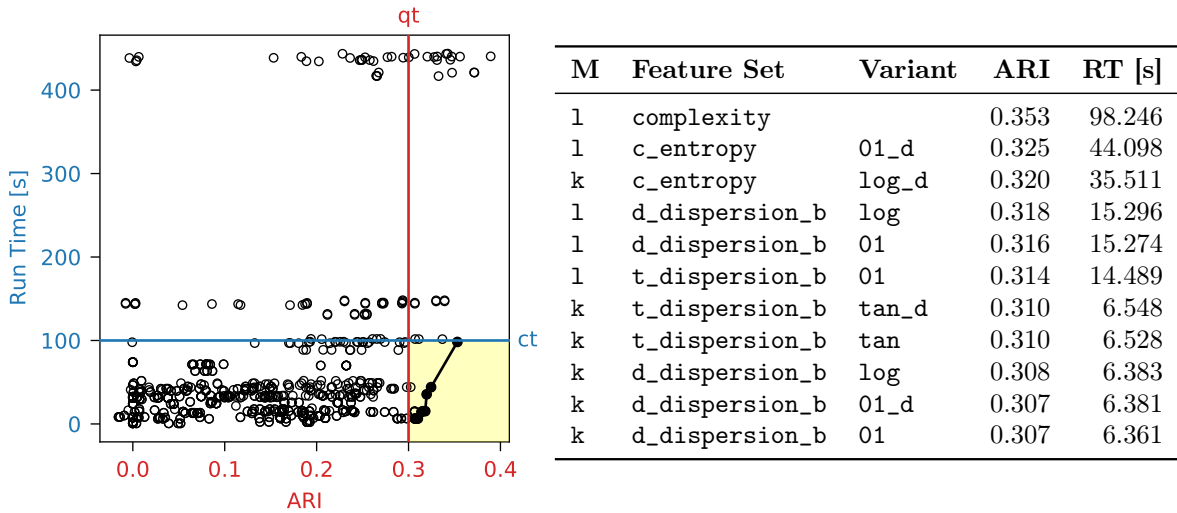
5.7 Discussion

In this section, we initially discuss some general aspects and then continue with the lessons we learned while working on this project. Afterwards, we list problems and limitations of our approach and finish with potential threats to validity.

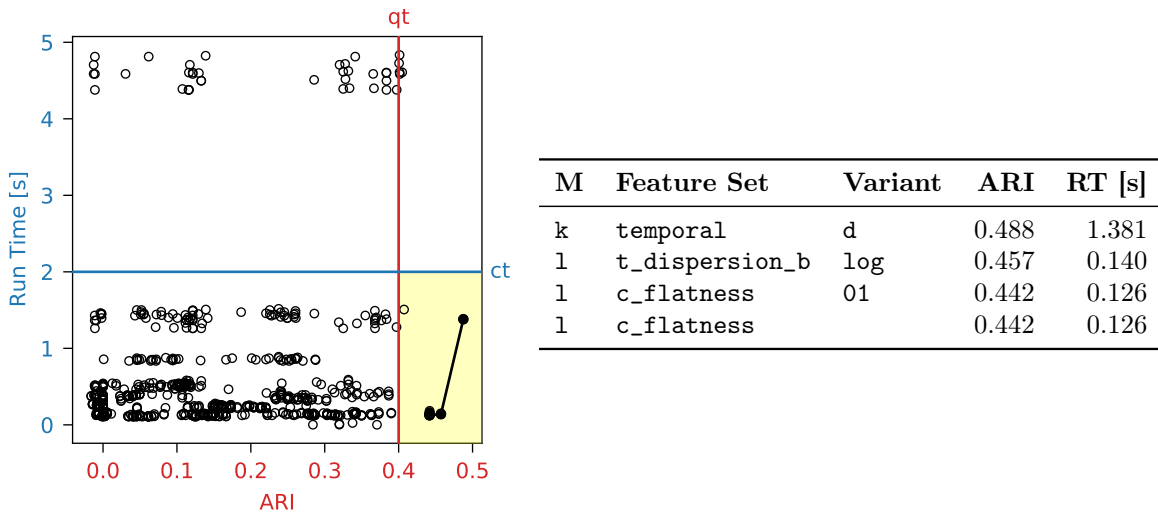
The first point of discussion is the creation of our feature sets. We proposed the time series characteristics (TSC) groups and subgroups, but of course, any kind of feature sets can be used. For example, we could create our own sets based on the TSC features in combination with their feature importance. Rather than redistributing the feature importance results back to their original TSC groups to obtain their merged importance, we could also simply use the n most important individual features and form a new feature set, which would then contain a mixture of features from various TSC groups. In this case, we would lose the group assignment that we carefully contrived, but if one is not interested in these groupings, then such an importance-based feature set could be an interesting idea for future work.

²⁷See also <https://github.com/cdl-mevss-m3/Time-Series-Characteristics>.

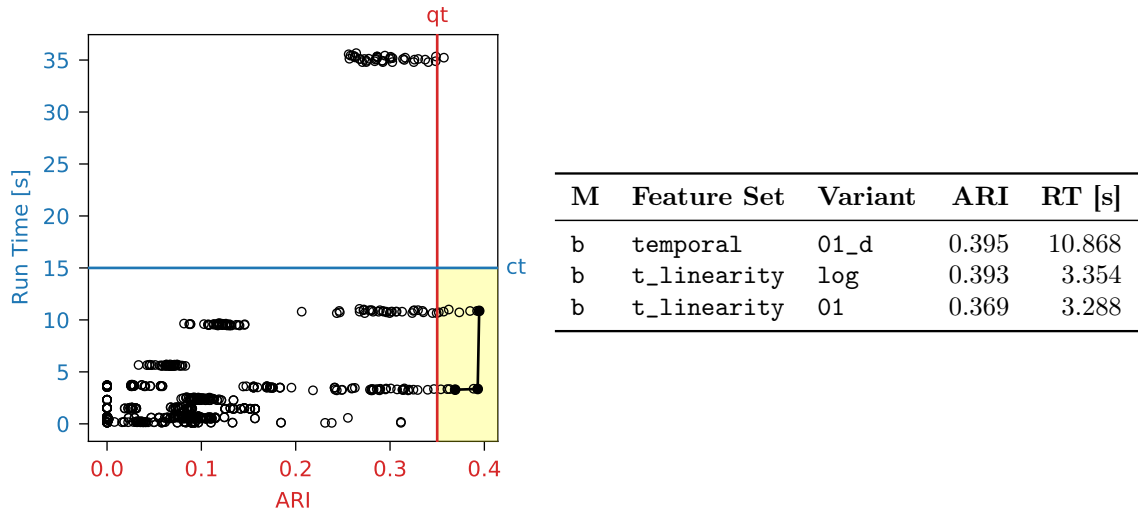
²⁸In our case, we selected arbitrary thresholds. Of course, in a real-world scenario, these thresholds should be chosen based on specific quality and performance criteria.



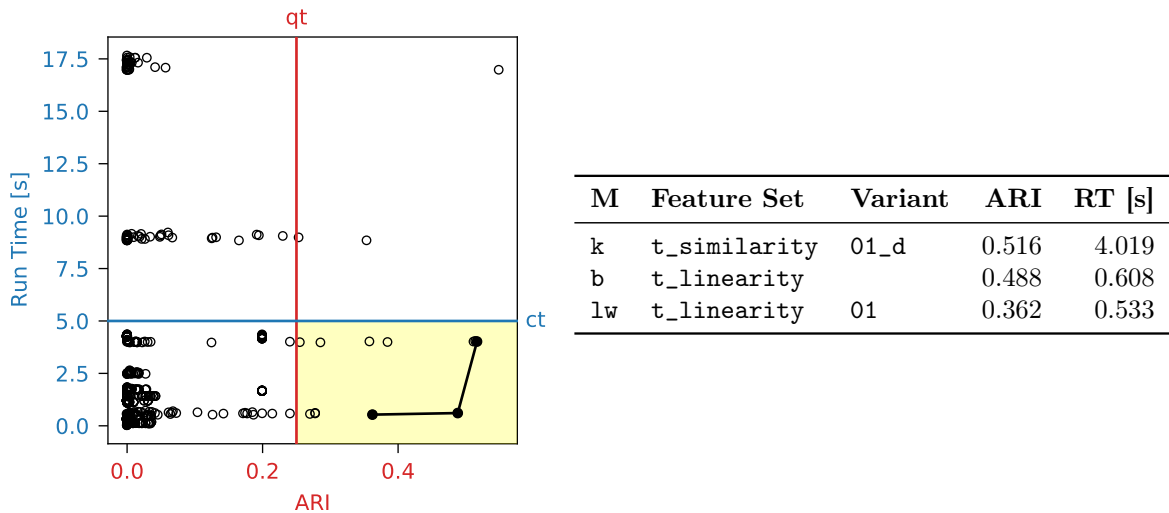
(a) Quality-cost trade-off graph for UCR dataset *ElectricDevices* (number of samples $n = 16637$, time series length $t = 96$). The lower quality threshold (qt) of $ARI \geq 0.3$ and upper run-time cost threshold (ct) of $\bar{r} \leq 100$ s result in 20 relevant methods (highlighted with yellow background), whose Pareto front (eleven methods) is listed in the table.



(b) Quality-cost trade-off graph for UCR dataset *FaceFour* (number of samples $n = 112$, time series length $t = 350$). The lower quality threshold (qt) of $ARI \geq 0.4$ and upper run-time cost threshold (ct) of $\bar{r} \leq 2$ s result in 13 relevant methods (highlighted with yellow background), whose Pareto front (four methods) is listed in the table.



(c) Quality-cost trade-off graph for UCR dataset *FiftyWords* (number of samples $n = 905$, time series length $t = 270$). The lower quality threshold (qt) of $ARI \geq 0.35$ and upper run-time cost threshold (ct) of $\bar{r} \leq 15s$ result in 13 relevant methods (highlighted with yellow background), whose Pareto front (three methods) is listed in the table.



(d) Quality-cost trade-off graph for UCR dataset *ItalyPowerDemand* (number of samples $n = 1096$, time series length $t = 24$). The lower quality threshold (qt) of $ARI \geq 0.25$ and upper run-time cost threshold (ct) of $\bar{r} \leq 5s$ result in twelve relevant methods (highlighted with yellow background), whose Pareto front (three methods) is listed in the table.

Figure 5.34: Quality-cost trade-off graphs and corresponding tables listing the Pareto front of the 725 evaluated methods for various UCR datasets. In the tables, M and RT are short for *Model* and *Run Time*. All run times are measured in seconds.

Another point is the fact that our diff-matrix does not show the absolute evaluation metric, e.g., the actual adjusted Rand index (ARI), so we do not know how well the methods performed in absolute terms but only in comparison to each other. While we could easily provide an additional table or figure that shows these absolute scores, we argue that this is not necessary at that stage. The primary goal of the diff-matrix is to show differences between methods to determine which ones performed better and which ones worse. Knowing the absolute evaluation metric would not change this ranking. Furthermore, the absolute scores are of little use, unless the labeled data contains the same clusters (e.g., historic data) that we seek in the unlabeled data, which could then serve as a performance estimate that we expect for this unlabeled data.

Lastly, we note that clustering is not limited to the data we used in selecting the best methods. In principle, we can use all kinds of data, since we did not train any model where the test data must match accordingly, e.g., such as in prediction and classification tasks. However, we should choose to cluster at least similar data (same domain, equally long time series), or otherwise, the selected method would lose significance. In other words, if we try to cluster completely different data than we used in the method selection process, we can no longer assume that the best-performing methods should also work here.

5.7.1 Lessons Learned

In the following, we present various lessons that we learned and general insights we gained when applying our approach on the UCR and IMTS datasets.

Clustering is feasible and beneficial in a multi-system environment. This is arguably the core lesson as clustering was the primary goal of our work. We showed that using time series characteristics enables us to extract useful cross-system clusters, which can then be the input for more detailed analyses and multi-system tool development. For example, we could develop a sophisticated multi-system forecasting and baselining model (for modeling the default time series behavior), where our findings can be a valuable starting point. For instance, given the results shown in [Figure 5.28](#), we could create three specific models for the three default behaviors and continue to integrate forecasting models, each designed to best fit the time series of the respective clusters [\[9\]](#). Such a tool could then be used throughout the different systems without having to explicitly develop a separate tool for each of the hundreds of software systems. Another possibility would be to create simple models for those systems which reside only within a single cluster, as can be extracted from the Venn diagram in [Figure 5.28c](#).

Features outperform raw data in terms of clustering quality. For the IMTS datasets, the results in [Figure 5.26c](#) and [Figure 5.26d](#) clearly indicate the superiority of feature-based time series clustering. Of course, raw-based clustering can work for other datasets such as the UCR time series archive (cf. [Figure 5.26a](#) and [Figure 5.26b](#)), where the best raw method was only outperformed by the full TSC group and the *temporal* group. Moreover, clustering raw data can potentially lead to significantly longer run times due to no dimensionality reduction, especially for longer time series. Consider the length of the feature vectors used by the various machine learning models: Given the IMTS₂ dataset, for example, the vector of the *distributional* group has a length of 34, compared to 40320 of the raw vector. However, it must be noted that the computational costs heavily depend on the selected clustering models and the (possibly expensive) features that should be calculated. Our run-time cost model can be helpful in this regard to obtain a concrete assessment.

Feature groups help to understand time series/cluster properties. Our TSC groups (cf. [Table 5.1](#)) assist engineers in quickly analyzing properties of time series or even entire clusters. For the UCR dataset *BirdChicken*, for example, [Figure 5.23](#) immediately reveals which groups

have a high impact and which ones are less important, and we can easily determine the differences and variances between classes/clusters. In contrast, simply having an unorganized set of features (or raw data altogether) is far less convenient and requires engineers to put much more time into gaining valuable insights from the results.

Finding the best methods requires domain as well as run-time cost considerations. With the countless number of machine learning models and their parameterizations, time series features and post-processing options that are at our disposal, running all possible combinations is infeasible. We thus have to limit what we apply our approach on, which heavily depends on both the domain of the data as well as on the ultimately chosen models and features. In our case, we only chose five clustering models in combination with selected sets of features (TSC groups, catch22, raw data), and we still ended up with 755 methods that we needed to evaluate. Our approach includes the analysis of feature importance, which can be helpful to potentially drop some irrelevant feature sets. However, there is no guarantee that any feature sets can be excluded in this step, so the initial choice of models and features is still essential to keep the run times manageable, even more so if (unlike in our evaluation) computational costs are critical, especially with increasingly large data sizes or changes in data patterns and characteristics that require a reevaluation.

5.7.2 Problems and Limitations

The requirement of labeled data can be problematic when such data is not available, which, unfortunately, is often the case in real-world scenarios. As mentioned in the approach and evaluation, we do provide alternatives to cope with this issue (generating labeled datasets ourselves by merging different time series sources, using publicly available labeled datasets), but as next steps, we should focus on an approach that can work solely with unlabeled data. Instead of the supervised random forest feature importance, we could look into unsupervised feature selection [53, 173], and we could replace the external evaluation metrics with internal evaluation metrics (cf. Section 2.4.4.1 on p. 23). On the other hand, this does come with additional challenges such as no longer knowing the ground truth, which makes the comparison of methods more difficult.

Our current approach is computationally expensive, since we need to run the full set of candidate methods on all selected datasets. This limits the overall capabilities because, for example, if we have limited hardware resources, we simply cannot include more models or different model parameterizations due to excessive run times, even if we wanted to. The concept of meta-learning [133, 183], whose goal is to learn from previous performances and transfer the results to new data and settings, could be a promising topic in future work.

Three limitations must be considered when using our run-time cost model approach. First, the run times are only valid on the specific machine they were measured on, which means that this machine should then also be the one where future data is expected to be clustered. Otherwise, the run-time costs can be misleading²⁹. The second point is the fact that measuring the run times of all candidate methods takes a significant amount of time itself, especially since we require multiple executions due to our robust measurement strategy. However, this can be done offline until a reevaluation is necessary. Lastly, the run-time cost model only measures how long it takes to cluster the entire data batch and not how long it takes to assign a new sample to an existing clustering (which is not even possible for some clustering models, such as the hierarchical clustering algorithms). Hence, it is important what the users want to

²⁹They might still be relatively comparable, i.e., judging which methods executed slower/faster than others, but even this is limited when the hardware and software components are different (parallelization, available memory, operating system, library versions).

accomplish. If they just want to get clusters for a batch of data, e.g., every one or two weeks, then this is perfectly fine. If they want to repeatedly cluster incoming time series, then the run-time cost model is possibly not applicable anymore because we would most likely use a clustering model that can be updated (e.g., k-means) to avoid re-clustering the entire data every time a new time series comes in. In such a case, the run-time measurements are not valid anymore because we would now need the time to update the model rather than the time it takes to cluster the entire data. We could extend our run-time cost model in future work to also support such a scenario.

5.7.3 Threats to Validity

We use a supervised random forest for selecting feature sets as guidance for the unsupervised clustering. We argue that the most informative features selected by the random forest should be informative for the clustering as well, since it tries to find those features that best separate the labels, i.e., the clusters. This is essentially the same idea in the unsupervised approach, where, for instance, some distance measure is used for separating the clusters, and clearly separable features should be clearly separable in distance as well. In the majority of the cases, our evaluation results support this claim when cross-analyzing the feature importance ranks in [Figure 5.24](#) and the diff-matrices in [Figure 5.26](#). For instance, the feature group *temporal similarity* was ranked as important for the UCR-merged dataset, and it was then among the top-performing methods (cf. [Figure 5.26b](#)). However, there are a few, scattered exceptions, so we should look into alternative feature importance assessments, such as incorporating unsupervised feature selection algorithms as already mentioned above.

Relying on the labeled data for the method ranking with respect to how our labeled datasets are created is another point of discussion. In the preparation phase, we merged the time series of different monitoring metrics and assigned labels according to these metrics to obtain our IMTS datasets. Two arguments against this procedure could be that the metrics are too similar, and that we then try to identify clusters within a single metric and not the merged, multiple metrics. We already discussed parts of this earlier (cf. [Section 5.5](#)), where we pointed out that we deliberately selected drastically different metrics precisely to avoid the issue of similar metrics that would indeed be problematic, since they are much more difficult to separate again. Regarding the second argument, we emphasize that we only want to identify those methods that should at least be able to separate the merged (sufficiently different) metrics, i.e., we try to find methods with a good “separation capability” or “pattern/cluster identification ability”, which we expect to also work when clustering the unlabeled, single-metric data. Other than that, there are no connections to the labeled, multi-metric data. Most notably, no form of model training takes place, where we would then apply such a trained model on the single-metric data afterwards. Naturally, the ideal scenario would be the access to already (manually) labeled, historic data of exactly the same type as the unlabeled data that we want to automatically cluster, in which case we would not need the merged, multi-metric dataset in the first place.

The selected models, features and variants represent not only a limitation but also a possible threat to validity, since we might have excluded some combinations that could potentially have performed much better and/or yielded different results. The variants are the least important part, as we showed in the evaluation that there are only small differences (if any) in most cases. For the features, we gathered a carefully chosen set of time series characteristics (TSC) from literature that cover various properties (represented by our groups and subgroups), and we compared them to a state-of-the-art feature set (catch22), which corroborated their classification as well as clustering performance. Thus, we are confident that our TSC groups are sufficient,

especially when considering that merely adding more features does not necessarily result in significantly improved performance [110]. Regarding the clustering models, we opted for a reduced set of five models (k-means, BIRCH and agglomerative clustering in three variations). We chose these models because they have successfully been applied throughout related work and because of their scalability when clustering large quantities of data. Nonetheless, many more clustering algorithms exist, and also hyperparameter tuning can make a drastic difference, both of which we should evaluate and investigate in future work.

Lastly, we discuss external threats to validity. Our main evaluation was performed on the two IMTS datasets, i.e., industrial monitoring data from our industry partner, so the results are not generally applicable since they heavily depend on the domain, characteristics and properties of this data. Hence, we cannot assume that the clustering methods identified by our automatic method ranking approach can be used for clustering arbitrary time series data. However, we additionally showed that the approach can work with other data than our IMTS datasets as well, namely when clustering data from the UCR archive, which contains time series from a variety of domains. While there were some overlaps (e.g., the full TSC feature set performed well in all datasets), the UCR evaluation also resulted in some different, top-ranked methods (e.g., methods based on the raw data or other TSC groups), i.e., methods more suited for the UCR data. Considering this comprehensive and diverse evaluation of both the IMTS and the UCR data, we are confident that our approach can be applied in different scenarios and environments as well.

5.8 Related Work

Clustering is a large and active field of research, so we focus on the most relevant topics. We start with features and characteristics that can be extracted from time series, continue with automatic clustering selection approaches, then present work on analyzing real-world, industrial (software) systems, and lastly, we talk about literature related to run-time costs.

5.8.1 Features for Time Series

Clustering and classifying time series has caught the interest of many researchers, both regarding raw-based as well as feature-based approaches [3]. In 2006, Wang et al. [187] presented 13 features for clustering time series. Their feature set included trend, seasonality, periodicity, serial correlation, skewness, kurtosis, chaos, non-linearity and self-similarity, which they calculated for both raw and trend plus seasonally adjusted data. The authors of [64] introduced a large collection of time series analysis operations containing about 1000 distinct features (over 9000 with different parameterizations) to identify structures within a diverse set of time series data across multiple domains. Two of these authors then continued with this work by applying the features in a classification scenario using 20 UCR time series datasets [63], and they published their Matlab-based framework *hctsa* for calculating those features [62]. Fulcher then also presented an overview of feature-based time series analysis [61]. Since the thousands of features of *hctsa* can be confusing and computationally expensive, the authors of *catch22* [110] reduced them to a set of 22 minimally redundant features, which still provide a strong classification performance, based on the UCR datasets. They decreased the computational time by a factor of 1000, while only reducing the average classification accuracy by about 7% and obtaining equally good results as Hyndman et al. [82]. Christ et al. [36, 35] presented the Python-based framework *tsfresh*, whose goal is to extract over 60 time series features (more than 1200 with different parameterizations). Another collection of features for classification purposes was introduced by Baldan and Benítez [8]. Besides listing a total of

41 different features, the authors also assigned them to two groups (ten complexity measures such as entropy, and 31 representative features such as autocorrelation and linearity) to aid interpretability. While this is a step in the right direction, we provide a full grouping into four main time series properties, which we split into even more detailed subgroups that allow us to analyze and utilize specific properties and characteristics. Furthermore, we selected a diverse and representative set of time series features based on multiple literature sources that already showed their effectiveness, all without being overwhelmed by thousands of features with respect to data analysis complexity as well as computational costs.

5.8.2 Automatic Clustering Selection

In this section, we discuss automatic filtering and selection approaches of both models, features and combinations thereof. Three of the above research groups not only introduced a set of features but also included a feature selection process. Wang et al. [187] proposed a greedy forward search for feature selection, tested their approach on the (old) UCR datasets, two synthetic datasets and a real-world dataset, and reported good clustering results. Fulcher et al. [64] filtered their 9000 features by removing redundant ones with the help of k-medoids clustering (similar to k-means but with medoids as cluster centroids), where they managed to reduce the number of features to 200 with a negligible increase in variance when clustering a mixed set of time series data. Christ et al. [36, 35] added an automatic feature selection based on statistical tests and labeled data. The problem of choosing the right algorithm for a given task was already formulated in 1976 by Rice, who called it the algorithm selection problem [143]. In the machine learning community, this was translated into a more general model selection problem, and much research focused on creating automated machine learning (AutoML) pipelines, which incorporate data cleaning, feature selection, model selection and hyperparameter optimization [75]. However, these tools are designed for classification and regression tasks, which means they are not applicable to clustering problems. Nevertheless, some experiments have been conducted with respect to the unsupervised model selection. In the area of gene expression data, the authors of [87] investigated the effect of 15 distance measures (ten correlation-based, four standard distance-based, five tailored to short gene time series) and four clustering models (k-medoids, three hierarchical variants) on 52 microarray datasets, containing both labeled and unlabeled data. For the former, they used the adjusted Rand index (ARI) for evaluating the performance, and for the unlabeled data, they calculated the Silhouette score. Mori et al. [123] automatically selected the best among five distance/similarity measures by training a classifier on the characteristics of labeled time series databases (e.g., number of time series, global shift and trend), given the performance of 15 parameterizations of k-medoids and four hierarchical clustering variants. They evaluated their approach using 45 UCR datasets and 555 synthetic datasets. In contrast, our approach identifies and ranks the best raw-based as well as feature-based clustering methods, i.e., we investigate clustering algorithms, features and variants (post-processing) as a full model selection approach [132].

5.8.3 Analysis of Industrial Systems

The majority of literature regarding the analysis of industrial, real-world (software) systems can be found in the area of workload characterization [29]. Much work focuses on the public trace datasets from Google [83, 141] (25 million tasks running on 12500 hosts over a period of one month in five-minute resolution) and Alibaba [70] (for the 2017 trace: 1300 machines over a period of twelve hours in different resolutions), and many researchers and practitioners evaluate their own (often private) datasets. However, to the best of our knowledge, no work contains such detailed feature-based clustering as proposed in this thesis.

5.8.3.1 Statistical Analysis

The following work mostly focuses on presenting statistical results, which provide detailed insights into global data characteristics. In [17], thousands of servers from different data centers hosting numerous enterprise-sized customers were analyzed. They extracted two years of monitoring data covering CPU, memory, disk, file system and network metrics, and, among others, presented various statistics such as total average resource utilization, average utilization across the 2-year period (monthly resolution), including groupings into customers or data center locations (countries). The work in [18] is an extension, where they focused on resource characterization of CPU, memory and disk metrics, and presented detailed statistical insights including cumulative distribution functions as well as correlations between resources. The authors of [166] analyzed data in five-minute resolution collected from two traces, one with 1250 virtual machines (VMs) over a period of one month, and another one with 500 VMs over a period of three months. The data traces included various CPU-, memory-, disk- and network-related metrics. They included several statistical evaluations, ranging from global measures (mean, median, standard deviation, etc.) and multiple cumulative distribution functions to evolution over time and correlation analyses to check resource dependencies. In [109], the Alibaba trace data was analyzed and various results, ranging from CPU and memory plots for each individual machine to merged utilization evolving over time, were presented. The paper presented in [39] characterizes the workload from Microsoft Azure. These characteristics were used for learning and prediction, which could then be utilized by resource manager systems, e.g., for sophisticated scheduling. Di et al. [50] investigated the differences between cloud and grid workloads by using the Google cluster trace and data from multiple grid/HPC (high performance computing) systems. They presented various job-related statistics and also global analyses on CPU and memory usage. Zhang et al. [203] also analyzed the Google cluster trace data and studied global statistics on CPU, memory and disk metrics for each compute cluster.

5.8.3.2 Applied Clustering

We continue with work on real-world systems where some sort of clustering was applied. Thalheim et al. [179] presented *Sieve* with the goal to get actionable insights from various time series metrics (CPU, memory, disk and network, etc.) that were collected from components of a distributed system. The framework consists of a metrics reduction part and a metrics dependency extractor. For each system component, k-Shape clustering [129] is applied in the reduction part on the raw data to create groups of similar time series. A representative metric is picked from each such group, which is the input for the extraction part. Based on experiments on the authors' applications, they stated that seven clusters per component were enough, considering that each component provided up to 300 different metrics. Besides global statistical evaluations, the authors of [88] also investigated clustering based on CPU and memory metrics of the server machines in the Alibaba trace data. Using k-means, they could identify seven different machine types. Streiffer et al. [174] proposed the tool *Minerva*, whose main purpose is clustering time series that were collected from components of a microservice-based system, extracting causality and finally making predictions of time series to detect anomalies. In the clustering step, the authors grouped all metrics into similar clusters with k-Shape. Afterwards, only the cluster centroids were used for subsequent processing, which drastically reduced computational costs. Canali and Lancellotti [30] also utilized cluster representatives to limit the amount of their data, which comprised eleven infrastructure monitoring metrics (CPU, memory, disk, network) of 110 VMs in five-minute resolution. For each VM, the authors first calculated the correlation between all its time series and then applied k-means clustering to

obtain groups of similar VMs based on these correlation values. The authors of [92] also created clusters of similar VMs using CPU utilization time series data with the goal to predict the workload of individual VMs. In contrast to many other papers, they did not use unsupervised machine learning but proposed a new co-clustering algorithm which determines similar VMs according to the variation-based *consistency* measure. They evaluated their approach on the data of 1212 VMs and identified 98 co-clusters, of which 65 were predictable. Xue et al. [197] introduced *PROST*, a tool to predict workload time series that were collected from large-scale data centers. To reduce the amount of data before the prediction, clustering is first performed on the raw time series (either with dynamic time warping or correlation-based clustering), followed by removing multicollinearity to obtain a set of representative signature series. The goal of *Sibyl* [208] is host load prediction, but again, to reduce dimensionality beforehand and drop irrelevant metrics, the authors used k-Shape to create clusters of similar time series from a total of 17 different monitoring metrics of 176 machines. Using k-means, the authors of [165] determined workload clusters within two datasets (CPU, memory, disk, network for the trace in [166], and task duration, CPU, memory, disk for the Google trace data), which they then set as ground truth for workload classification. Gupta et al. [72] combined system log information and performance metrics for anomaly detection. They first detected context patterns (states of a machine) with k-means and the log data, which they separated further by identifying metric patterns, for which they used a modified k-means algorithm that clusters metrics based on the similarity of their top k components obtained by a principal component analysis. Shortly before the release of Google’s large trace data, Chen et al. [34] analyzed a smaller subset (only 375 minutes available, monitored metrics only contain used CPU cores and memory per task). They could extract eight job clusters based on 14 normalized characteristics (including extracted statistics on used CPU cores, memory and the memory-core-ratio). As an extension of [50], Di et al. [49] applied an optimized k-means algorithm on the Google trace data on the granularity of applications. Besides clustering task events according to their task statistics, they also identified workload clusters using the mean CPU workload and mean memory workload. The majority of the above work utilizes clustering based on the raw time series data. We, on the other hand, additionally provide an in-depth clustering based on features and present detailed results of several infrastructure monitoring time series from multiple, independent software systems, where we also focus on statistics specifically addressing this multi-system environment.

5.8.4 Run-time Costs

While there exist numerous approaches on improving the efficiency of existing algorithms, models and feature calculations themselves, the explicit comparison of the run-time costs in combination with clustering quality of a set of candidate methods as we propose in this thesis has not yet been focused on. However, meta-learning [133, 183], i.e., learning from previous performances and transferring the results to new scenarios, has caught the interest of some researchers. For example, the authors of [23] proposed the Adjusted Ratio of Ratios (ARR) metric that ranks algorithms based on their accuracy as well as run-time costs, which they then incorporated into a meta-learning approach for selecting learning algorithms. Later, Abdulrahman et al. [2] presented further improvements to exclude redundant and non-competitive algorithms. Hutter et al. [80] created so-called empirical performance models for predicting the run time of various machine learning algorithms and their parameterizations, given various input feature sizes and sample sizes. Since our current run-time cost model requires a full evaluation of all methods and all datasets, meta-learning seems to be a promising topic in future work, where we could predict both the clustering quality as well as the computational costs, which would significantly speed up our approach.

5.9 Outlook

There are many ways how we could improve our feature-based time series clustering selection approach. As already mentioned in the discussion section (cf. [Section 5.7](#)), the most pressing matter would be the adaptation to purely unlabeled data, i.e., changing our supervised feature importance as well as external evaluation metric diff-matrix comparison to techniques that do not require labeled data. Moreover, we could test much more methods, namely by analyzing different hyperparameters of the clustering models and, more generally, by evaluating further unsupervised algorithms (based on features as well as raw time series data), so we get even better insights into how well each method performs and which ones work best for the data of interest. Since increasing the number of methods naturally increases the time required for their evaluation, a logical next step would be to look into meta-learning as discussed above, which would also greatly benefit our current run-time cost model. The run-time cost model itself could also be extended to support online clustering updates in addition to the offline batch processing. With this outlook, we conclude the third and final part of this thesis.

Chapter 6

Conclusion

In this thesis, we presented our work on three major topics in the area of a multi-system environment (provided by our industry partner Dynatrace), where each system is independent of each other and consists of various components where data such as events, component properties and time series are collected. Analyzing such multi-system data can potentially yield advantages and reveal interesting insights, which includes coping with insufficient single-system data, combining and merging data across multiple systems, and identifying common multi-system patterns. Despite these promising benefits, research on data analysis and error analytics in such a multi-system environment is still lacking, which is why the main focus of this thesis was the analysis of multi-system data.

Our first part covered a topology-driven process crash analysis approach, where we investigated how process crash events and software technologies running on these processes could be related. To this end, we recorded how often such technologies crashed during their lifetime and how often they did not crash, which is stored in our so-called software technology tuples. Afterwards, we aggregated the results of all recorded systems and ranked the tuples based on a custom metric that rewards tuples that crashed often and in many systems. Leveraging crash properties such as the occurred exception, we analyzed the top-ranked tuples by grouping the tuples' crashes first according to the selected property and then the systems where the crashes occurred. This revealed common crash properties across multiple systems, where fixing a potentially common root cause could benefit all affected systems. Our evaluation of over one year's worth of monitoring data from 500 software systems yielded promising results. However, we did not have access to more detailed crash properties (e.g., library versions or stack traces) to enhance our approach.

Therefore, we decided to include the time series data in the second part of the thesis, where we introduced a multi-system event prediction approach. We wanted to investigate if certain performance-related events (service slowdowns) can be explained by infrastructure monitoring time series (CPU, memory, disk and network metrics), which, in turn, would mean that such events could be predicted solely based on these time series. We thus developed a sophisticated data preprocessing framework to extract the complex multi-system event and time series data, and then we trained various (single- and multi-system) supervised machine learning models using data from 57 different software systems covering a monitoring period of 20 days. In the testing phase, we started with a first (balanced) evaluation to see whether the service slowdown event prediction can work in the first place. After promising initial results, we continued with real-world testing scenarios with drastically unbalanced data, where we observed a significant drop in prediction performance. Further investigations (data augmentation and synthetic data) revealed that our event prediction approach unfortunately suffers from an upper bound

performance limit when applied within such real-world testing scenarios, and that our initial experiments already yielded poor results when addressing the data imbalance, indicating the possibility that the infrastructure monitoring time series might just not contain enough or the right information to explain service slowdowns.

In the third and last part of the thesis, we thus decided to drop the events and to solely focus on the time series data. We presented a feature-based time series clustering approach, whose main objective is to rank a set of candidate methods, where a method represents a triplet of an unsupervised clustering model, a feature set and options how to post-process the features. We also introduced our own feature sets, namely the time series characteristics (TSC), where we carefully collected various features from related work and assigned them to groups according to which properties of time series they represent. Given labeled datasets, our approach yields so-called diff-matrices, where we can easily visualize how well all candidate methods performed in comparison to each other, which allows us to select one of the best-performing methods for clustering the unlabeled data. In the evaluation, we included both the UCR time series archive as well as two infrastructure monitoring time series datasets from our industry partner (one with over 30000 time series from over 600 different software systems over the period of two weeks, the other with over 8000 time series from eight Dynatrace-internal systems over a period of four weeks), and we obtained several promising and interesting results, especially when analyzing the system distribution within the discovered clusters. As an addendum, we also presented a run-time cost model, where users can not only choose the best-ranked methods according to the clustering quality but also to their run-time costs, which is often important in real-world scenarios.

There are still plenty of different topics and challenges in our multi-system environment waiting to be discovered and researched, but this would simply go beyond the scope of a single project. With this closing remark, we thus conclude this thesis.

Appendix A

Background

In this appendix chapter, we provide additional information regarding the background chapter presented in [Chapter 2 on p. 5](#).

A.1 Feature Importance

Here, we provide the complete feature importance calculation for the example we presented in [Figure 2.9 on p. 22](#), i.e., ten samples characterized with three features f_1 , f_2 and f_3 . For convenience, the decision tree is also shown in [Figure A.1](#) below.

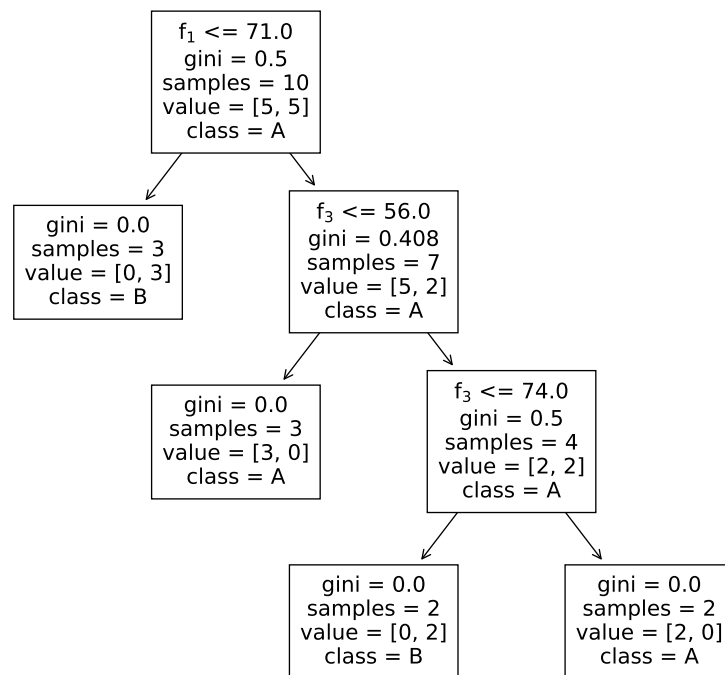


Figure A.1: The same example of a decision tree after fitting ten samples as shown in [Figure 2.9 on p. 22](#).

The default scikit-learn implementation [\[131\]](#) calculates each feature importance for non-leaf nodes as defined in [Equation A.1](#), which is the weighted impurity decrease [\[177\]](#):

$$\frac{N_t}{N} \cdot \left(\text{impurity} - \frac{N_{tR}}{N_t} \cdot \text{impurity}_R - \frac{N_{tL}}{N_t} \cdot \text{impurity}_L \right) \quad (\text{A.1})$$

“where N is the total number of samples, N_t is the number of samples at the current node, N_{t_L} is the number of samples in the left child, and N_{t_R} is the number of samples in the right child.” [177]. In the example, the impurity is the Gini importance. Given this formula and the decision tree in [Figure A.1](#), we can now calculate the importance of each of the three features ($N = 10$ since we have ten samples in total):

- f_1 : This feature appears in only a single node of the tree (the root node), and the feature importance thus evaluates to:

$$\frac{10}{10} \cdot \left(0.5 - \frac{7}{10} \cdot 0.408 - \frac{3}{10} \cdot 0.0 \right) = 0.2144$$

- f_2 : This feature was not selected by the decision tree at all, so it automatically gets an importance value of 0.
- f_3 : This feature appears in two nodes of the tree, so we calculate the importance for each node as follows:

$$\begin{aligned} \text{node}_1 &= \frac{7}{10} \cdot \left(0.408 - \frac{4}{7} \cdot 0.5 - \frac{3}{7} \cdot 0.0 \right) = 0.0856 \\ \text{node}_2 &= \frac{4}{10} \cdot \left(0.5 - \frac{2}{4} \cdot 0.0 - \frac{2}{4} \cdot 0.0 \right) = 0.2 \end{aligned}$$

The aggregated feature importance evaluates to $0.0856 + 0.2 = 0.2856$.

The final step is normalizing the features to achieve a total importance of one, i.e., we divide each feature by the total sum of all feature importance values, which is $0.2144 + 0 + 0.2856 = 0.5$ for features f_1 , f_2 and f_3 . The normalized feature importance values thus result in $\frac{0.2144}{0.5} \approx 0.43$ for feature f_1 , $\frac{0}{0.5} = 0$ for feature f_2 and $\frac{0.2856}{0.5} \approx 0.57$ for feature f_3 .

Appendix B

Topology-driven Crash Analysis

In this appendix chapter, we provide additional results and figures that allow more detailed insights into the data, the approach and the evaluation presented in [Chapter 3 on p. 37](#).

B.1 Data Exploration

This section covers additional information and figures for the raw data we had at our disposal for evaluating our tuple-crash analysis approach.

In [Figure B.1](#), the total number of raw, annotated (cf. [Table 3.2 on p. 41](#)) 1-tuples and 2-tuples across the 15 months of export data before the tuple merging step are shown, more formally, the tuple count is $\sum_{s \in S} |T_s|$, where $|T_s|$ is the size of the set of annotated tuples of system s , and S represents the set of all systems.

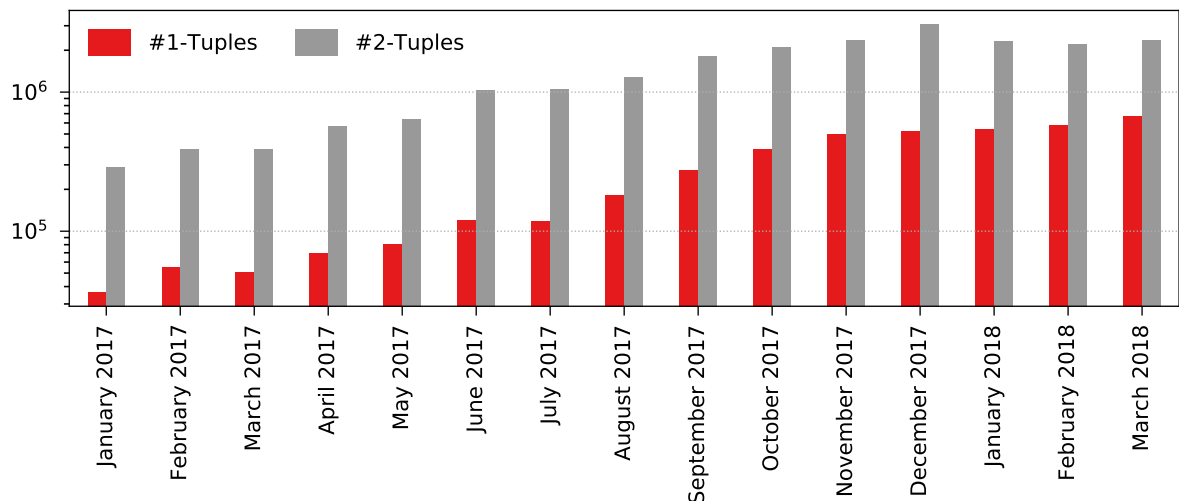
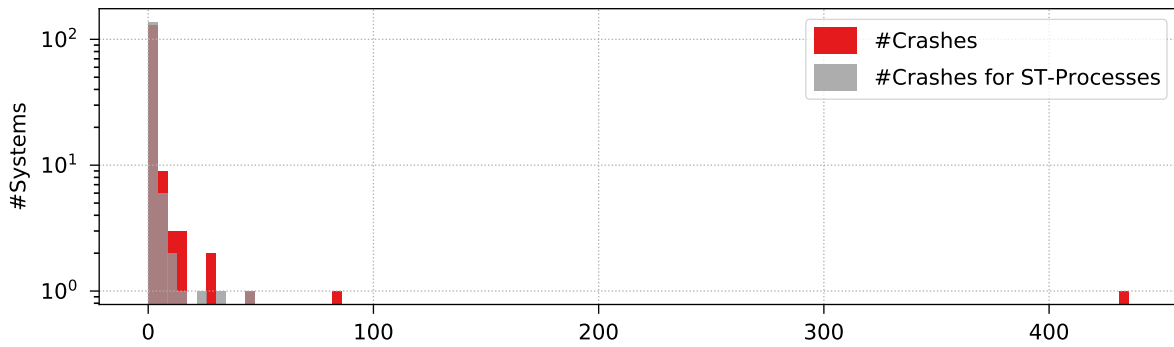
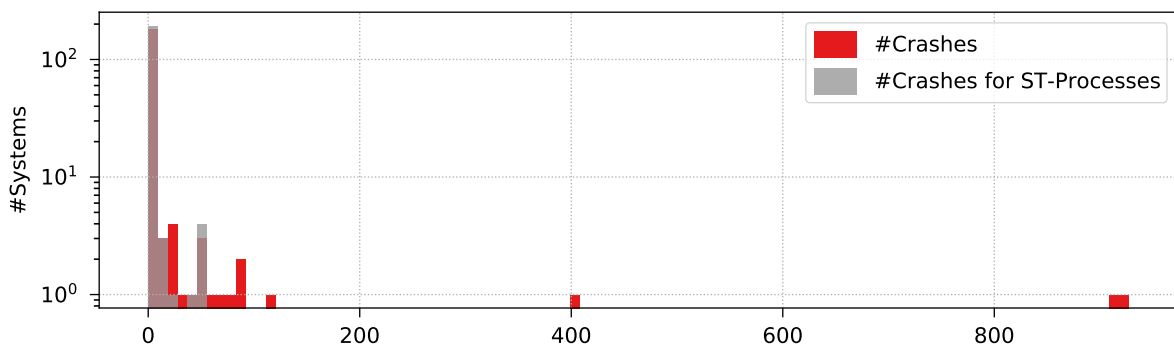


Figure B.1: The total number of created but not yet merged 1-tuples and 2-tuples, i.e., the raw, annotated tuples for each month, displayed on a logarithmic scale.

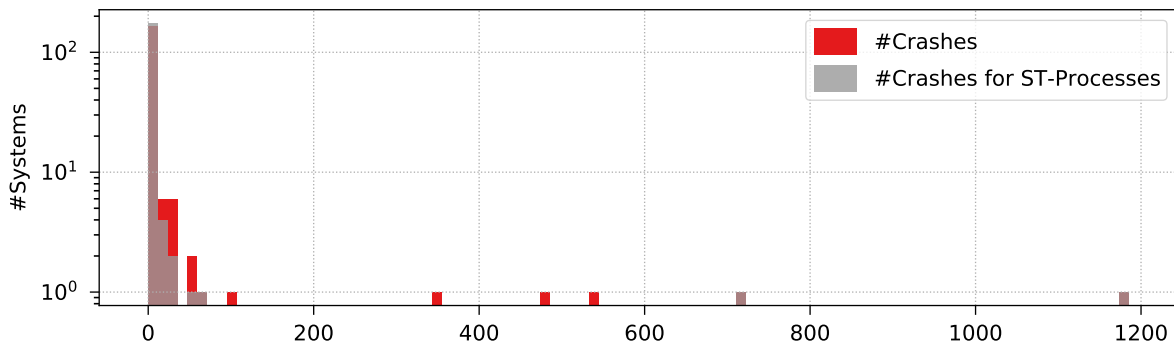
[Figure B.2](#) shows the histograms for all the one-week exports of the 15 months (cf. [Table 3.5 on p. 46](#)), which indicate the distribution of the crashes (x-axis) across the different systems (y-axis, logarithmic scale). We can see that the crash count is comparatively low in most systems, with a few (heavy) outliers, i.e., we have right-skewed/positively skewed distributions.



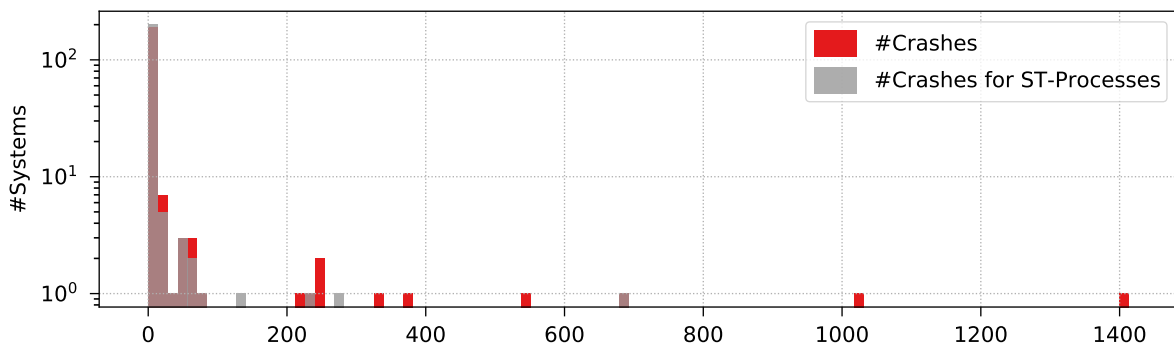
(a) Export January 2017: Histogram showing the distribution of 788 crashes and the filtered 192 crashes of processes with at least one software technology across 149 systems.



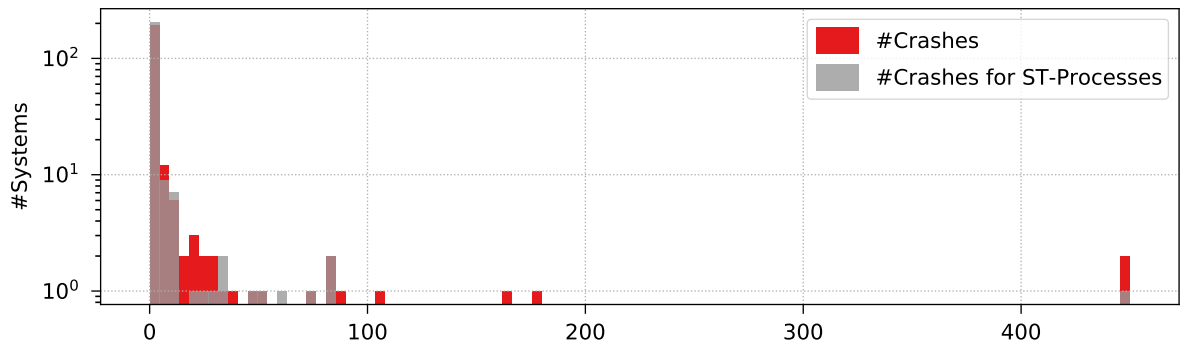
(b) Export February 2017: Histogram showing the distribution of 3201 crashes and the filtered 387 crashes of processes with at least one software technology across 203 systems.



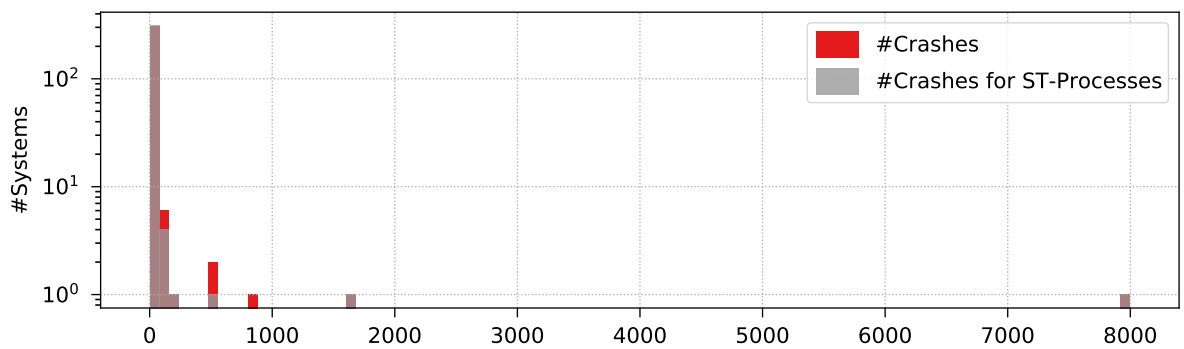
(c) Export March 2017: Histogram showing the distribution of 4027 crashes and the filtered 2253 crashes of processes with at least one software technology across 185 systems.



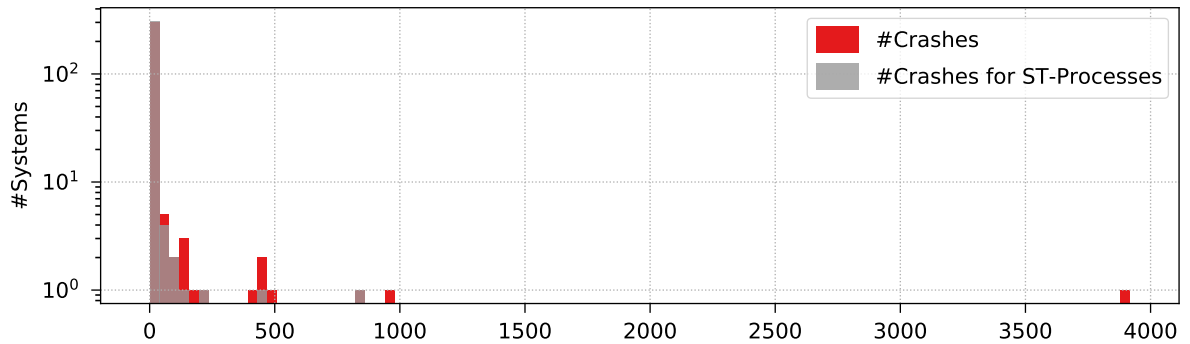
(d) Export April 2017: Histogram showing the distribution of 6103 crashes and the filtered 1967 crashes of processes with at least one software technology across 216 systems.



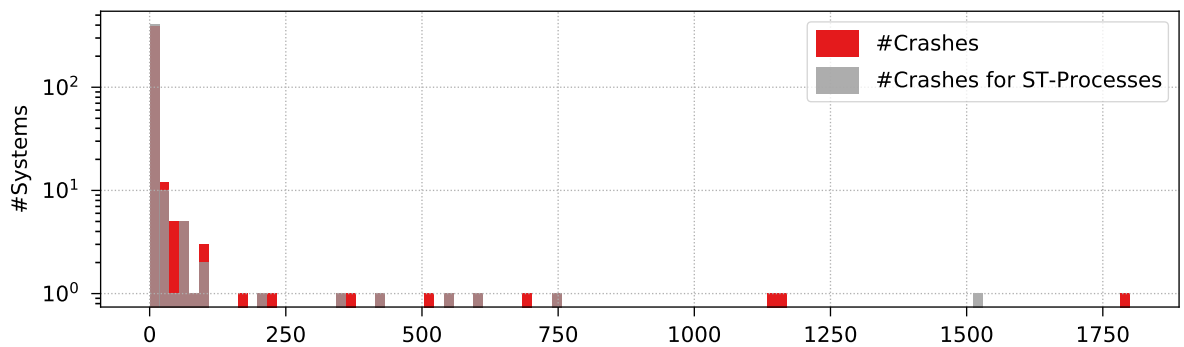
(e) Export May 2017: Histogram showing the distribution of 2216 crashes and the filtered 1154 crashes of processes with at least one software technology across 233 systems.



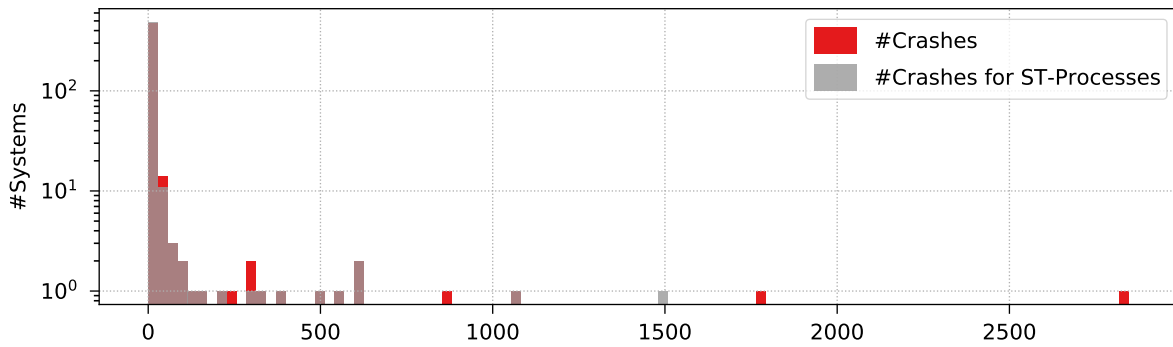
(f) Export June 2017: Histogram showing the distribution of 13300 crashes and the filtered 11390 crashes of processes with at least one software technology across 319 systems.



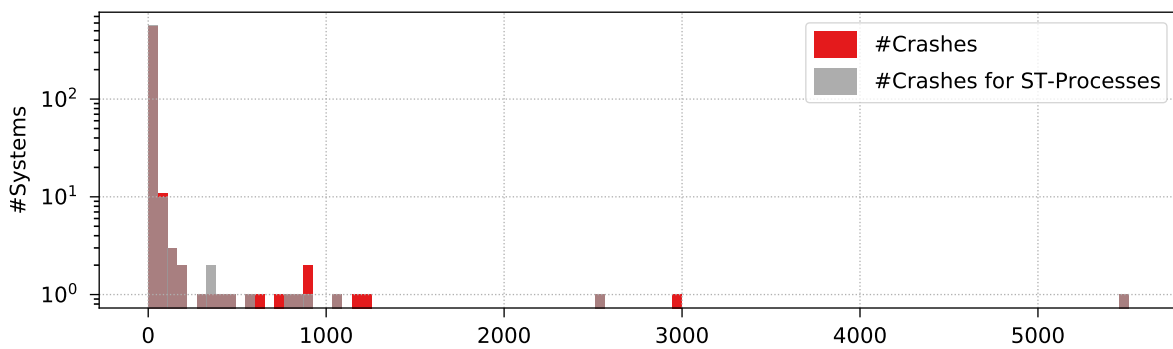
(g) Export July 2017: Histogram showing the distribution of 9159 crashes and the filtered 2476 crashes of processes with at least one software technology across 318 systems.



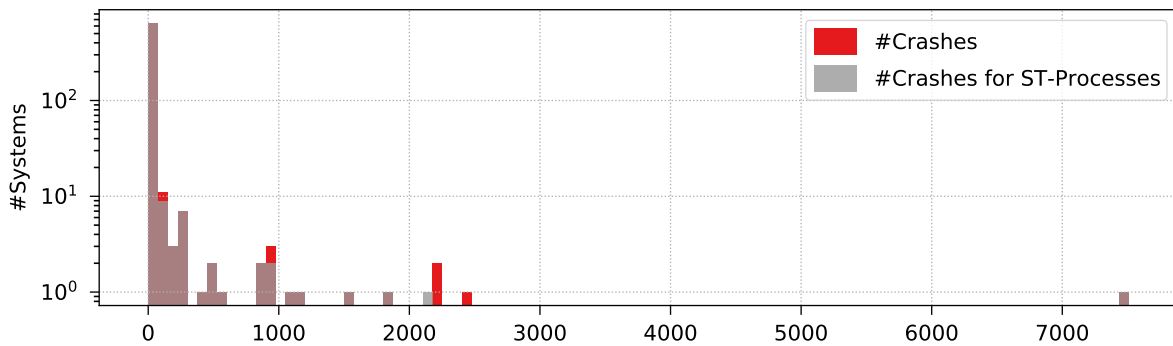
(h) Export August 2017: Histogram showing the distribution of 10466 crashes and the filtered 5550 crashes of processes with at least one software technology across 430 systems.



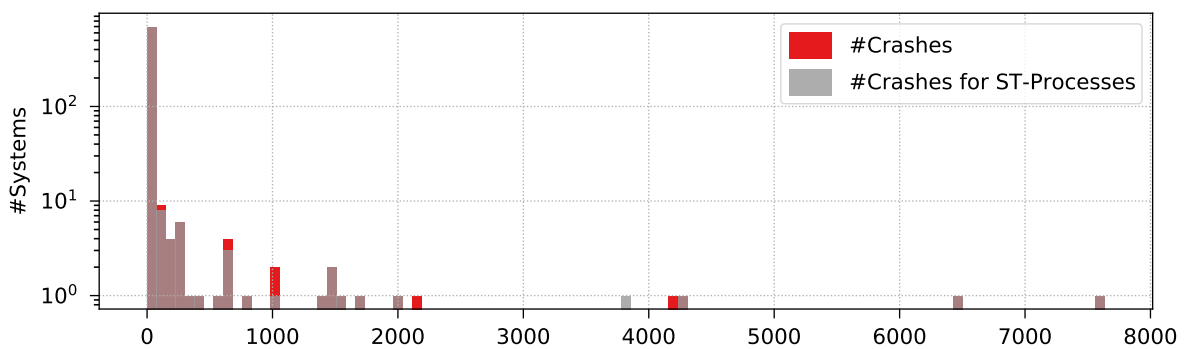
(i) Export September 2017: Histogram showing the distribution of 12353 crashes and the filtered 7653 crashes of processes with at least one software technology across 514 systems.



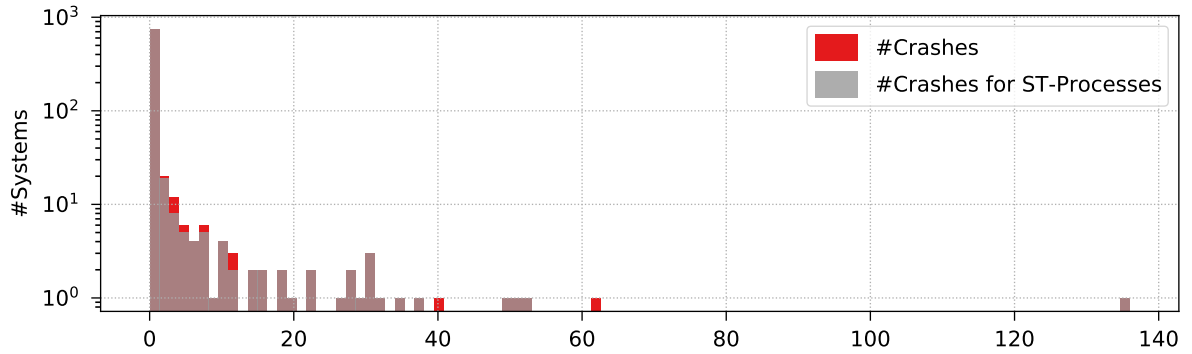
(j) Export October 2017: Histogram showing the distribution of 24266 crashes and the filtered 16682 crashes of processes with at least one software technology across 591 systems.



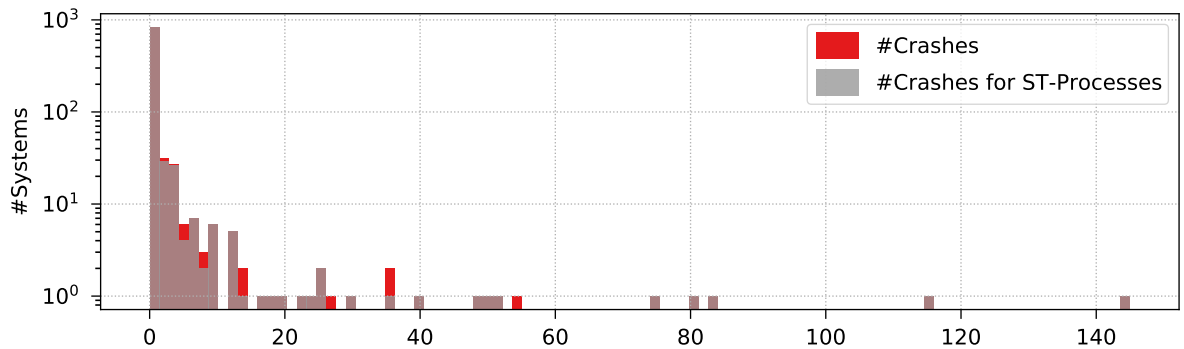
(k) Export November 2017: Histogram showing the distribution of 31662 crashes and the filtered 25872 crashes of processes with at least one software technology across 675 systems.



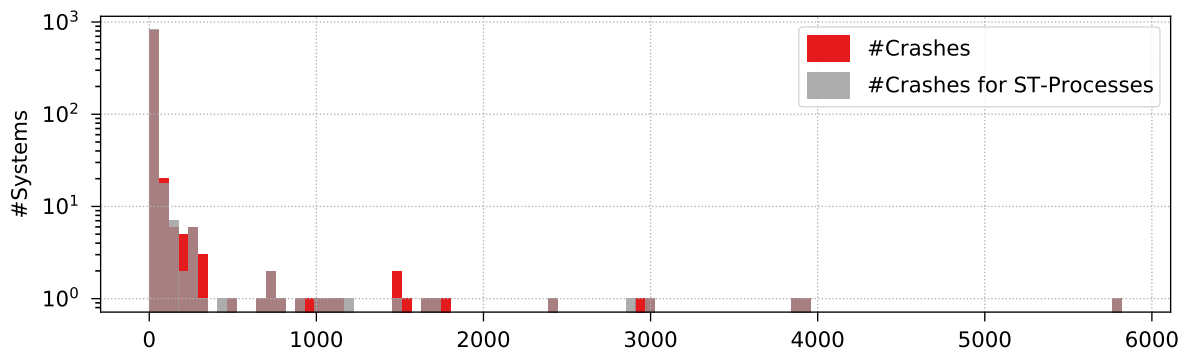
(l) Export December 2017: Histogram showing the distribution of 46648 crashes and the filtered 42056 crashes of processes with at least one software technology across 734 systems.



(m) Export January 2018: Histogram showing the distribution of 1160 crashes and the filtered 1015 crashes of processes with at least one software technology across 819 systems.



(n) Export February 2018: Histogram showing the distribution of 1482 crashes and the filtered 1314 crashes of processes with at least one software technology across 931 systems.

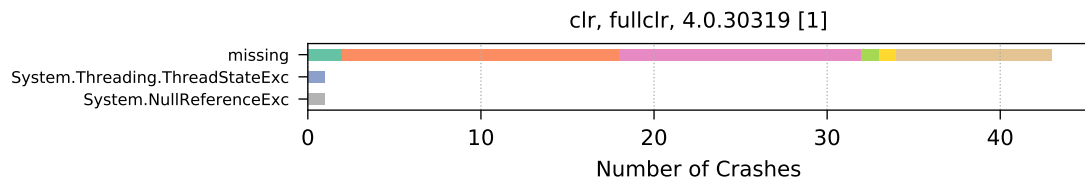


(o) Export March 2018: Histogram showing the distribution of 48421 crashes and the filtered 42882 crashes of processes with at least one software technology across 879 systems.

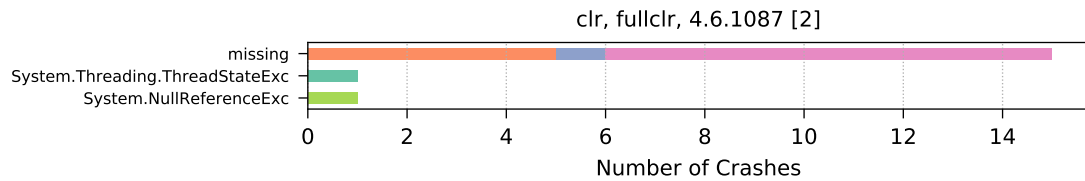
Figure B.2: Crash distribution histograms of the dataset exports in [Table 3.5 on p. 46](#), each with the number of systems displayed on a logarithmic scale. The # character represents the number of systems and crashes. *ST-Processes* are processes with at least one software technology.

B.2 Evaluation Results

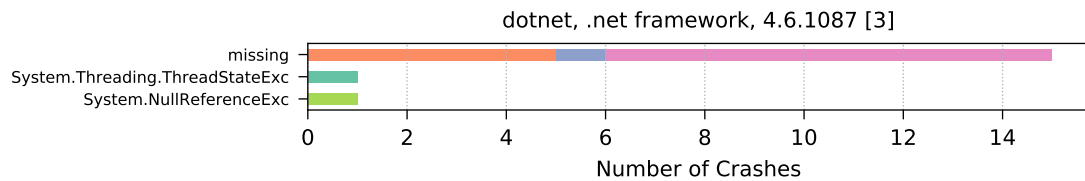
From [Figure B.3](#) to [Figure B.17](#), the five top-ranked 1-tuples and their crash groups for the class name property are shown for all our one-week exports of the 15 months. In each plot, up to a maximum of eleven crash groups are displayed (ten plus one for missing property entries), and all groups are shown, even if the crashes occurred only in a single system (in contrast to the plots in [Figure 3.5](#) on p. 50). Regarding the labels of the groups, *Exception* is abbreviated to *Exc*, and excessively long names are shortened by only showing the last few characters (e.g., the label name `...icationServices.CantStartSingleInstanceExc` in [Figure B.12c](#)) to provide more readable plots.



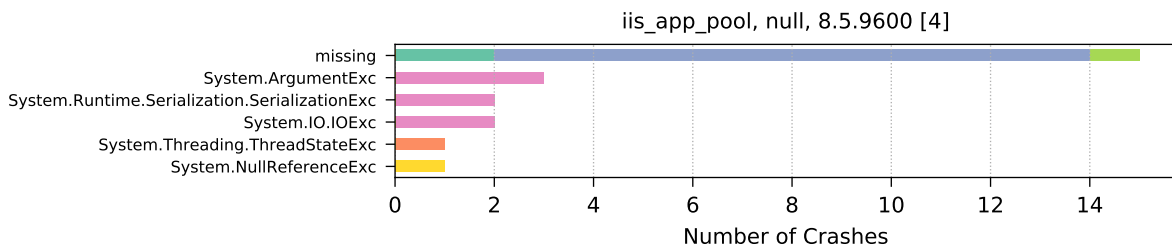
(a) Export January 2017: The 1st-ranked 1-tuple with a total of 45 crashes.



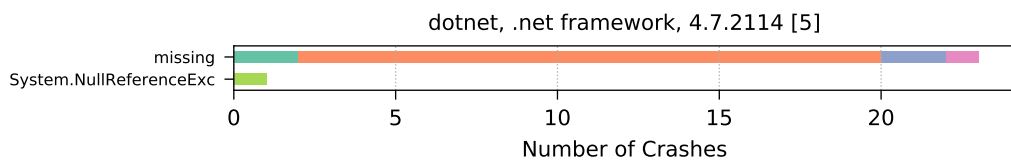
(b) Export January 2017: The 2nd-ranked 1-tuple with a total of 17 crashes.



(c) Export January 2017: The 3rd-ranked 1-tuple with a total of 17 crashes.

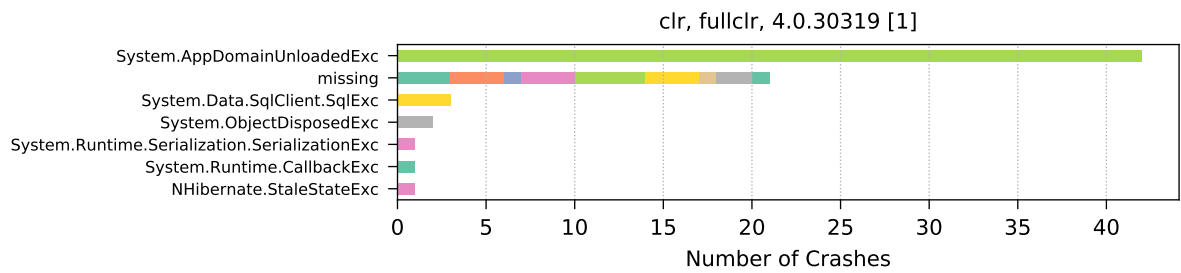


(d) Export January 2017: The 4th-ranked 1-tuple with a total of 24 crashes.

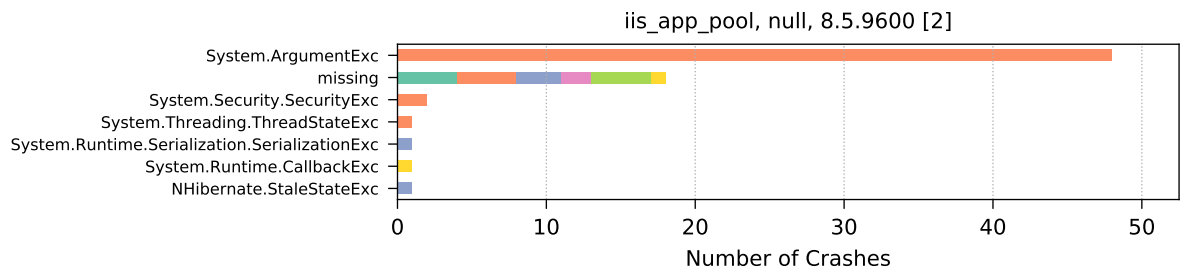


(e) Export January 2017: The 5th-ranked 1-tuple with a total of 24 crashes.

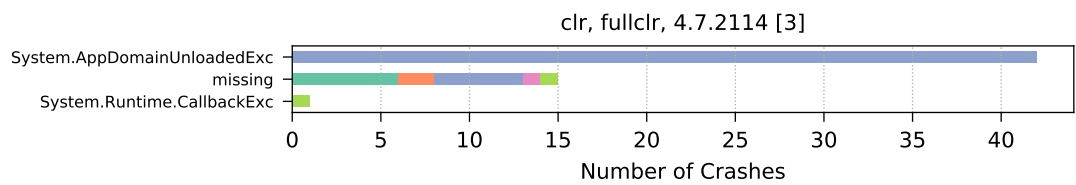
Figure B.3: Export January 2017: The top five 1-tuples for the class name crash property.



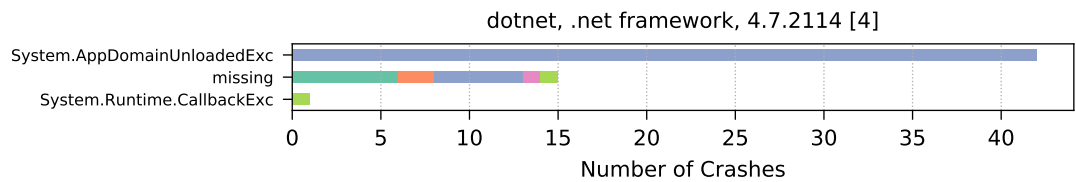
(a) Export February 2017: The 1st-ranked 1-tuple with a total of 71 crashes.



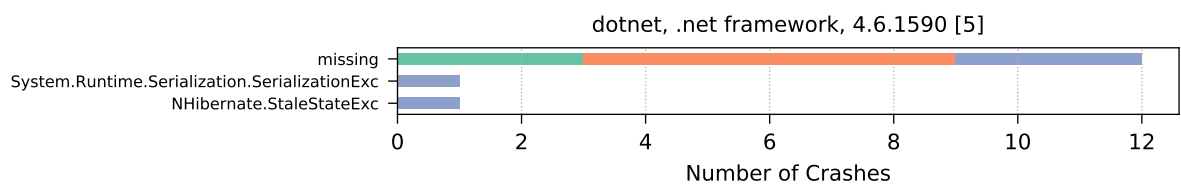
(b) Export February 2017: The 2nd-ranked 1-tuple with a total of 72 crashes.



(c) Export February 2017: The 3rd-ranked 1-tuple with a total of 58 crashes.

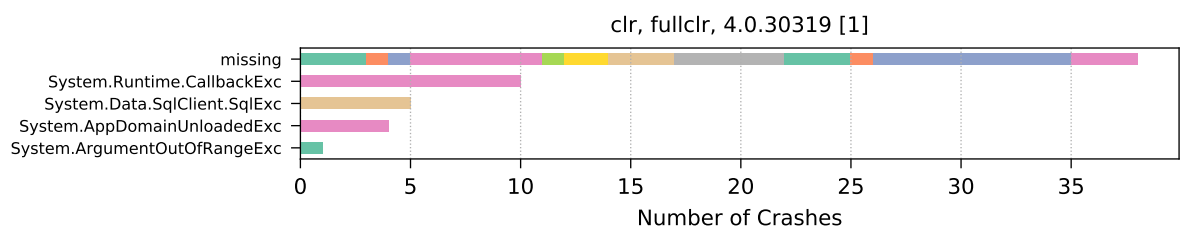


(d) Export February 2017: The 4th-ranked 1-tuple with a total of 58 crashes.

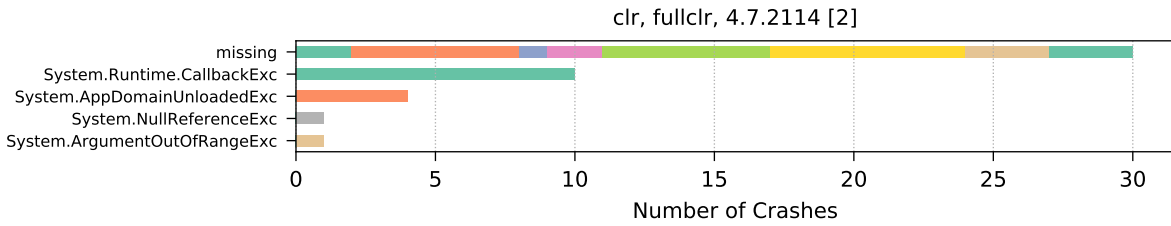


(e) Export February 2017: The 5th-ranked 1-tuple with a total of 14 crashes.

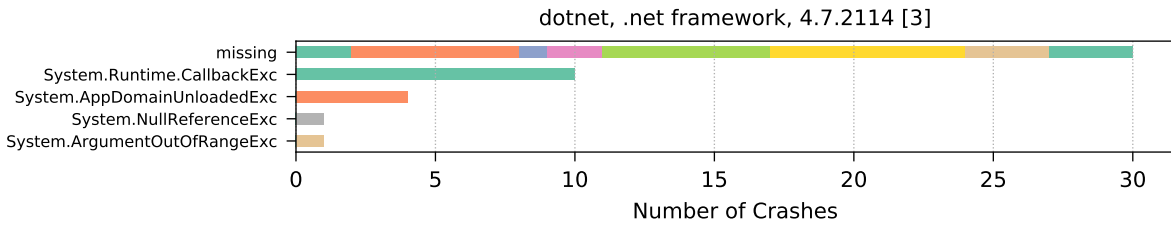
Figure B.4: Export February 2017: The top five 1-tuples for the class name crash property.



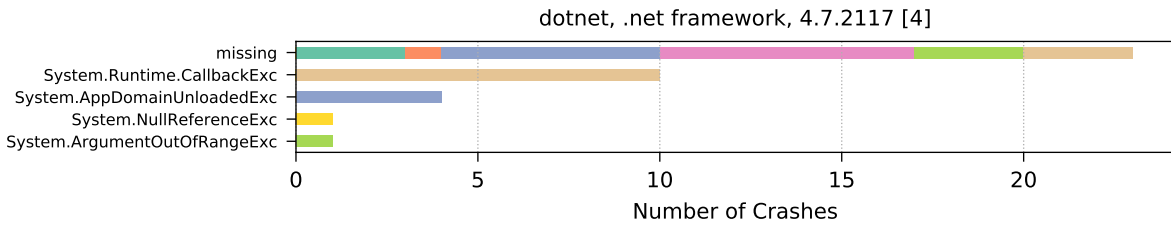
(a) Export March 2017: The 1st-ranked 1-tuple with a total of 58 crashes.



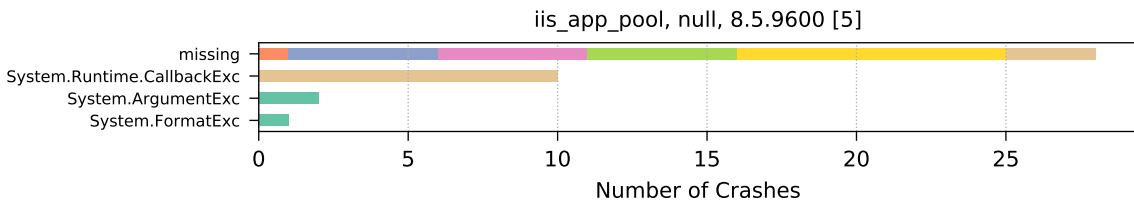
(b) Export March 2017: The 2nd-ranked 1-tuple with a total of 46 crashes.



(c) Export March 2017: The 3rd-ranked 1-tuple with a total of 46 crashes.

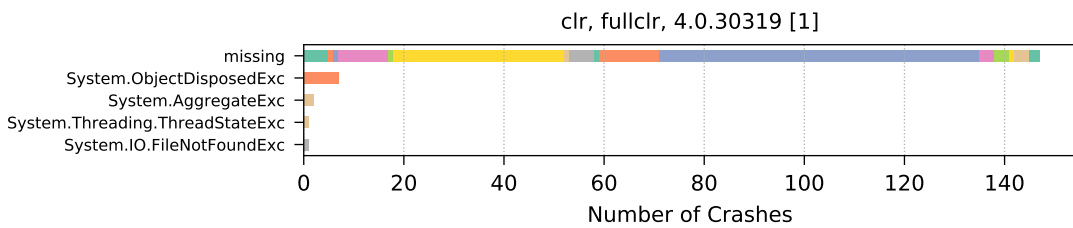


(d) Export March 2017: The 4th-ranked 1-tuple with a total of 39 crashes.

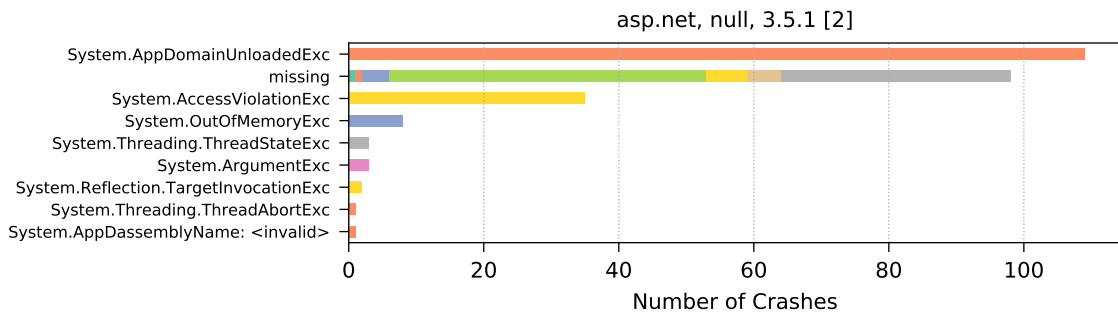


(e) Export March 2017: The 5th-ranked 1-tuple with a total of 41 crashes.

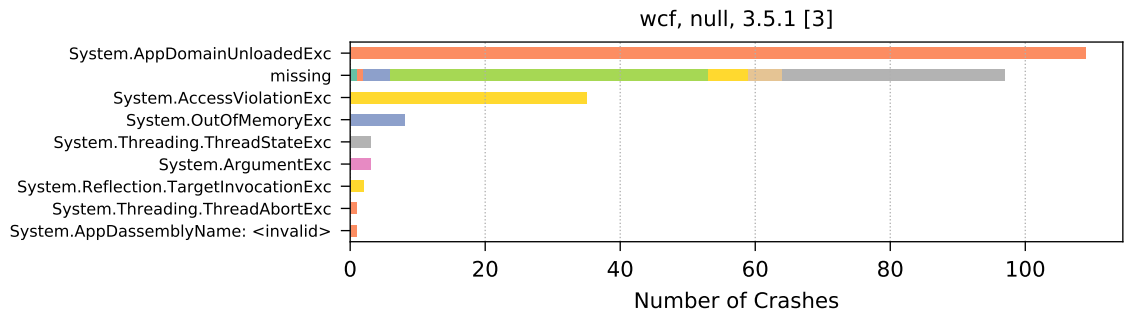
Figure B.5: Export March 2017: The top five 1-tuples for the class name crash property.



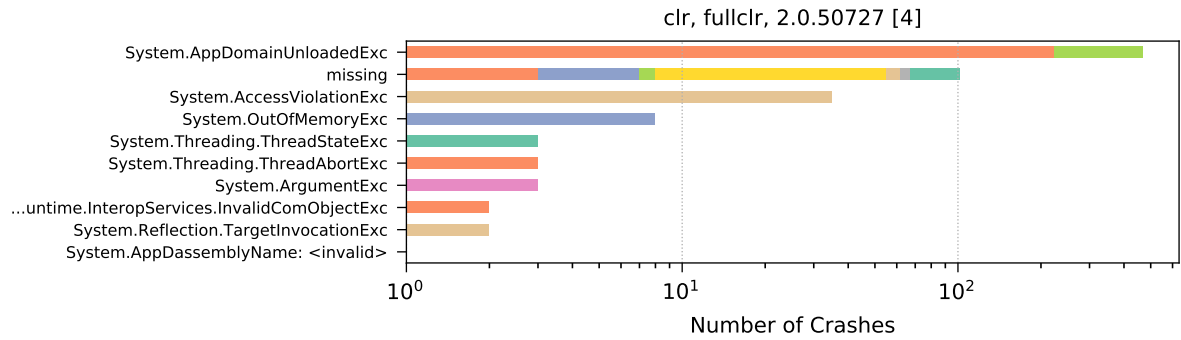
(a) Export April 2017: The 1st-ranked 1-tuple with a total of 158 crashes.



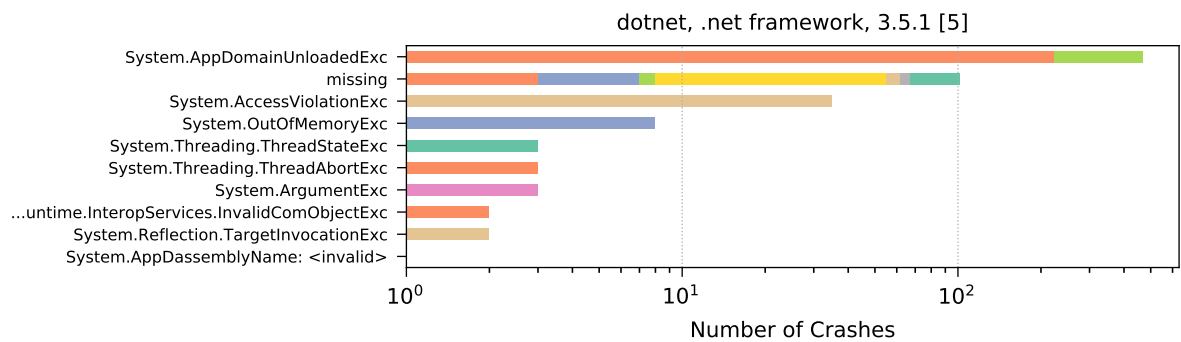
(b) Export April 2017: The 2nd-ranked 1-tuple with a total of 260 crashes.



(c) Export April 2017: The 3rd-ranked 1-tuple with a total of 259 crashes.

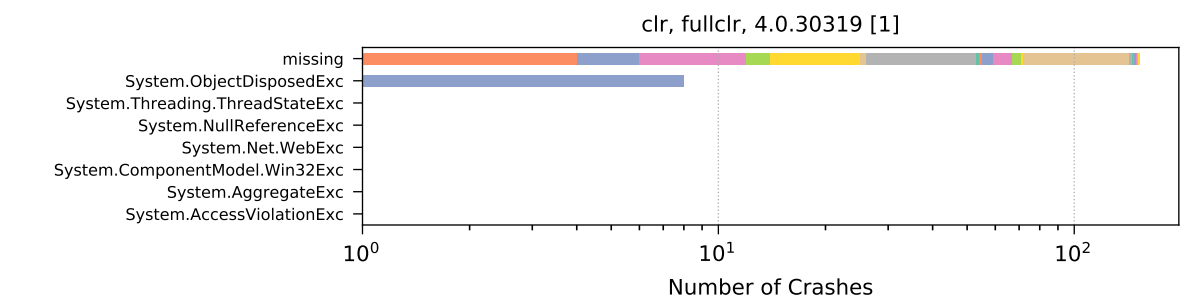


(d) Export April 2017: The 4th-ranked 1-tuple with a total of 624 crashes.

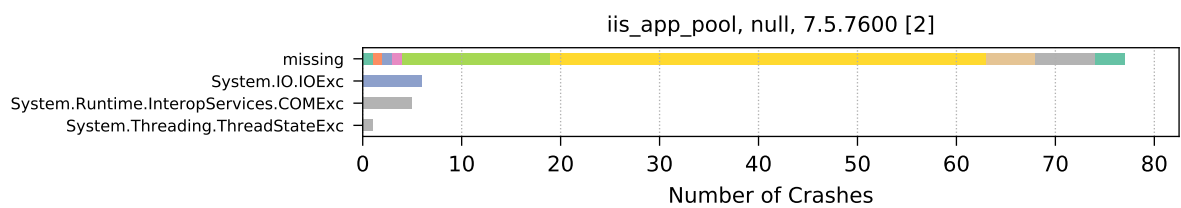


(e) Export April 2017: The 5th-ranked 1-tuple with a total of 624 crashes.

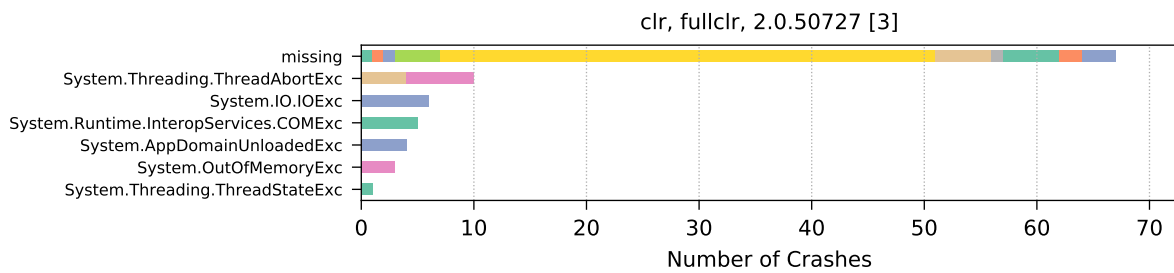
Figure B.6: Export April 2017: The top five 1-tuples for the class name crash property.



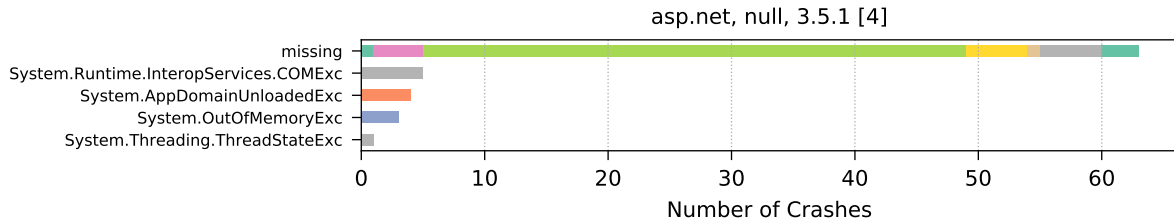
(a) Export May 2017: The 1st-ranked 1-tuple with a total of 167 crashes.



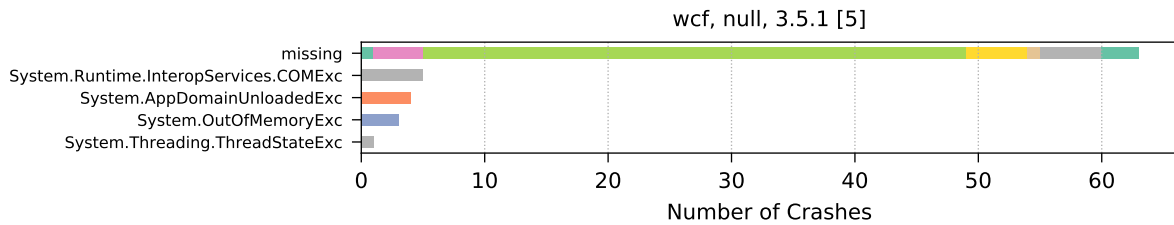
(b) Export May 2017: The 2nd-ranked 1-tuple with a total of 89 crashes.



(c) Export May 2017: The 3rd-ranked 1-tuple with a total of 96 crashes.

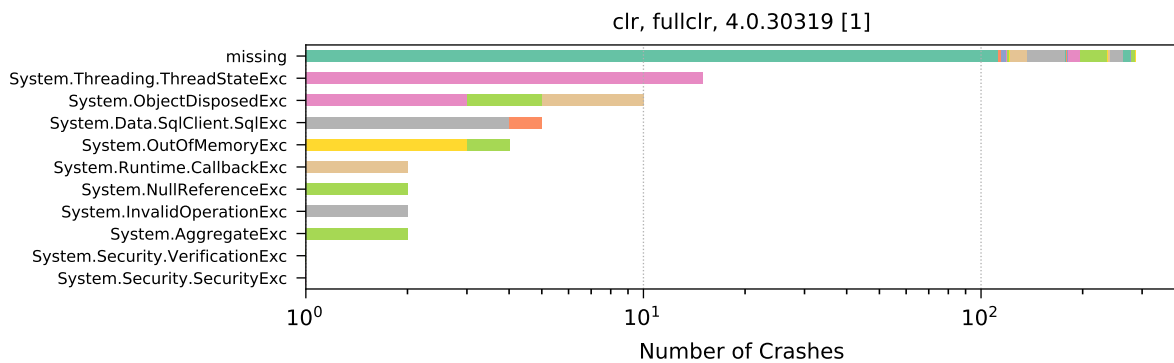


(d) Export May 2017: The 4th-ranked 1-tuple with a total of 76 crashes.

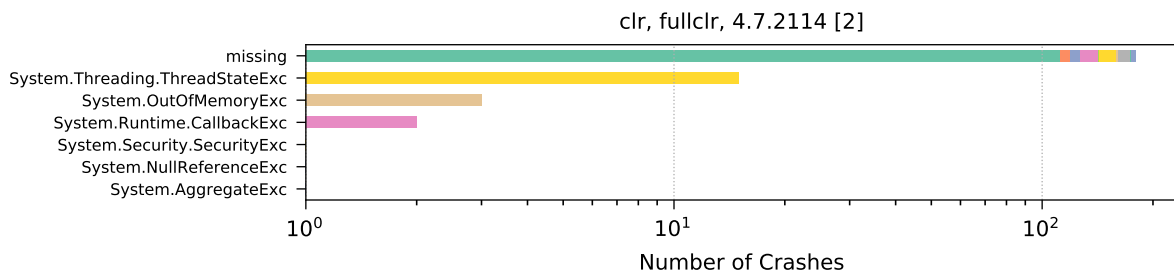


(e) Export May 2017: The 5th-ranked 1-tuple with a total of 76 crashes.

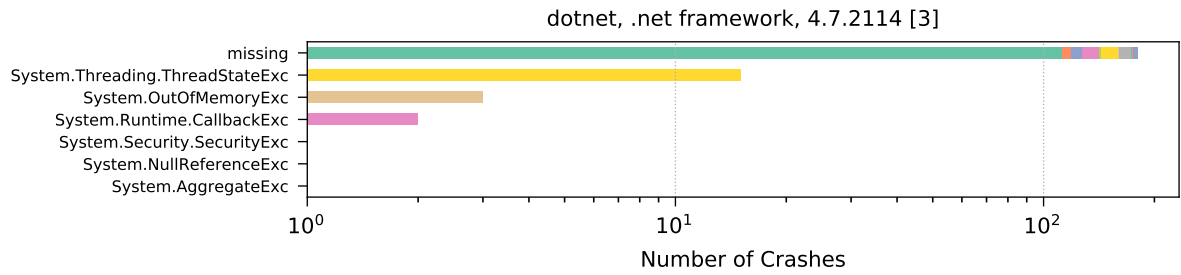
Figure B.7: Export May 2017: The top five 1-tuples for the class name crash property.



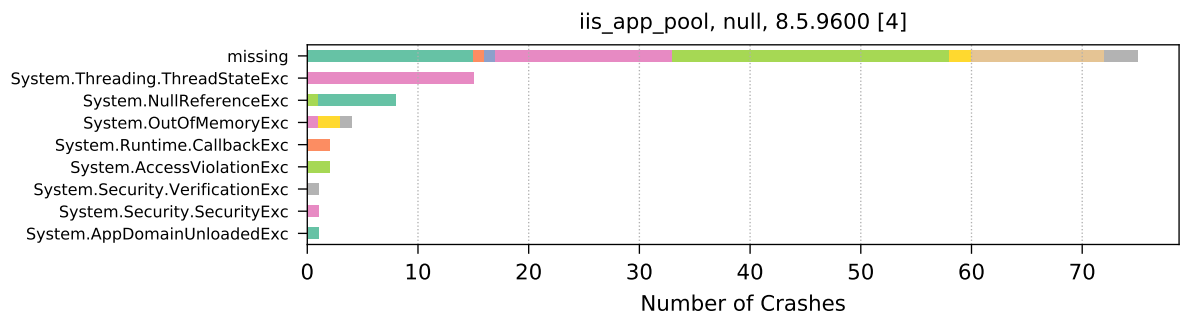
(a) Export June 2017: The 1st-ranked 1-tuple with a total of 332 crashes.



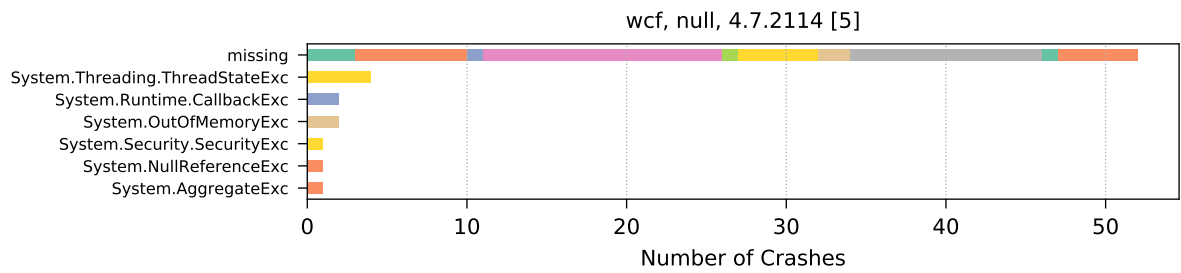
(b) Export June 2017: The 2nd-ranked 1-tuple with a total of 203 crashes.



(c) Export June 2017: The 3rd-ranked 1-tuple with a total of 203 crashes.

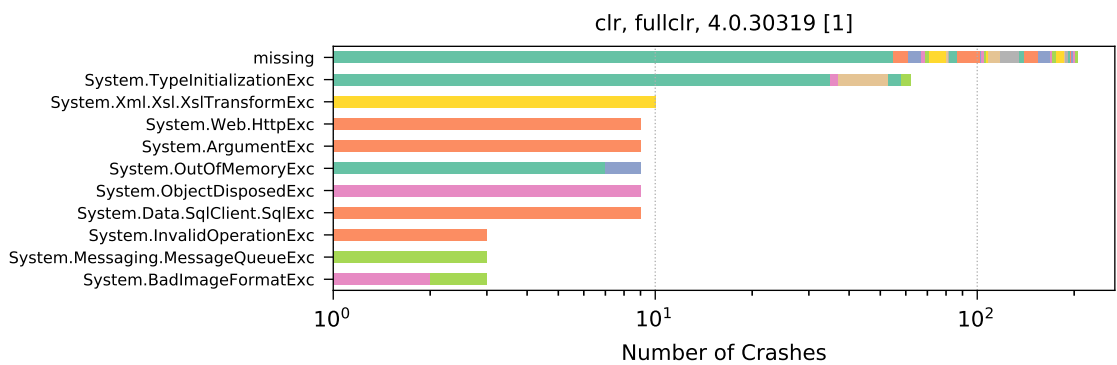


(d) Export June 2017: The 4th-ranked 1-tuple with a total of 109 crashes.

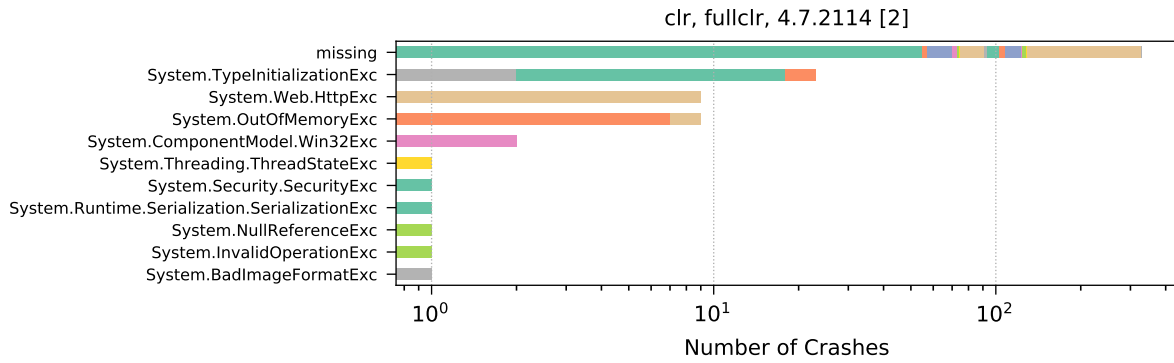


(e) Export June 2017: The 5th-ranked 1-tuple with a total of 63 crashes.

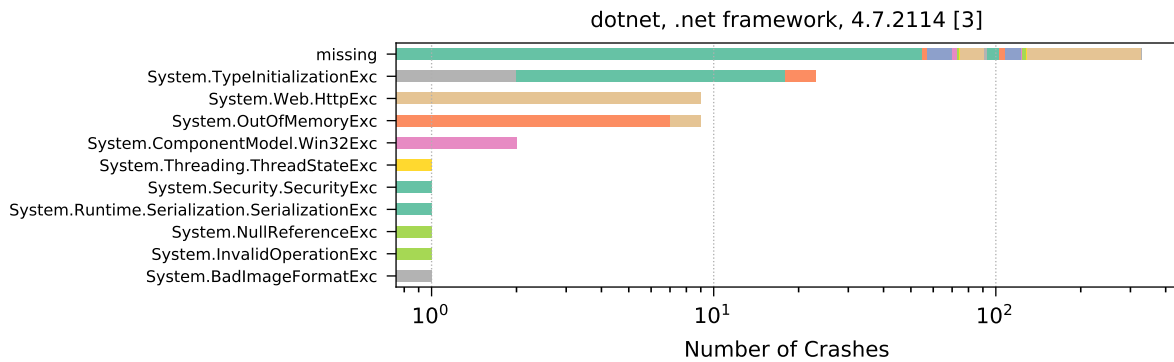
Figure B.8: Export June 2017: The top five 1-tuples for the class name crash property.



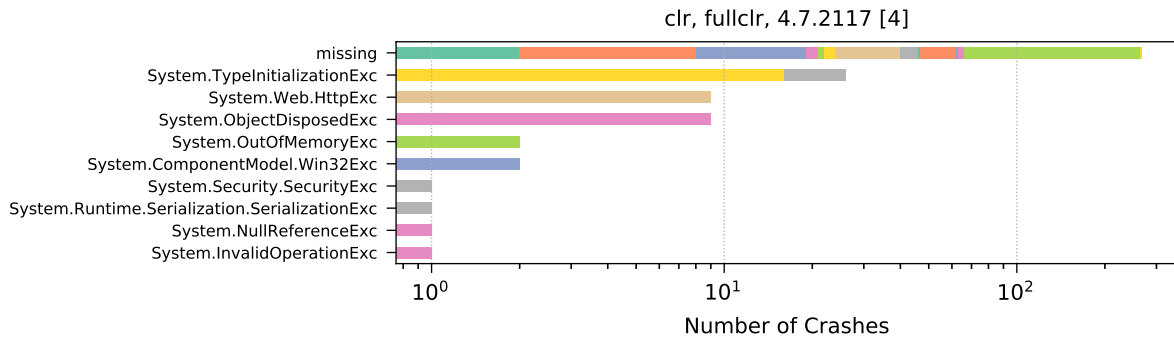
(a) Export July 2017: The 1st-ranked 1-tuple with a total of 331 crashes.



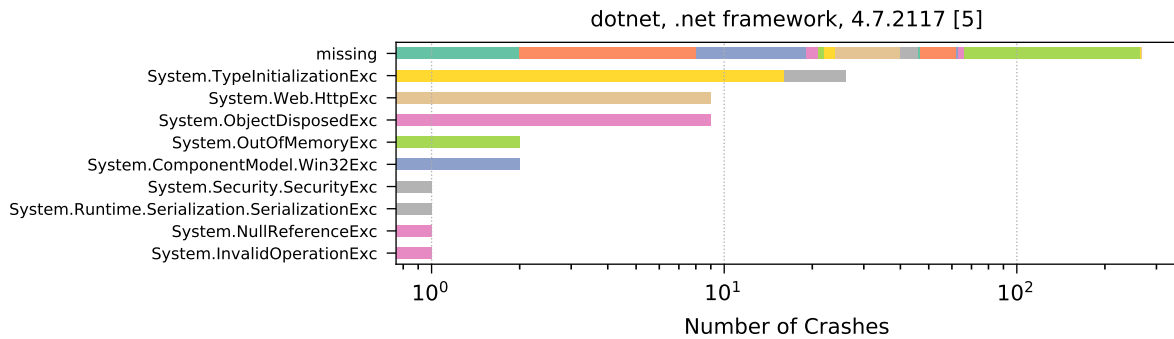
(b) Export July 2017: The 2nd-ranked 1-tuple with a total of 379 crashes.



(c) Export July 2017: The 3rd-ranked 1-tuple with a total of 379 crashes.

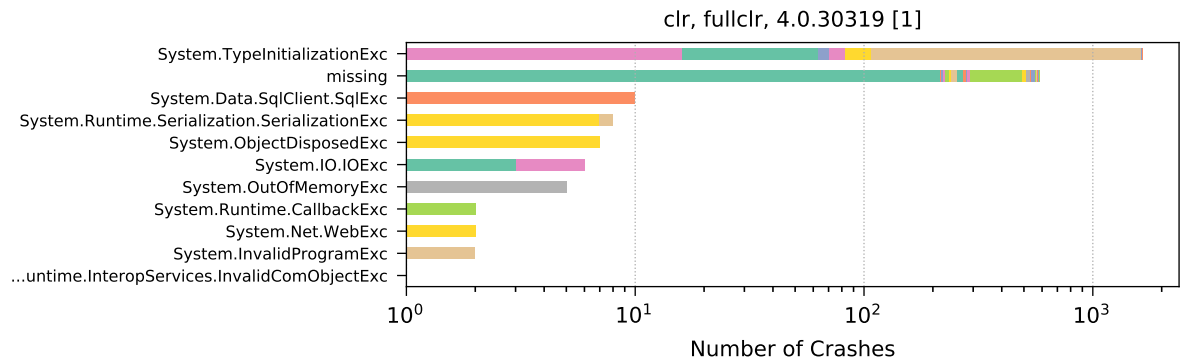


(d) Export July 2017: The 4th-ranked 1-tuple with a total of 320 crashes.

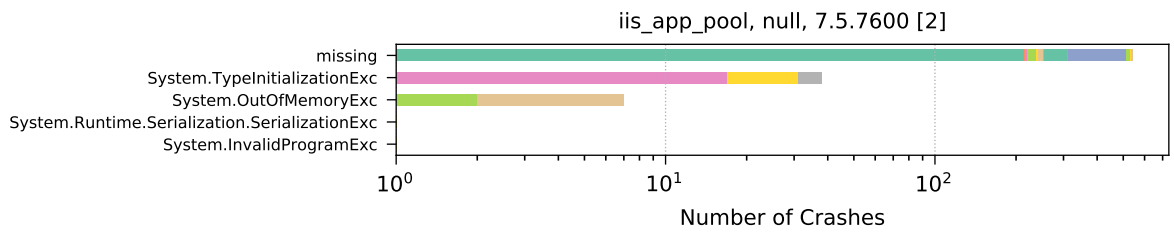


(e) Export July 2017: The 5th-ranked 1-tuple with a total of 320 crashes.

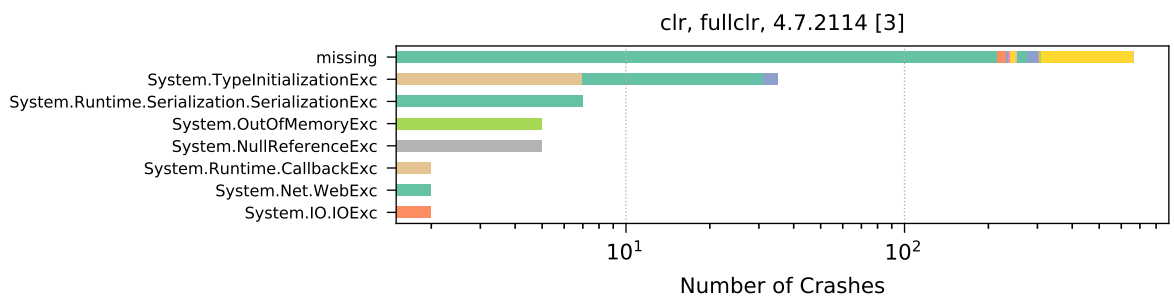
Figure B.9: Export July 2017: The top five 1-tuples for the class name crash property.



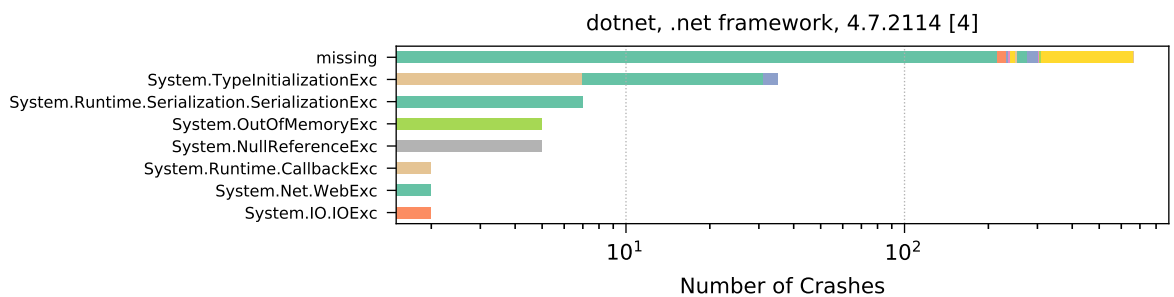
(a) Export August 2017: The 1st-ranked 1-tuple with a total of 2274 crashes.



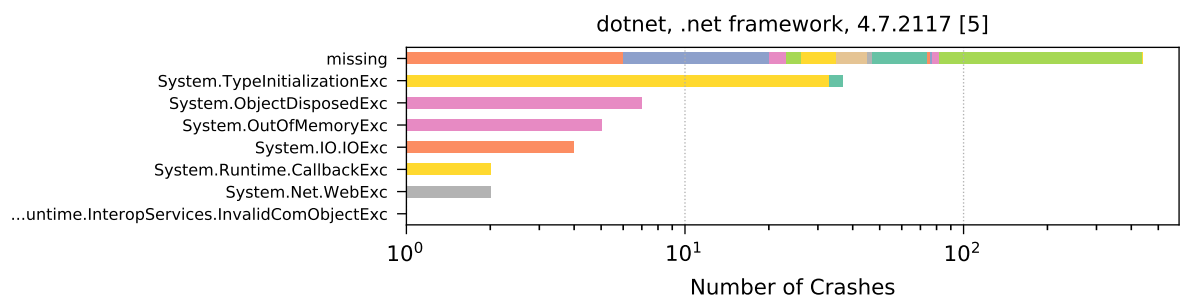
(b) Export August 2017: The 2nd-ranked 1-tuple with a total of 586 crashes.



(c) Export August 2017: The 3rd-ranked 1-tuple with a total of 723 crashes.

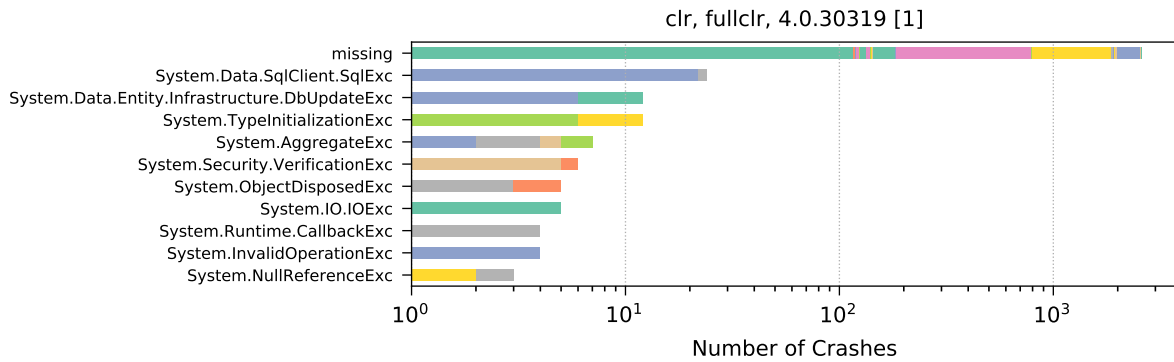


(d) Export August 2017: The 4th-ranked 1-tuple with a total of 723 crashes.

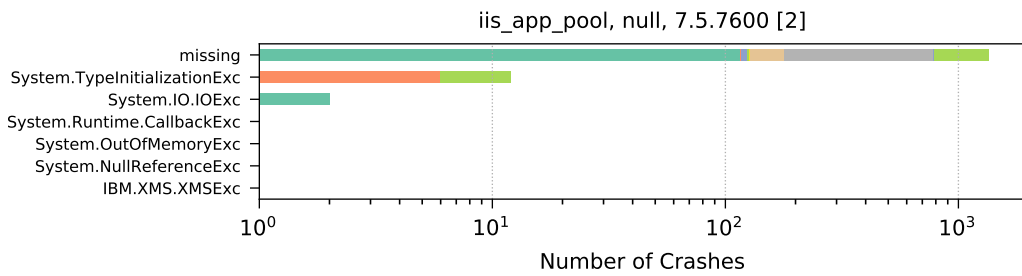


(e) Export August 2017: The 5th-ranked 1-tuple with a total of 496 crashes.

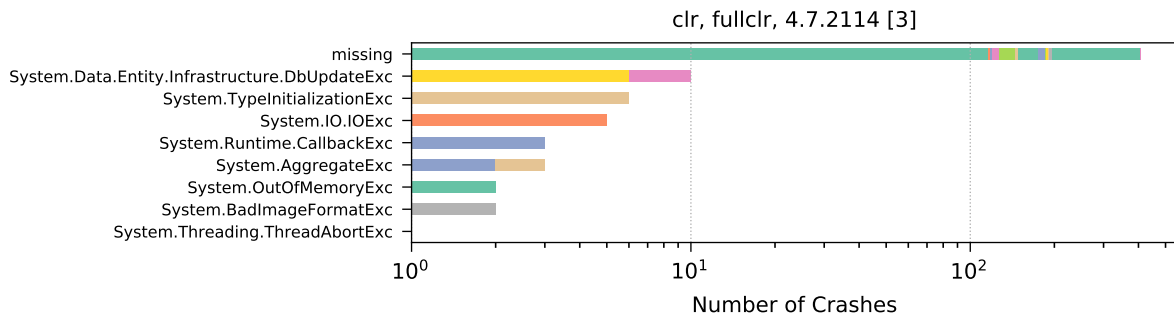
Figure B.10: Export August 2017: The top five 1-tuples for the class name crash property.



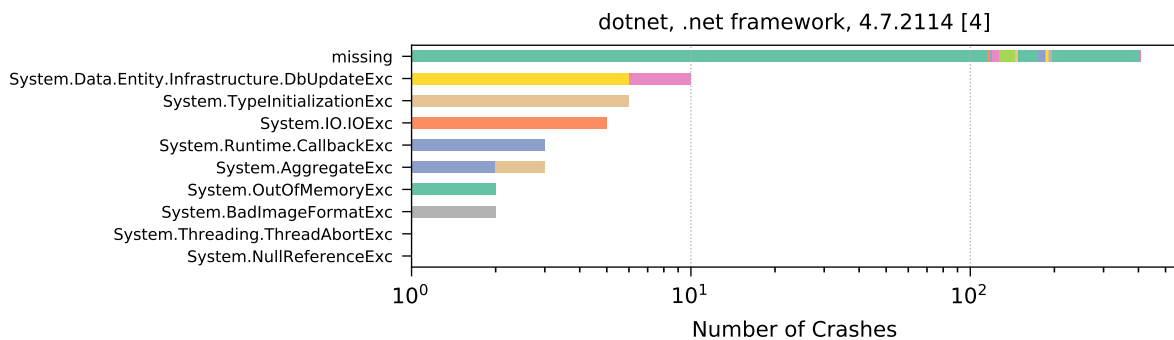
(a) Export September 2017: The 1st-ranked 1-tuple with a total of 2656 crashes.



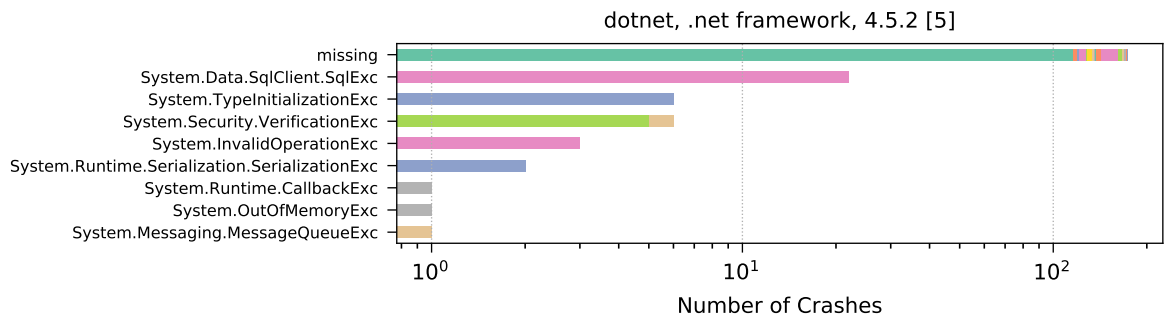
(b) Export September 2017: The 2nd-ranked 1-tuple with a total of 1370 crashes.



(c) Export September 2017: The 3rd-ranked 1-tuple with a total of 441 crashes.

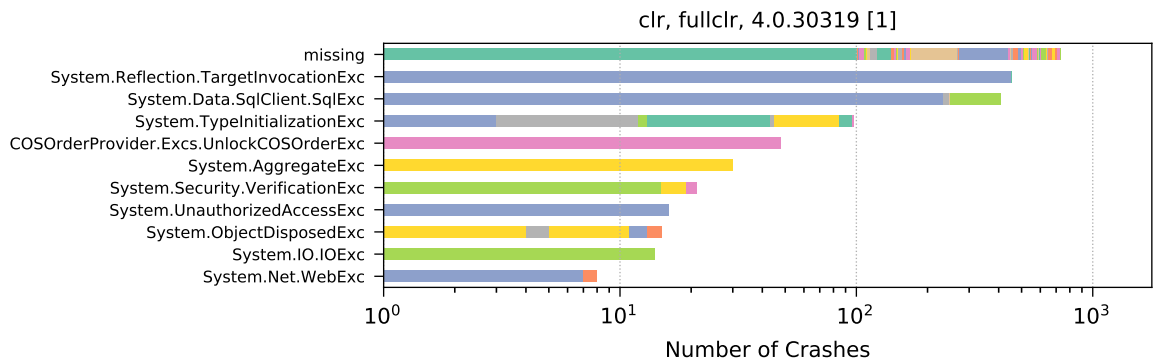


(d) Export September 2017: The 4th-ranked 1-tuple with a total of 442 crashes.

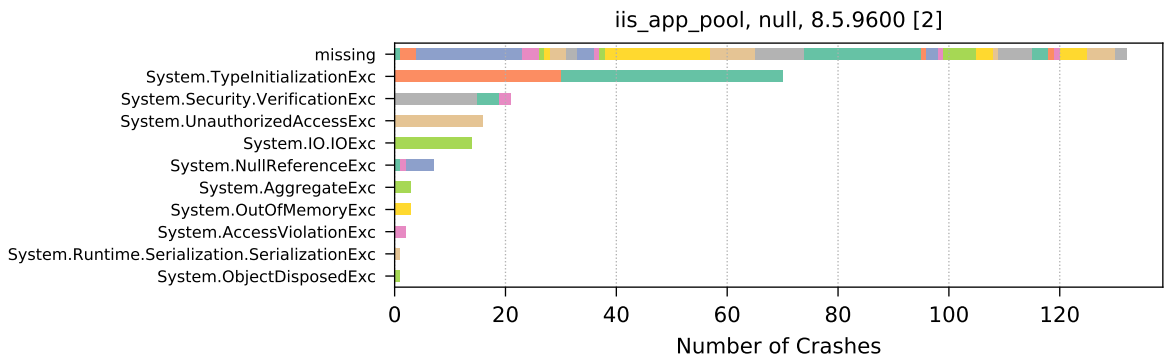


(e) Export September 2017: The 5th-ranked 1-tuple with a total of 216 crashes.

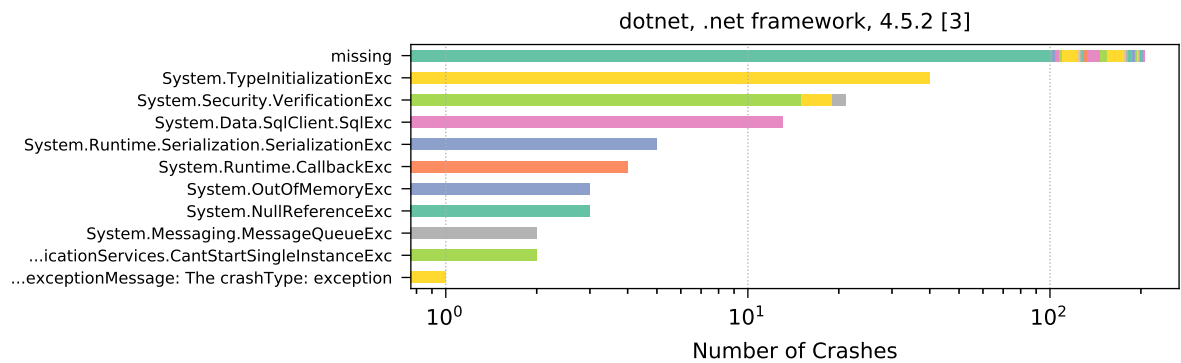
Figure B.11: Export September 2017: The top five 1-tuples for the class name crash property.



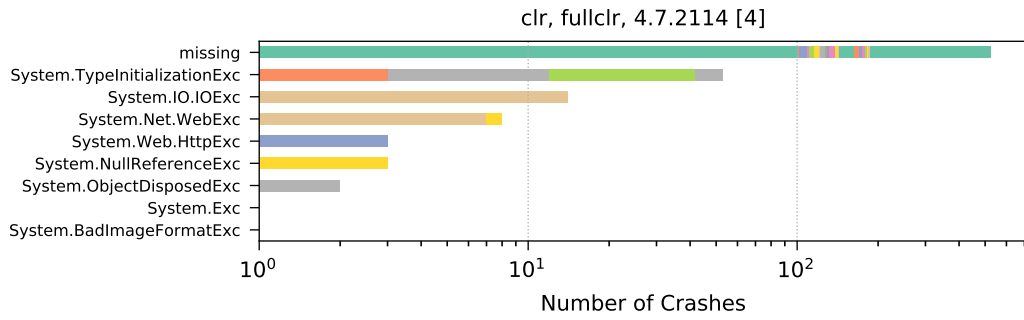
(a) Export October 2017: The 1st-ranked 1-tuple with a total of 1841 crashes.



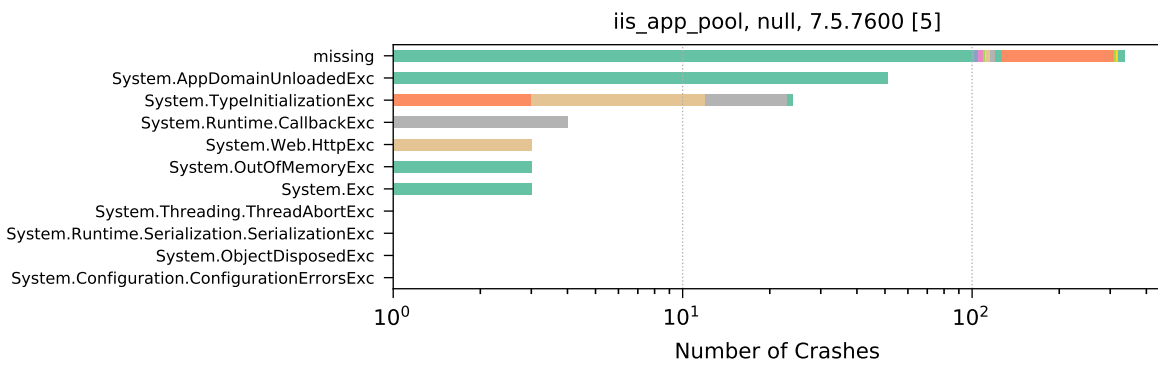
(b) Export October 2017: The 2nd-ranked 1-tuple with a total of 270 crashes.



(c) Export October 2017: The 3rd-ranked 1-tuple with a total of 299 crashes.

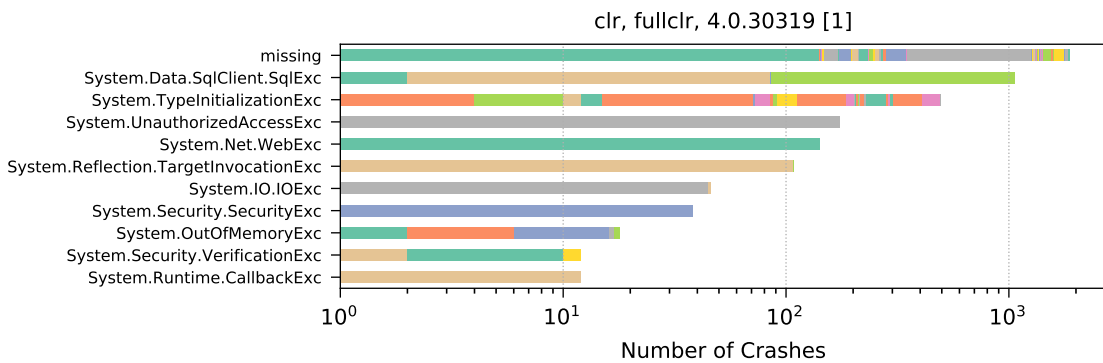


(d) Export October 2017: The 4th-ranked 1-tuple with a total of 610 crashes.

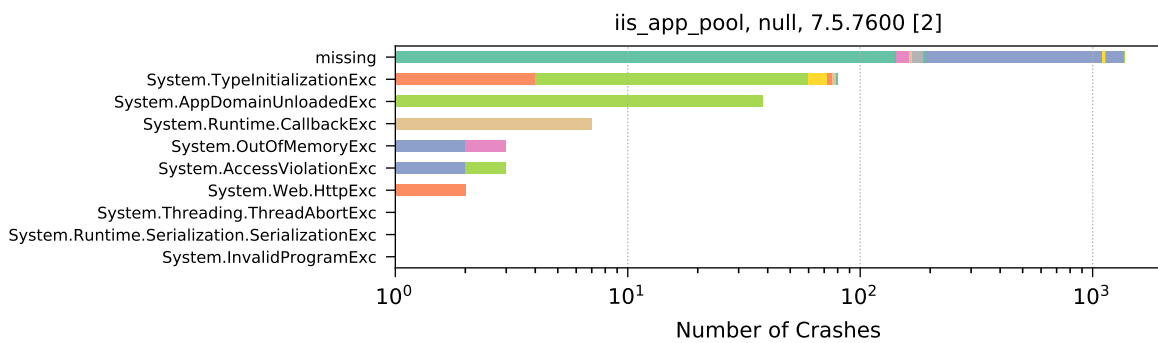


(e) Export October 2017: The 5th-ranked 1-tuple with a total of 429 crashes.

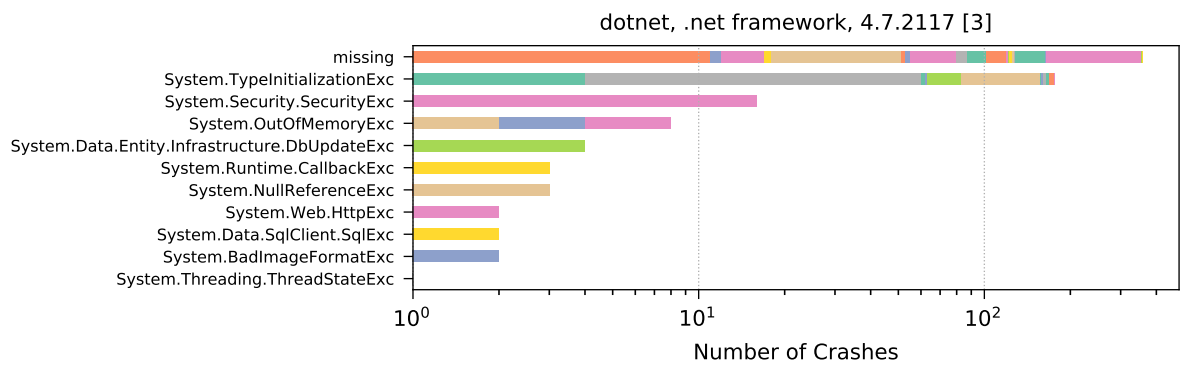
Figure B.12: Export October 2017: The top five 1-tuples for the class name crash property.



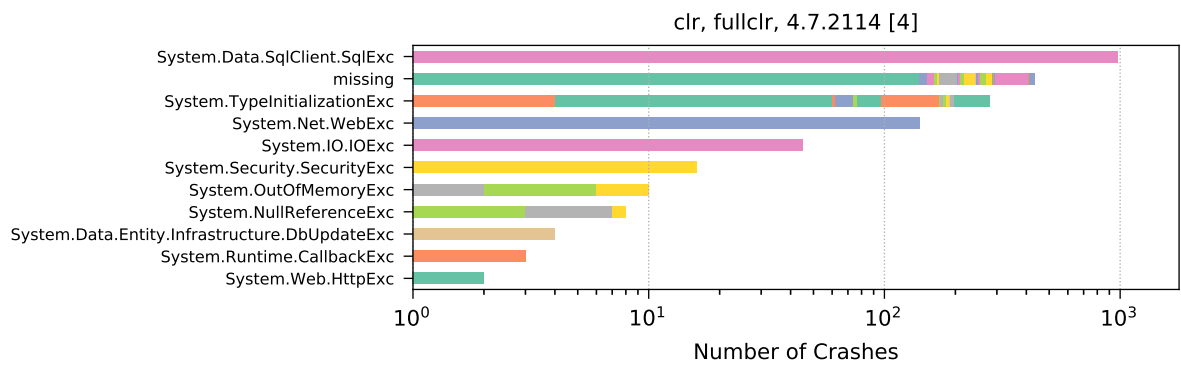
(a) Export November 2017: The 1st-ranked 1-tuple with a total of 3994 crashes.



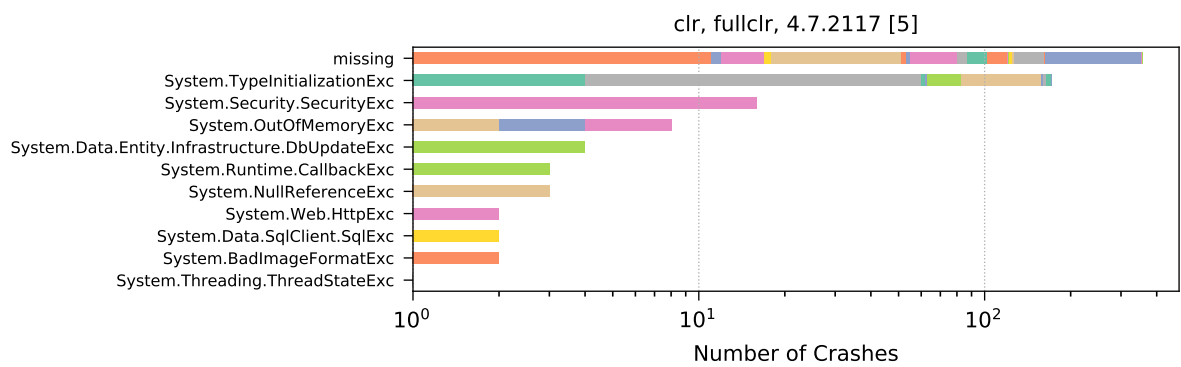
(b) Export November 2017: The 2nd-ranked 1-tuple with a total of 1513 crashes.



(c) Export November 2017: The 3rd-ranked 1-tuple with a total of 576 crashes.

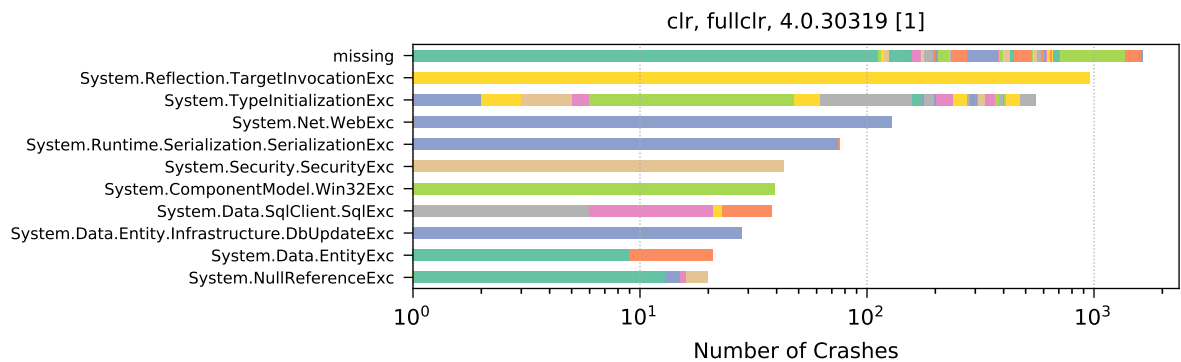


(d) Export November 2017: The 4th-ranked 1-tuple with a total of 1924 crashes.

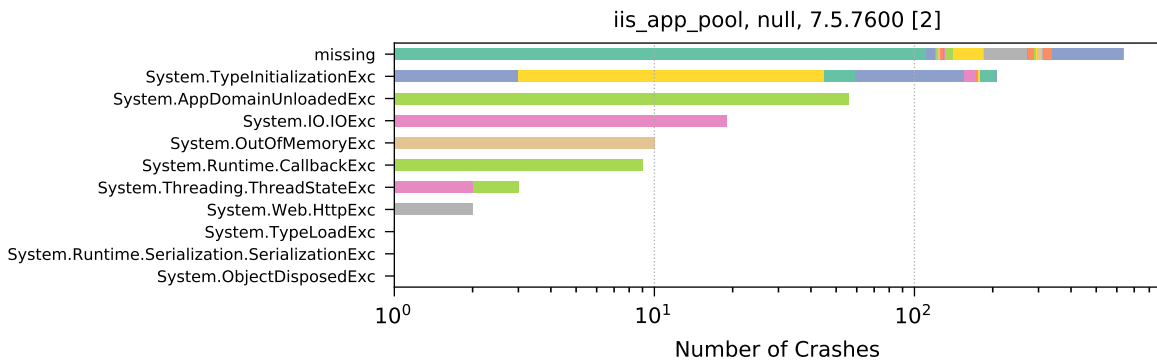


(e) Export November 2017: The 5th-ranked 1-tuple with a total of 570 crashes.

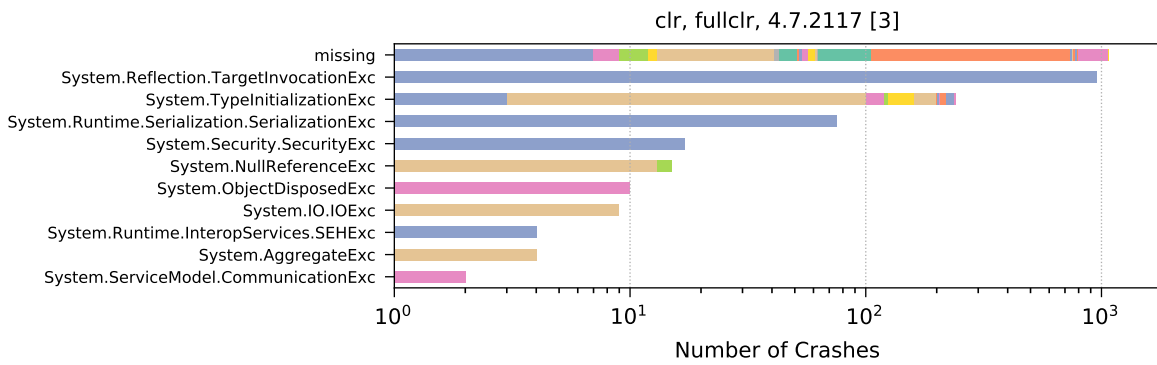
Figure B.13: Export November 2017: The top five 1-tuples for the class name crash property.



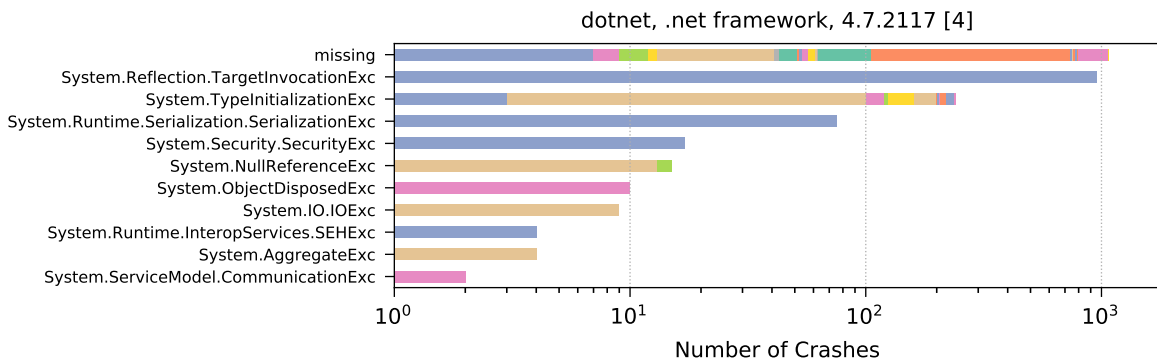
(a) Export December 2017: The 1st-ranked 1-tuple with a total of 3540 crashes.



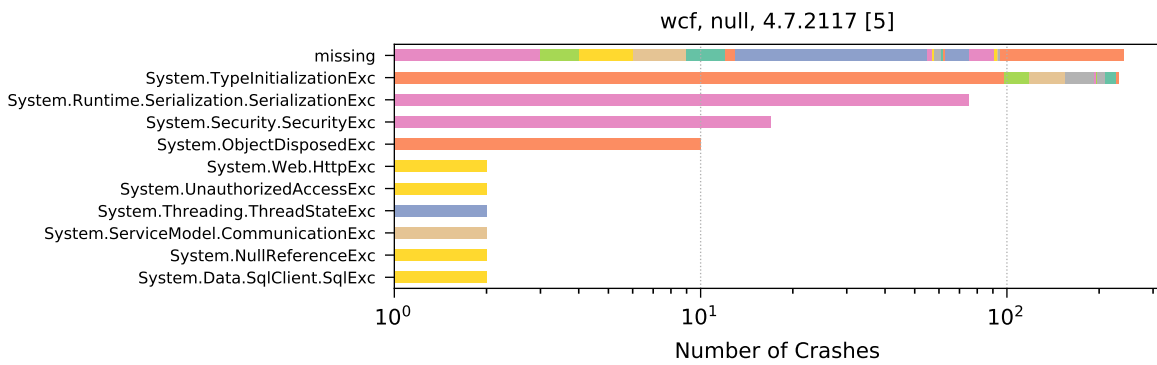
(b) Export December 2017: The 2nd-ranked 1-tuple with a total of 947 crashes.



(c) Export December 2017: The 3rd-ranked 1-tuple with a total of 2409 crashes.

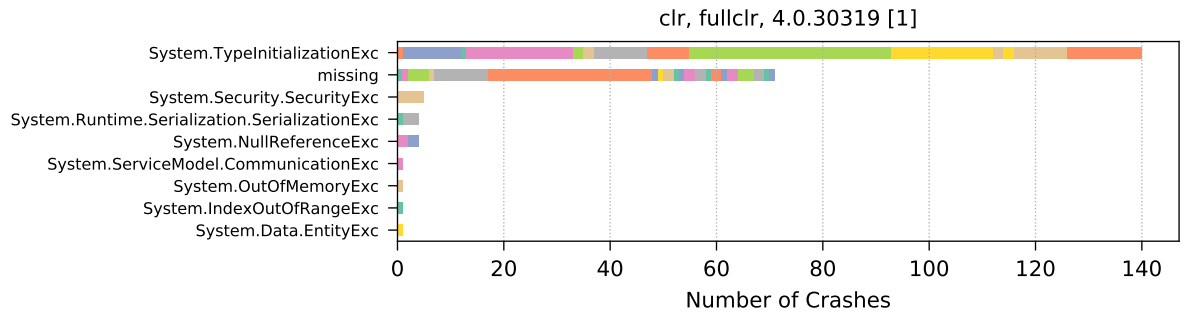


(d) Export December 2017: The 4th-ranked 1-tuple with a total of 2409 crashes.

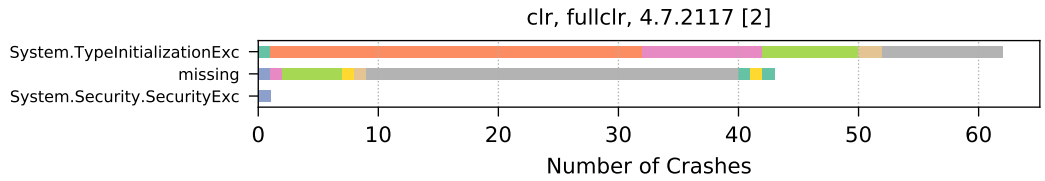


(e) Export December 2017: The 5th-ranked 1-tuple with a total of 586 crashes.

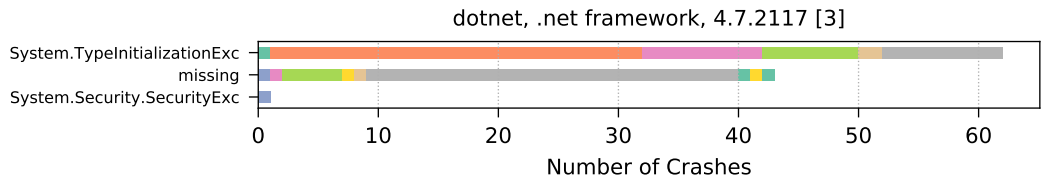
Figure B.14: Export December 2017: The top five 1-tuples for the class name crash property.



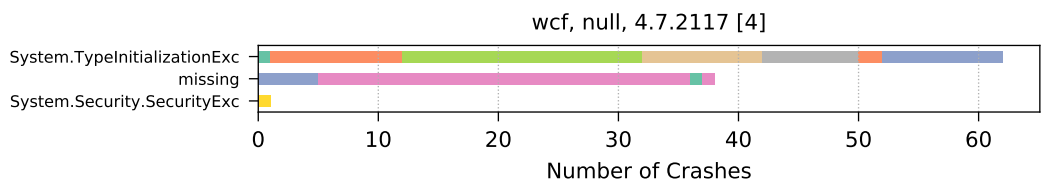
(a) Export January 2018: The 1st-ranked 1-tuple with a total of 228 crashes.



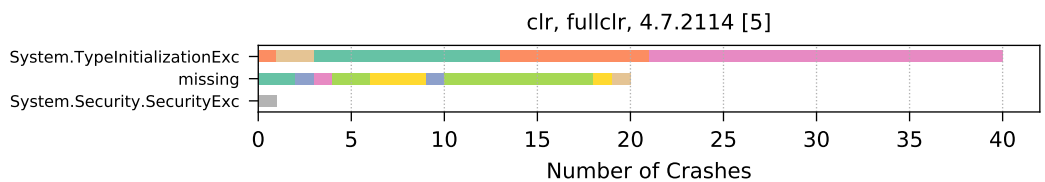
(b) Export January 2018: The 2nd-ranked 1-tuple with a total of 106 crashes.



(c) Export January 2018: The 3rd-ranked 1-tuple with a total of 106 crashes.

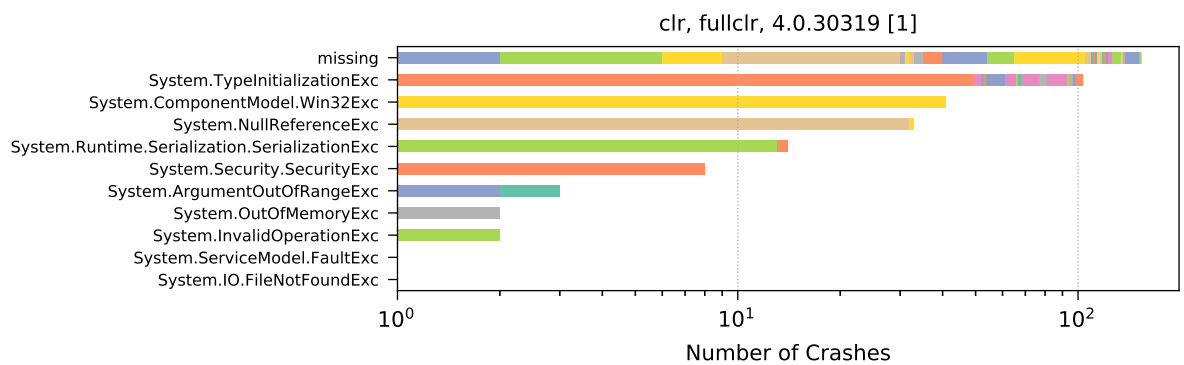


(d) Export January 2018: The 4th-ranked 1-tuple with a total of 101 crashes.

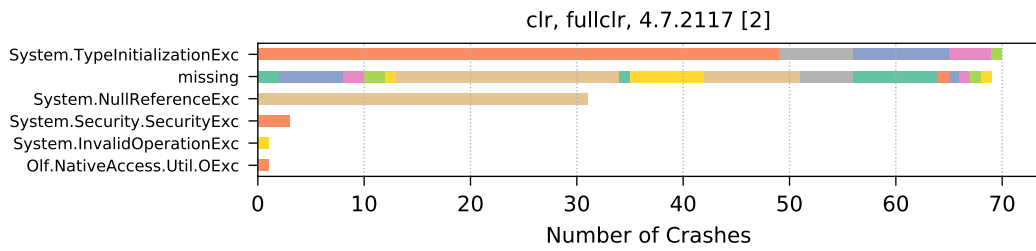


(e) Export January 2018: The 5th-ranked 1-tuple with a total of 61 crashes.

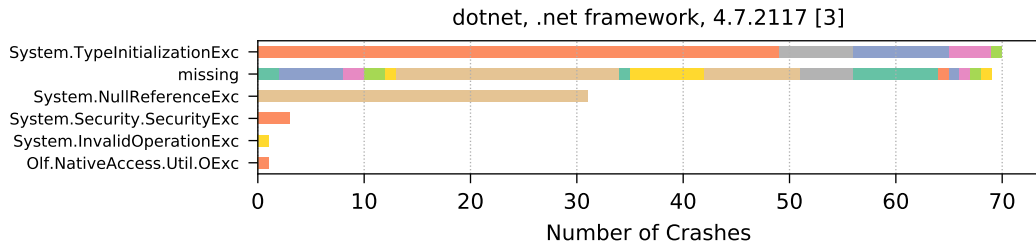
Figure B.15: Export January 2018: The top five 1-tuples for the class name crash property.



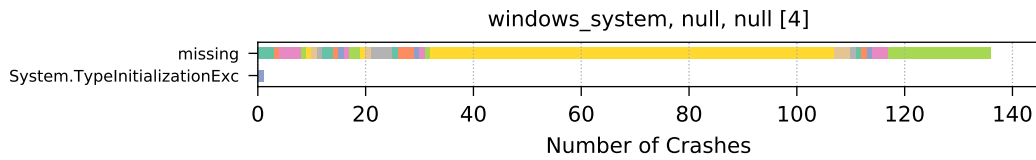
(a) Export February 2018: The 1st-ranked 1-tuple with a total of 362 crashes.



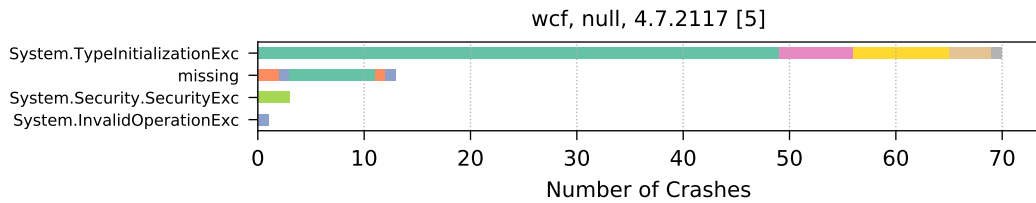
(b) Export February 2018: The 2nd-ranked 1-tuple with a total of 175 crashes.



(c) Export February 2018: The 3rd-ranked 1-tuple with a total of 175 crashes.

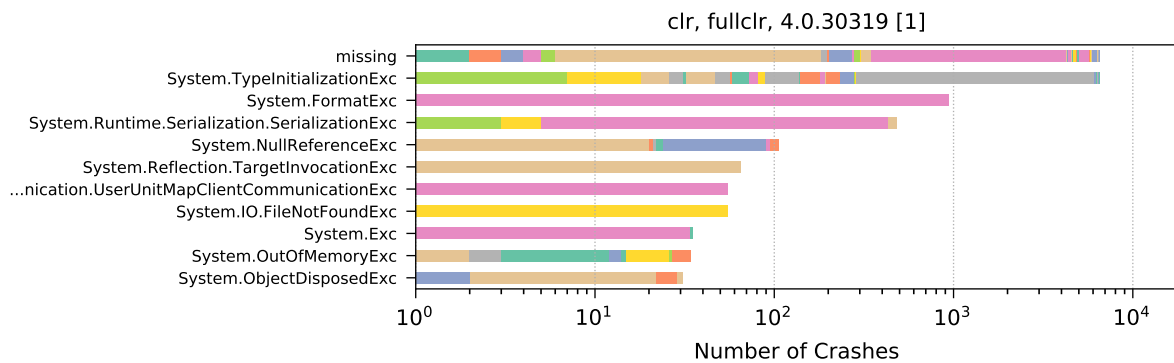


(d) Export February 2018: The 4th-ranked 1-tuple with a total of 137 crashes.

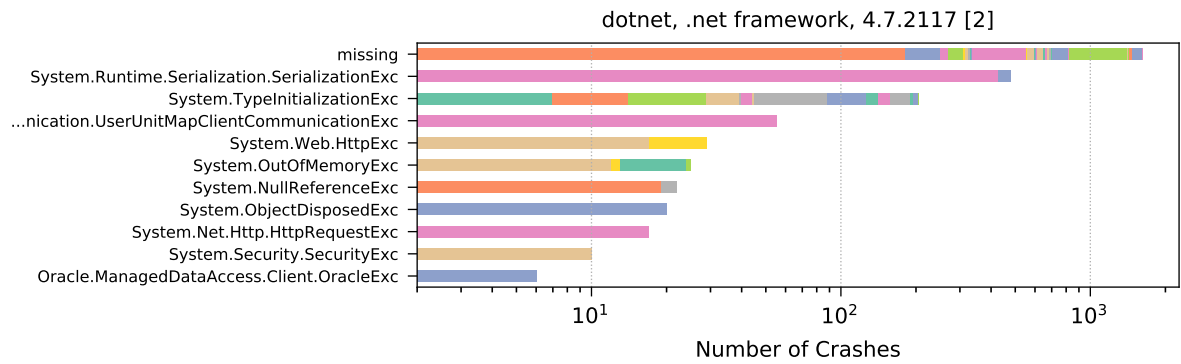


(e) Export February 2018: The 5th-ranked 1-tuple with a total of 87 crashes.

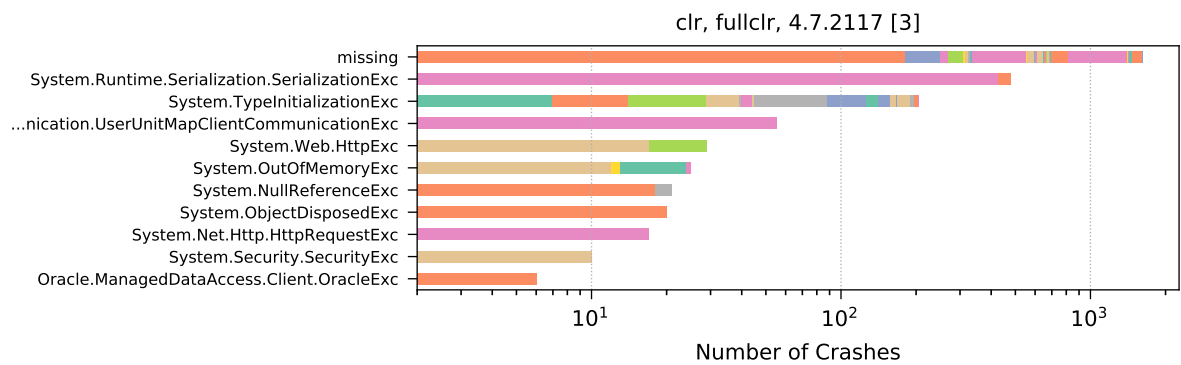
Figure B.16: Export February 2018: The top five 1-tuples for the class name crash property.



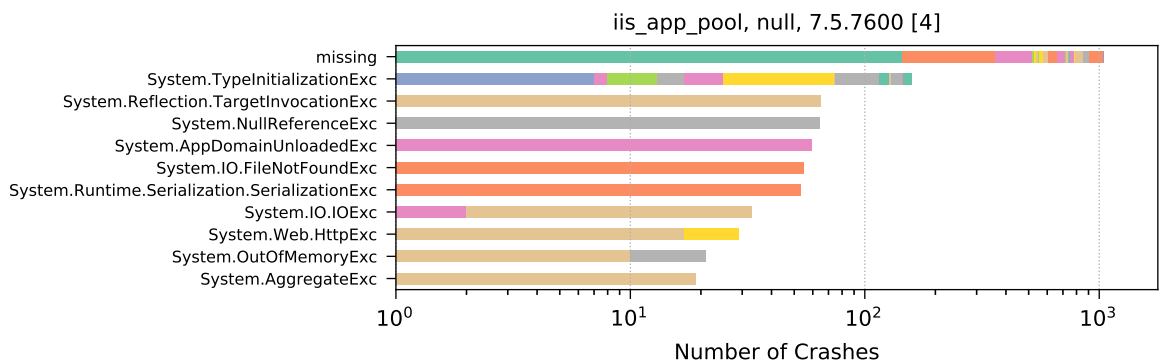
(a) Export March 2018: The 1st-ranked 1-tuple with a total of 14848 crashes.



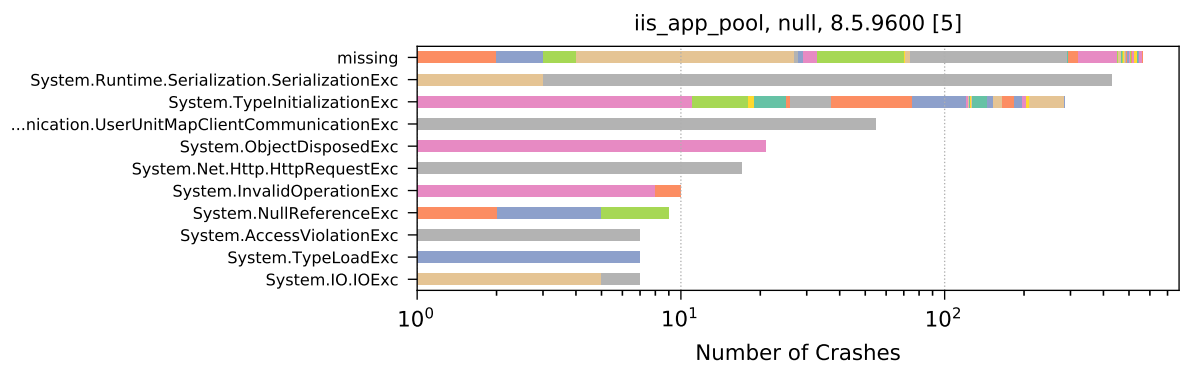
(b) Export March 2018: The 2nd-ranked 1-tuple with a total of 2491 crashes.



(c) Export March 2018: The 3rd-ranked 1-tuple with a total of 2488 crashes.



(d) Export March 2018: The 4th-ranked 1-tuple with a total of 1604 crashes.



(e) Export March 2018: The 5th-ranked 1-tuple with a total of 1413 crashes.

Figure B.17: Export March 2018: The top five 1-tuples for the class name crash property.

Appendix C

Time-Series-based Event Prediction

In this appendix chapter, we provide additional results and figures that allow more detailed insights into the data, the approach and the evaluation presented in [Chapter 4 on p. 63](#).

C.1 Data Exploration

This section covers additional information and figures for the raw data we had at our disposal for evaluating our multi-system event prediction approach. Since the data was all collected in the year 2018, we omit this information in all dates and times below to ease readability. To avoid overloaded and skewed box plot visualizations, outlier values are deliberately hidden in the majority of the cases, and corresponding data tables provide additional information.

In [Figure C.1](#), the average number of different components per system is shown for all the available 705 systems. [Table C.1](#) provides more detailed insights, including the total number of components (cf. column *Total*). Directly following is the average number of component connections, which is shown in [Figure C.2](#) and [Table C.2](#). In the table, we can see that the maximum number of observed disk-to-host connections is three, which is one of the extremely rare cases as briefly mention in [Section 2.3.2 on p. 8](#). In fact, the entire dataset contains four such cases, which is only 0.0036% of all disk-to-host connections.

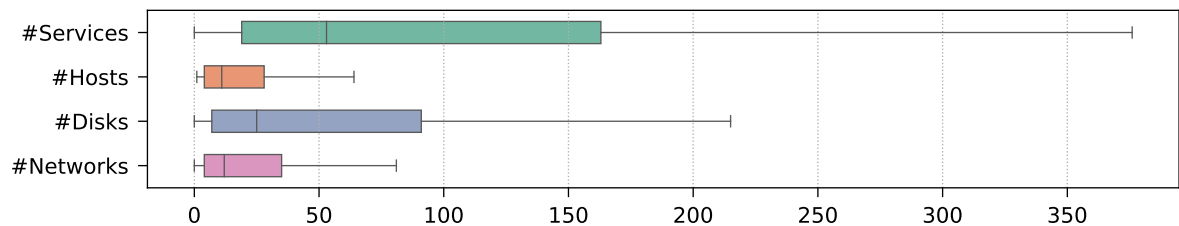


Figure C.1: Component count statistics of the 705 systems, visualized with a box plot. Detailed information is available in [Table C.1](#).

In 434 out of the 705 systems, 17733 slowdown events occurred on 2084 services. The system-averaged event count and number of services (all services, services where events occurred, services where no events occurred) for these 434 systems have already been presented in the main evaluation (cf. [Figure 4.12 on p. 84](#) and [Table 4.5 on p. 84](#)).

The system-averaged available observation periods [*From*, *To*] of the 34 time series metrics are displayed in [Figure C.3](#) for all the 705 systems. In [Figure C.4](#), the average number of time series data points per system is shown for each metric (details are listed in [Table C.3](#)). In total,

Component Type	Total	μ	σ	min	p_{10}	p_{25}	p_{50}	p_{75}	p_{90}	max
#Services	161414	228.96	806.05	0	7	19	53	163	387.8	12245
#Hosts	42703	60.57	442.75	1	2	4	11	28	78.6	10934
#Disks	111978	158.83	538.59	0	3	7	25	91	280.2	6025
#Networks	138405	196.32	2256.03	0	2	4	12	35	112.6	52097

Table C.1: Component count statistics of the 705 systems. μ = average, σ = standard deviation, $p_i = i\%$ percentile, min = minimum, max = maximum.

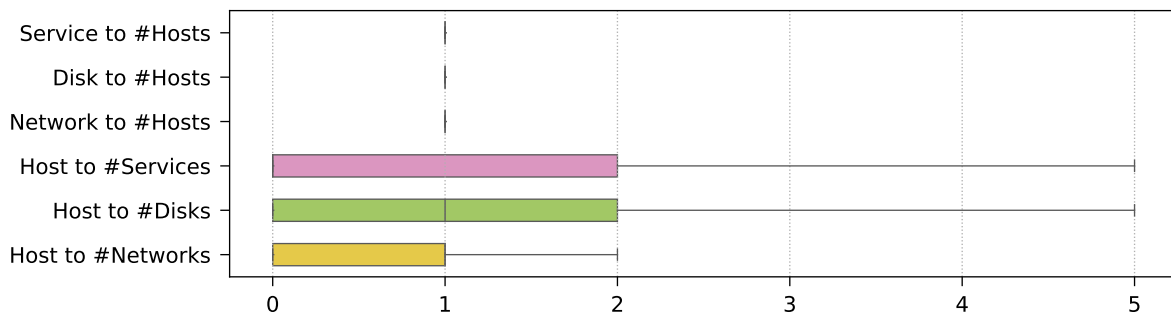


Figure C.2: Connection count statistics of the 705 systems, visualized with a box plot. Detailed information is available in [Table C.2](#).

Connection Type	μ	σ	min	p_{10}	p_{25}	p_{50}	p_{75}	p_{90}	max
Service to #Hosts	1.52	3.88	0	0	1	1	1	3	419
Disk to #Hosts	1	0.02	0	1	1	1	1	1	3
Network to #Hosts	0.89	0.31	0	0	1	1	1	1	1
Host to #Services	5.75	59.05	0	0	0	0	2	9	5671
Host to #Disks	2.63	28.97	0	0	0	1	2	4	2873
Host to #Networks	2.88	132.98	0	0	0	1	1	1	14734

Table C.2: Connection count statistics of the 705 systems. μ = average, σ = standard deviation, $p_i = i\%$ percentile, min = minimum, max = maximum.

the entire observation period contains about 18 billion individual time series data points. Their actual completeness per metric is shown in [Figure C.5](#), once with system-based normalization (the average completeness for all components within a system, i.e., the completeness per system, is averaged across all systems) and once with component-based normalization (the completeness per component is averaged across all components). [Figure C.6](#) covers the completeness for each of the 20 export days. Finally, [Figure C.7](#) even shows the completeness for each individual system out of all the 705 systems (due to confidentiality, all systems are represented via a 5-digit hash code), where the results are first sorted by the most complete system (top to bottom across both columns) and then by the most complete time series metric (left to right), which means that systems with higher time series data availability are at the upper left and less complete ones are at the lower right (cf. system *07628* in [Figure C.7a](#) vs. system *11919* in [Figure C.7d](#)).

C.2 Evaluation Results

This section covers the remaining evaluation metrics obtained from our synthetic system by running the 169 different slide-through sampling configurations with six observation window sizes, as detailed in [Section 4.6.4 on p. 110](#). The accuracy (ACC) is shown in [Figure C.8](#), the true positive rate (TPR) in [Figure C.9](#), the positive predictive values (PPV) in [Figure C.10](#), the false positive rate (FPR) in [Figure C.11](#) and the F1 score in [Figure C.12](#). The results for the augmented training data are also listed here, including the full results regarding the MCC metric. Starting with the 5-times augmented data, the ACC is shown in [Figure C.13](#), the TPR in [Figure C.14](#), the PPV in [Figure C.15](#), the FPR in [Figure C.16](#), the F1 score in [Figure C.17](#) and the MCC in [Figure C.18](#). Continuing with the 10-times augmented data, the ACC is shown in [Figure C.19](#), the TPR in [Figure C.20](#), the PPV in [Figure C.21](#), the FPR in [Figure C.22](#), the F1 score in [Figure C.23](#) and the MCC in [Figure C.24](#).

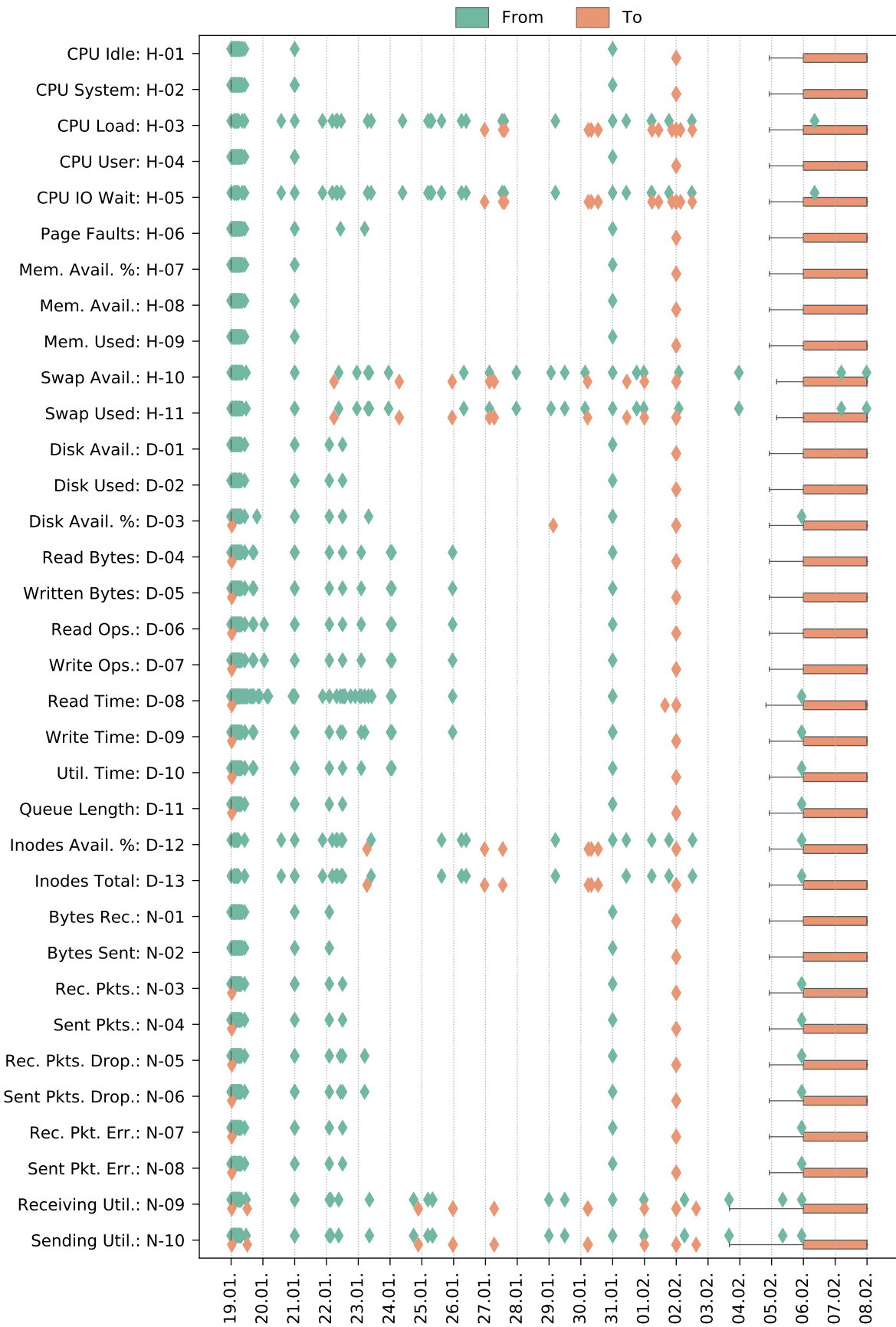


Figure C.3: Time span statistics of the 705 systems given by $[From, To)$ markers in the format *day.month*, visualized with a box plot.

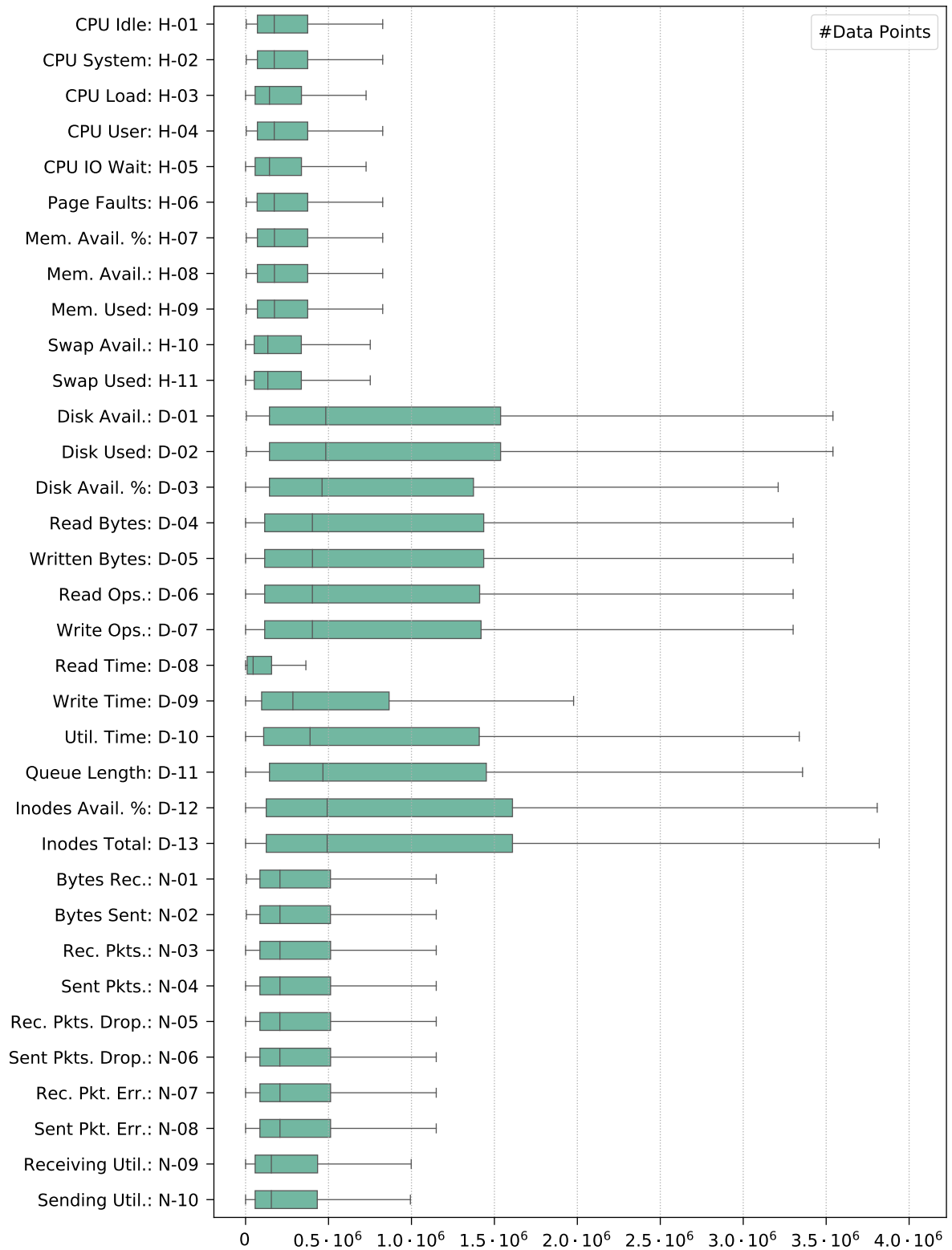


Figure C.4: Time series data point statistics of the 705 systems, visualized with a box plot. Detailed information is available in [Table C.3](#).

ID	#Sys.	Total	μ	σ	min	p_{50}	max
H-01	704	$290.51 \cdot 10^6$	$0.41 \cdot 10^6$	$0.80 \cdot 10^6$	4148	$0.17 \cdot 10^6$	$12.12 \cdot 10^6$
H-02	704	$290.36 \cdot 10^6$	$0.41 \cdot 10^6$	$0.80 \cdot 10^6$	4148	$0.17 \cdot 10^6$	$12.13 \cdot 10^6$
H-03	533	$203.11 \cdot 10^6$	$0.38 \cdot 10^6$	$0.82 \cdot 10^6$	19	$0.14 \cdot 10^6$	$11.71 \cdot 10^6$
H-04	704	$290.45 \cdot 10^6$	$0.41 \cdot 10^6$	$0.80 \cdot 10^6$	4148	$0.17 \cdot 10^6$	$12.12 \cdot 10^6$
H-05	533	$201.97 \cdot 10^6$	$0.38 \cdot 10^6$	$0.82 \cdot 10^6$	19	$0.14 \cdot 10^6$	$11.71 \cdot 10^6$
H-06	704	$291.11 \cdot 10^6$	$0.41 \cdot 10^6$	$0.80 \cdot 10^6$	4148	$0.17 \cdot 10^6$	$12.12 \cdot 10^6$
H-07	704	$291.20 \cdot 10^6$	$0.41 \cdot 10^6$	$0.80 \cdot 10^6$	4148	$0.17 \cdot 10^6$	$12.12 \cdot 10^6$
H-08	704	$291.19 \cdot 10^6$	$0.41 \cdot 10^6$	$0.80 \cdot 10^6$	4148	$0.17 \cdot 10^6$	$12.12 \cdot 10^6$
H-09	704	$291.01 \cdot 10^6$	$0.41 \cdot 10^6$	$0.80 \cdot 10^6$	4148	$0.17 \cdot 10^6$	$12.12 \cdot 10^6$
H-10	586	$197.81 \cdot 10^6$	$0.34 \cdot 10^6$	$0.71 \cdot 10^6$	6	$0.13 \cdot 10^6$	$12.12 \cdot 10^6$
H-11	586	$197.80 \cdot 10^6$	$0.34 \cdot 10^6$	$0.71 \cdot 10^6$	6	$0.13 \cdot 10^6$	$12.12 \cdot 10^6$
D-01	697	$1108.71 \cdot 10^6$	$1.59 \cdot 10^6$	$3.20 \cdot 10^6$	4689	$0.48 \cdot 10^6$	$33.30 \cdot 10^6$
D-02	697	$1105.76 \cdot 10^6$	$1.59 \cdot 10^6$	$3.18 \cdot 10^6$	4689	$0.48 \cdot 10^6$	$33.30 \cdot 10^6$
D-03	667	$995.56 \cdot 10^6$	$1.49 \cdot 10^6$	$2.93 \cdot 10^6$	14	$0.46 \cdot 10^6$	$28.01 \cdot 10^6$
D-04	689	$1039.14 \cdot 10^6$	$1.51 \cdot 10^6$	$3.16 \cdot 10^6$	72	$0.40 \cdot 10^6$	$30.41 \cdot 10^6$
D-05	689	$1035.32 \cdot 10^6$	$1.50 \cdot 10^6$	$3.12 \cdot 10^6$	72	$0.40 \cdot 10^6$	$30.41 \cdot 10^6$
D-06	689	$1029.35 \cdot 10^6$	$1.49 \cdot 10^6$	$3.15 \cdot 10^6$	72	$0.40 \cdot 10^6$	$30.41 \cdot 10^6$
D-07	689	$1025.98 \cdot 10^6$	$1.49 \cdot 10^6$	$3.13 \cdot 10^6$	72	$0.40 \cdot 10^6$	$30.42 \cdot 10^6$
D-08	683	$108.74 \cdot 10^6$	$0.16 \cdot 10^6$	$0.31 \cdot 10^6$	7	45565	$2.71 \cdot 10^6$
D-09	683	$591.50 \cdot 10^6$	$0.87 \cdot 10^6$	$1.65 \cdot 10^6$	14	$0.29 \cdot 10^6$	$18.30 \cdot 10^6$
D-10	684	$971.38 \cdot 10^6$	$1.42 \cdot 10^6$	$2.93 \cdot 10^6$	14	$0.39 \cdot 10^6$	$27.98 \cdot 10^6$
D-11	687	$1071.49 \cdot 10^6$	$1.56 \cdot 10^6$	$3.07 \cdot 10^6$	14	$0.47 \cdot 10^6$	$28.01 \cdot 10^6$
D-12	481	$811.75 \cdot 10^6$	$1.69 \cdot 10^6$	$3.24 \cdot 10^6$	14	$0.49 \cdot 10^6$	$27.62 \cdot 10^6$
D-13	481	$817.99 \cdot 10^6$	$1.70 \cdot 10^6$	$3.32 \cdot 10^6$	14	$0.49 \cdot 10^6$	$27.84 \cdot 10^6$
N-01	700	$380.14 \cdot 10^6$	$0.54 \cdot 10^6$	$1.16 \cdot 10^6$	5019	$0.21 \cdot 10^6$	$15.31 \cdot 10^6$
N-02	700	$380.69 \cdot 10^6$	$0.54 \cdot 10^6$	$1.17 \cdot 10^6$	5019	$0.21 \cdot 10^6$	$15.76 \cdot 10^6$
N-03	686	$357.96 \cdot 10^6$	$0.52 \cdot 10^6$	$1.06 \cdot 10^6$	7	$0.21 \cdot 10^6$	$14.97 \cdot 10^6$
N-04	686	$356.68 \cdot 10^6$	$0.52 \cdot 10^6$	$1.03 \cdot 10^6$	7	$0.21 \cdot 10^6$	$13.96 \cdot 10^6$
N-05	686	$356.51 \cdot 10^6$	$0.52 \cdot 10^6$	$1.03 \cdot 10^6$	7	$0.21 \cdot 10^6$	$13.96 \cdot 10^6$
N-06	686	$356.61 \cdot 10^6$	$0.52 \cdot 10^6$	$1.04 \cdot 10^6$	7	$0.21 \cdot 10^6$	$13.96 \cdot 10^6$
N-07	686	$357.15 \cdot 10^6$	$0.52 \cdot 10^6$	$1.04 \cdot 10^6$	7	$0.21 \cdot 10^6$	$13.95 \cdot 10^6$
N-08	686	$356.48 \cdot 10^6$	$0.52 \cdot 10^6$	$1.03 \cdot 10^6$	7	$0.21 \cdot 10^6$	$13.50 \cdot 10^6$
N-09	603	$266.21 \cdot 10^6$	$0.44 \cdot 10^6$	$0.98 \cdot 10^6$	2	$0.16 \cdot 10^6$	$13.37 \cdot 10^6$
N-10	603	$266 \cdot 10^6$	$0.44 \cdot 10^6$	$0.98 \cdot 10^6$	2	$0.16 \cdot 10^6$	$13.48 \cdot 10^6$

Table C.3: Time series data point statistics of the 705 systems, where $\#Sys.$ represents the actual number of systems that provide the particular metric. μ = average, σ = standard deviation, $p_i = i\%$ percentile, min = minimum, max = maximum.

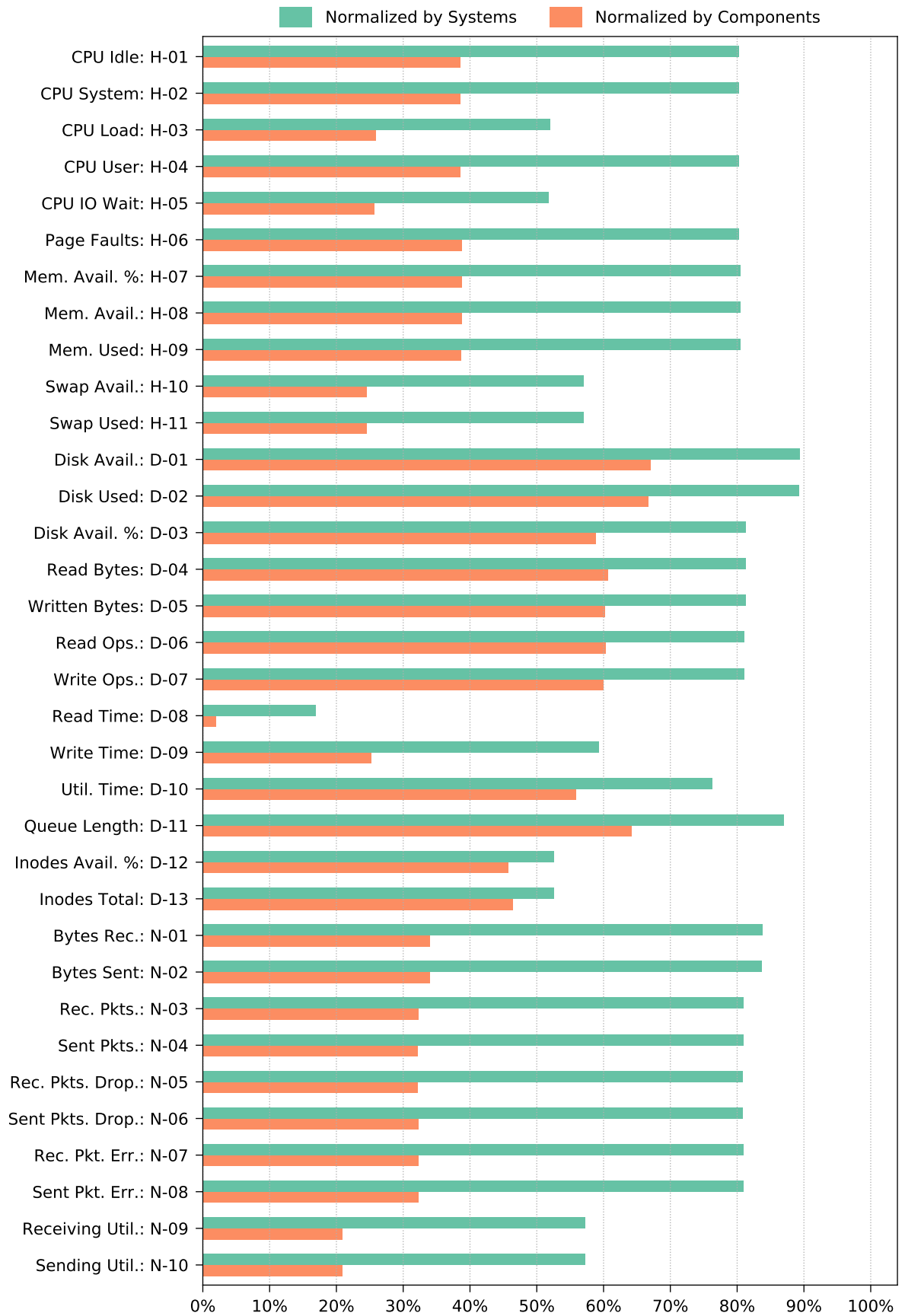


Figure C.5: Time series data point completeness (in percent) of the 705 systems, normalized across all systems and all components, respectively.

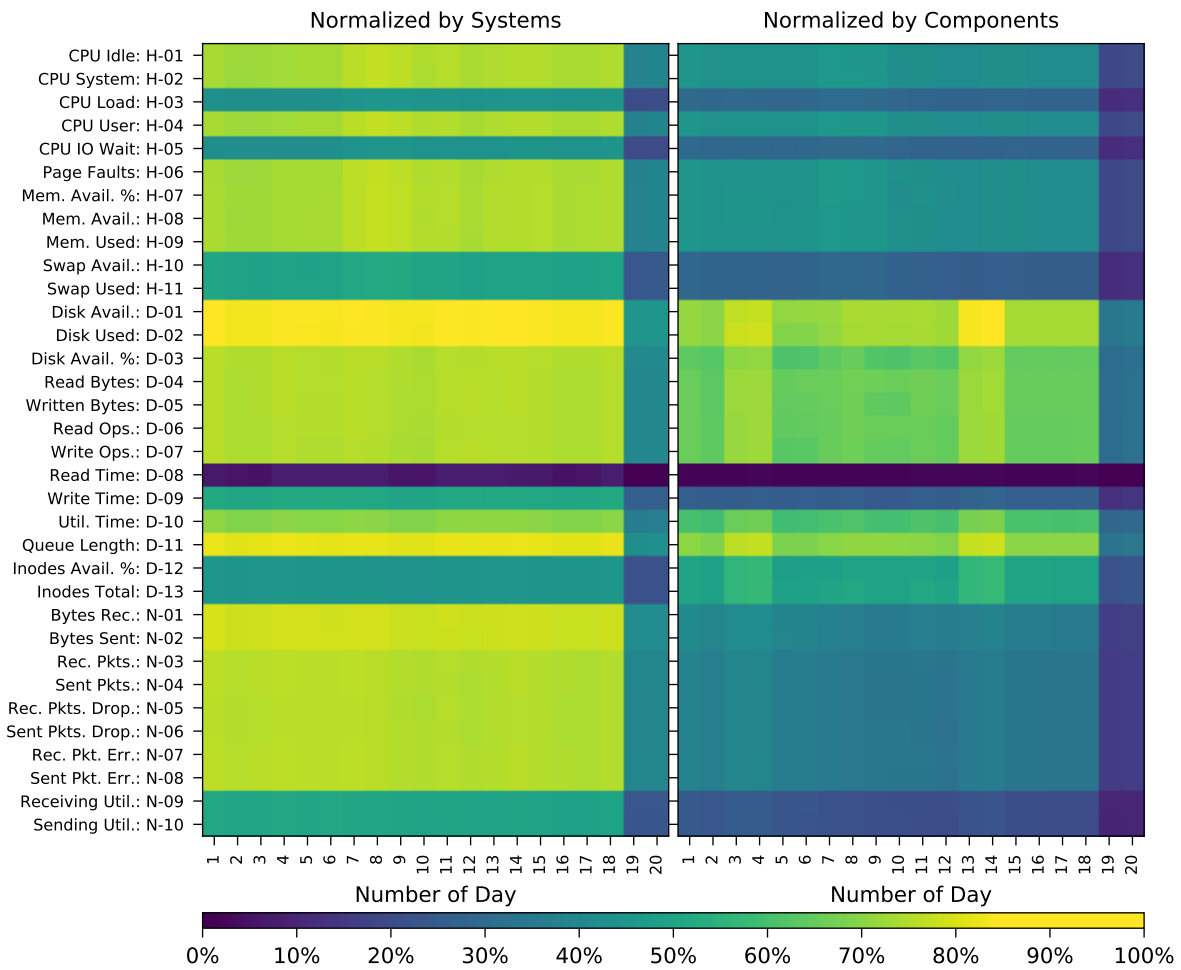
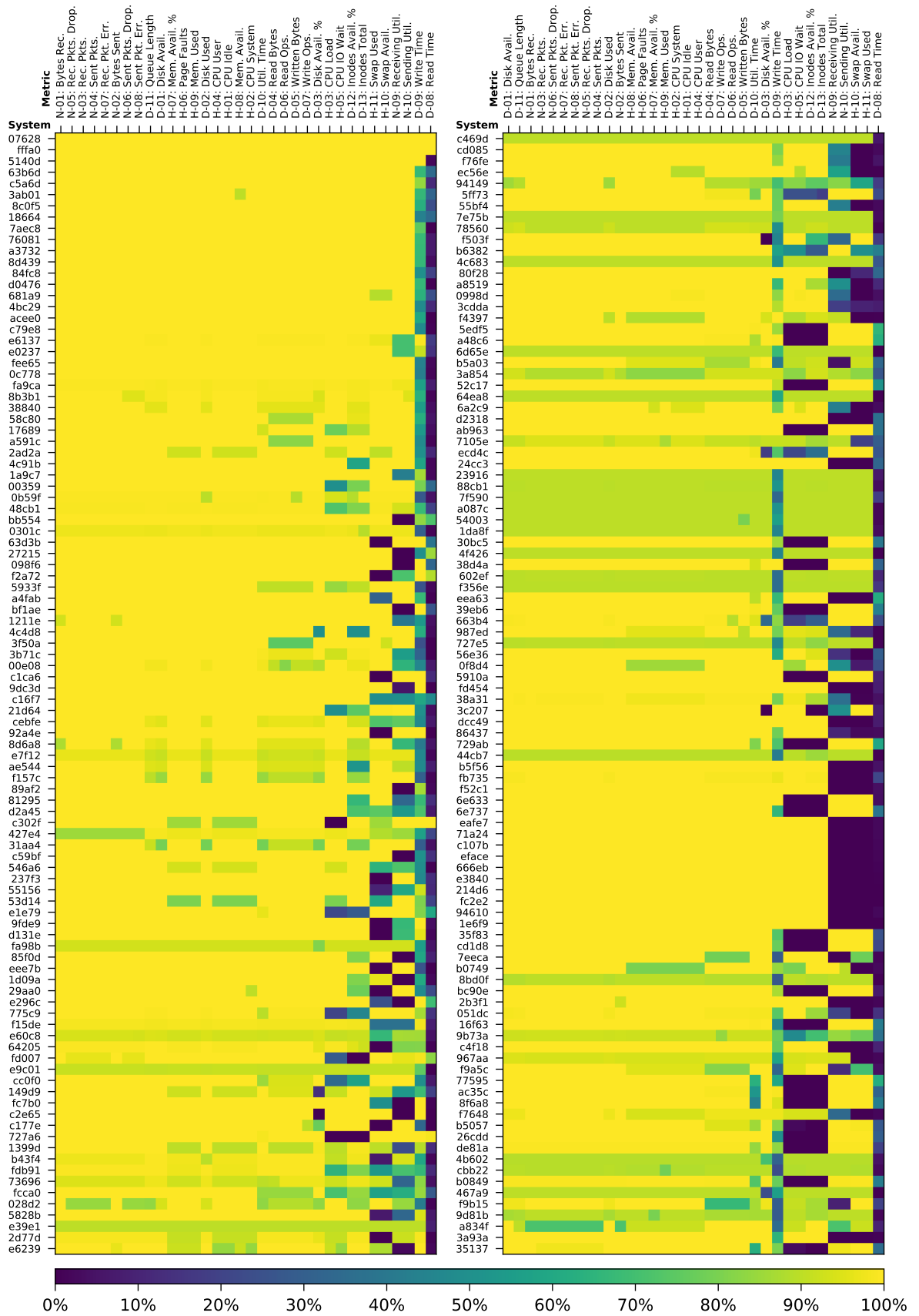
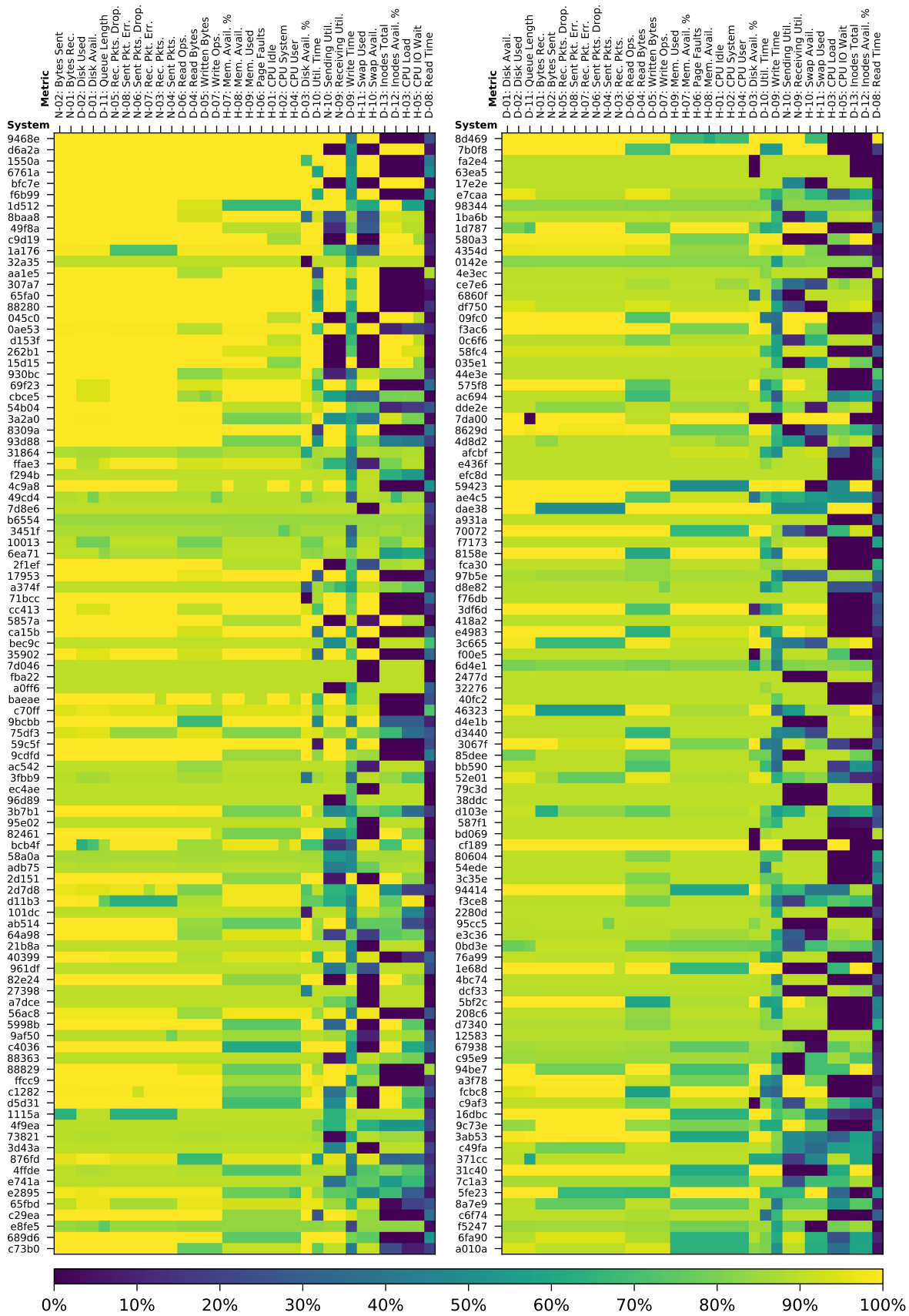


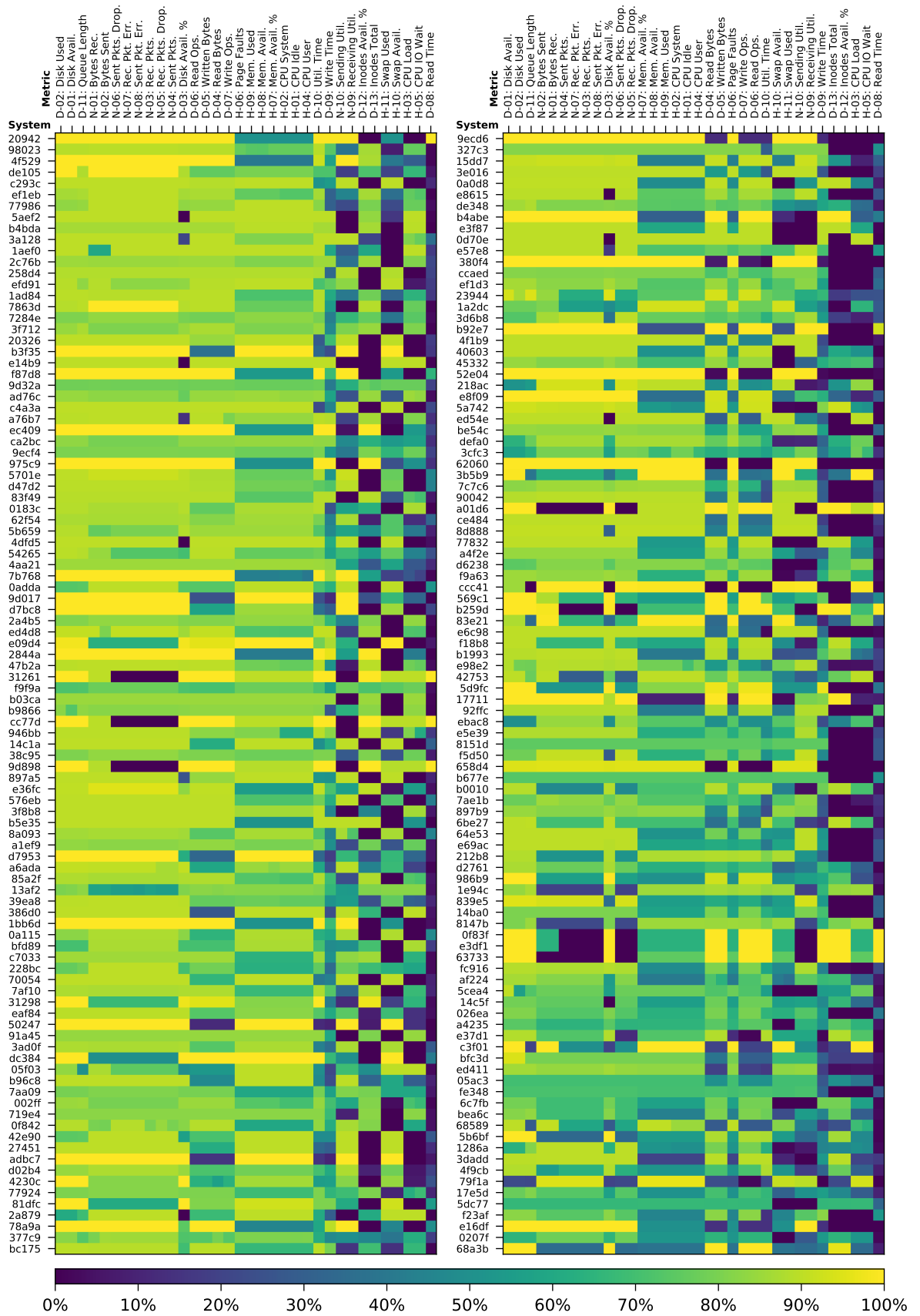
Figure C.6: Time series data point completeness per day (in percent) of the 705 systems, normalized across all systems and all components, respectively.



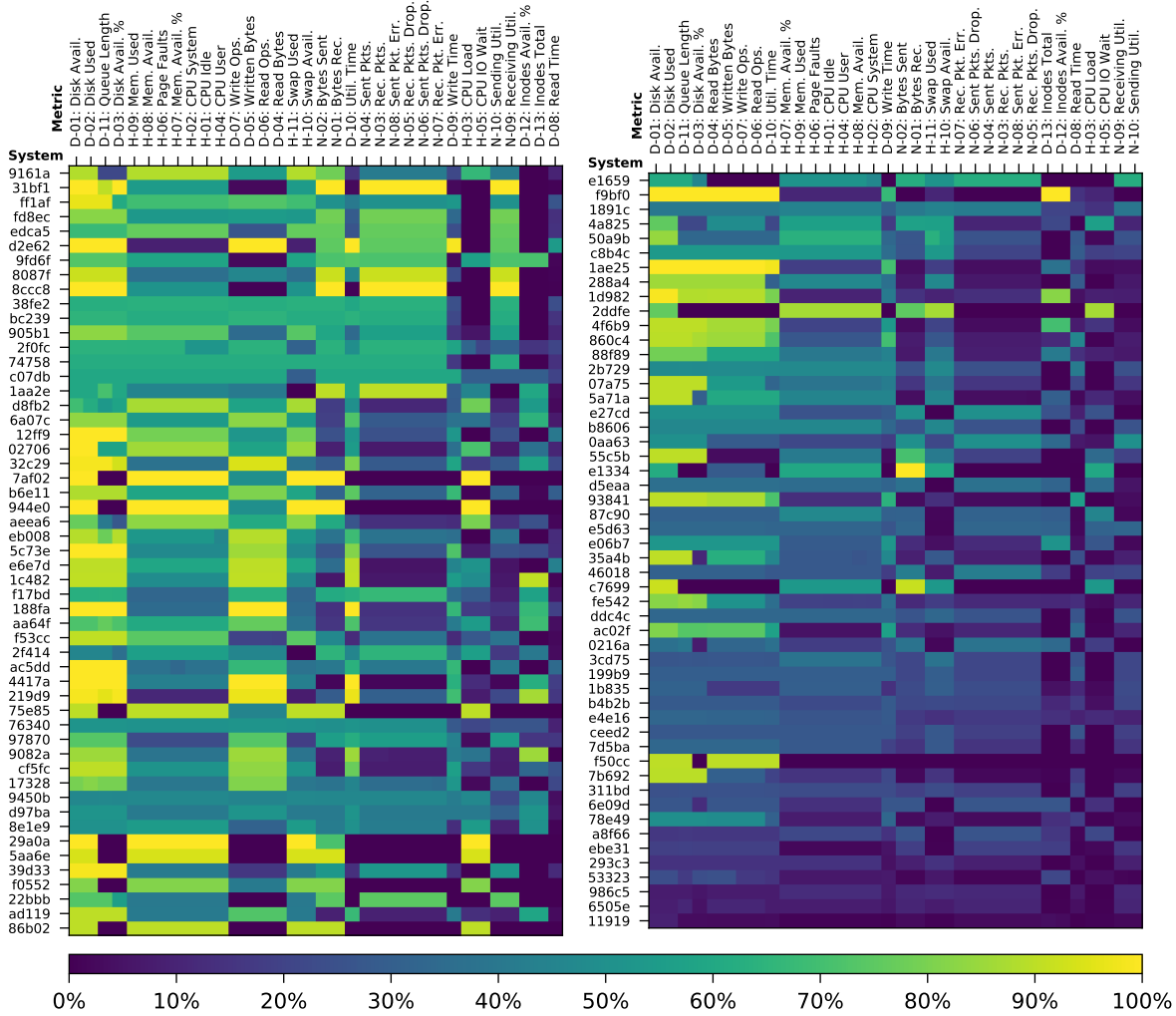
(a) Time series data point completeness per system (in percent) of the 200 systems $S_1 = \{s_1, \dots, s_{200}\}$ out of all 705 systems $S = \{s_1, \dots, s_{705}\}$, i.e., $S_1 \subset S$, first sorted by the most complete system (top to bottom) and then by the most complete time series metric (left to right).



(b) Time series data point completeness per system (in percent) of the 200 systems $S_2 = \{s_{201}, \dots, s_{400}\}$ out of all 705 systems $S = \{s_1, \dots, s_{705}\}$, i.e., $S_2 \subset S$, first sorted by the most complete system (top to bottom) and then by the most complete time series metric (left to right).

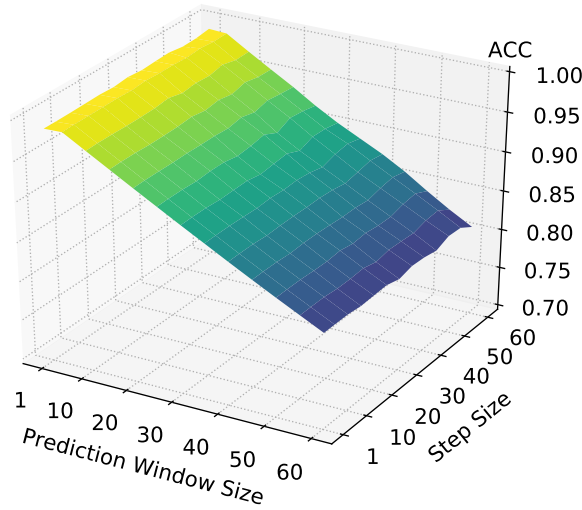


(c) Time series data point completeness per system (in percent) of the 200 systems $S_3 = \{s_{401}, \dots, s_{600}\}$ out of all 705 systems $S = \{s_1, \dots, s_{705}\}$, i.e., $S_3 \subset S$, first sorted by the most complete system (top to bottom) and then by the most complete time series metric (left to right).

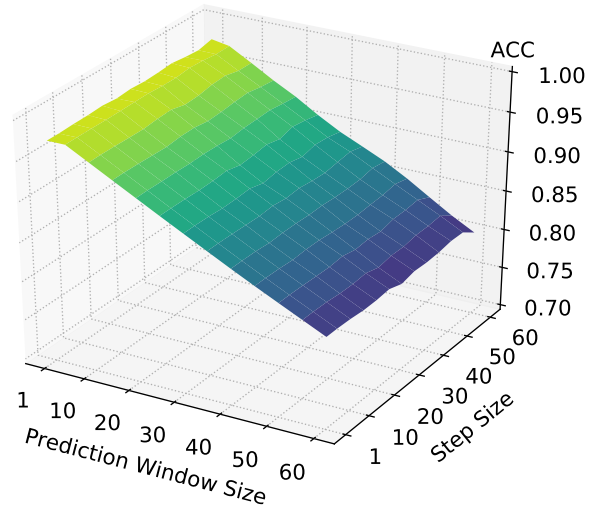


(d) Time series data point completeness per system (in percent) of the 105 systems $S_4 = \{s_{601}, \dots, s_{705}\}$ out of all 705 systems $S = \{s_1, \dots, s_{705}\}$, i.e., $S_4 \subset S$, first sorted by the most complete system (top to bottom) and then by the most complete time series metric (left to right).

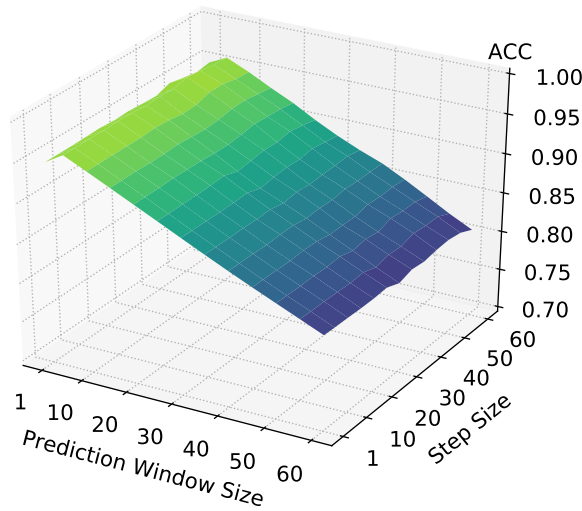
Figure C.7: Time series data point completeness per system (in percent) of the 705 systems, first sorted by the most complete system (top to bottom) and then by the most complete time series metric (left to right).



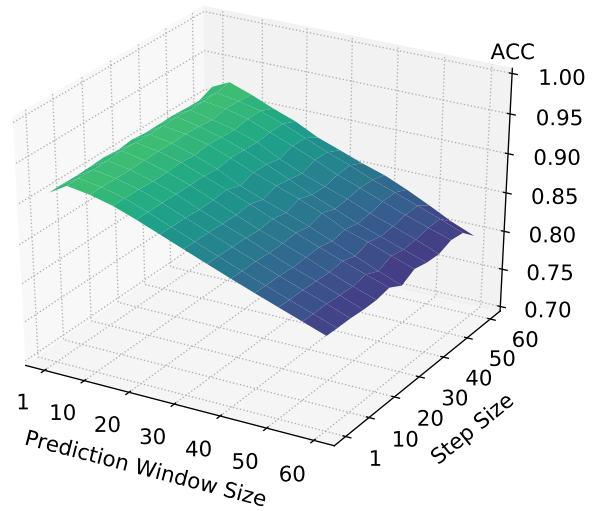
(a) ACC for the 5min observation windows.



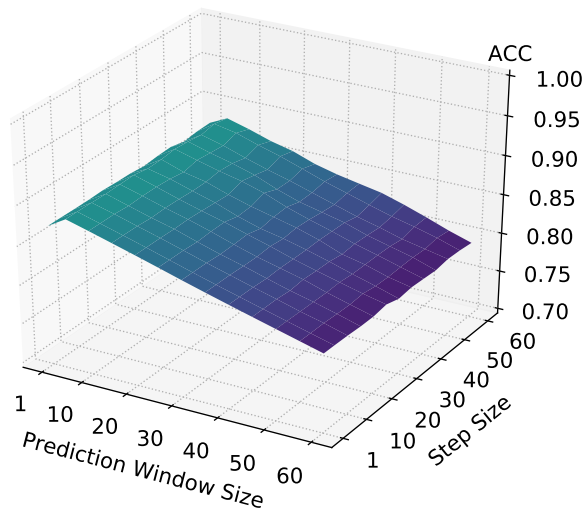
(b) ACC for the 10min observation windows.



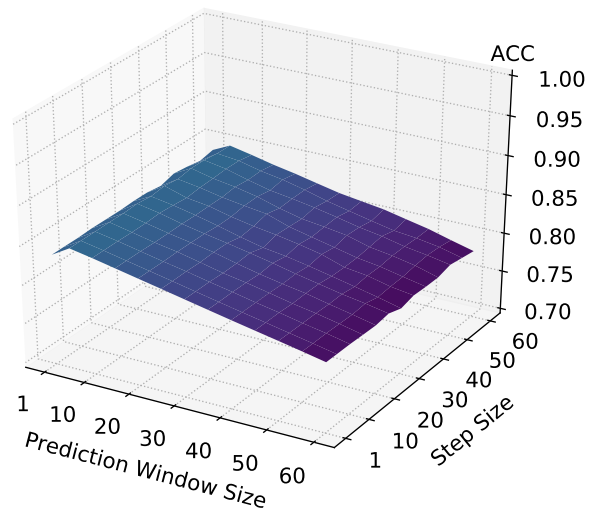
(c) ACC for the 15min observation windows.



(d) ACC for the 30min observation windows.

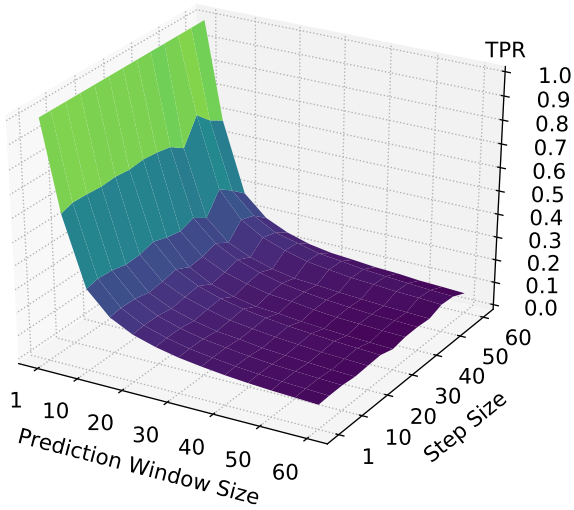


(e) ACC for the 45min observation windows.

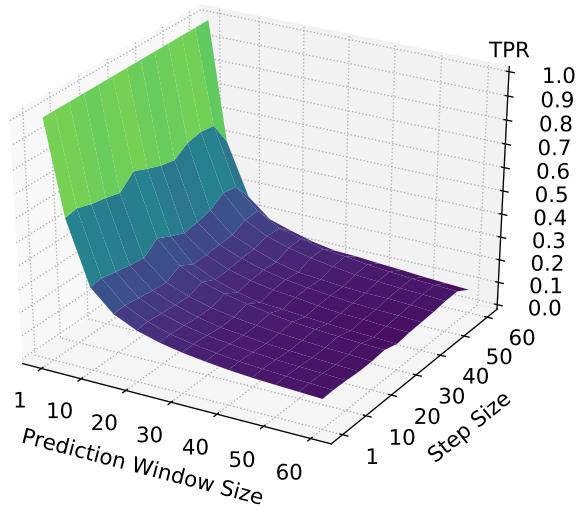


(f) ACC for the 60min observation windows.

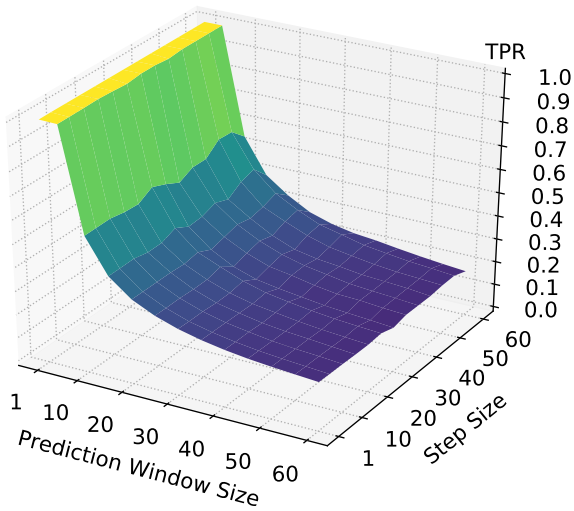
Figure C.8: ACC for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes. Yellow tones indicate better values, purple worse values.



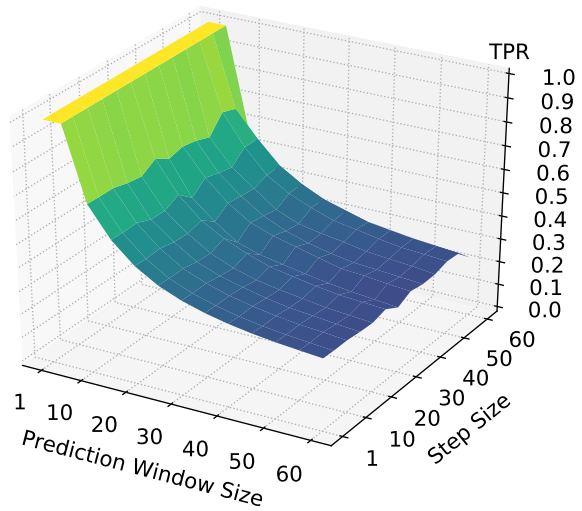
(a) TPR for the 5min observation windows.



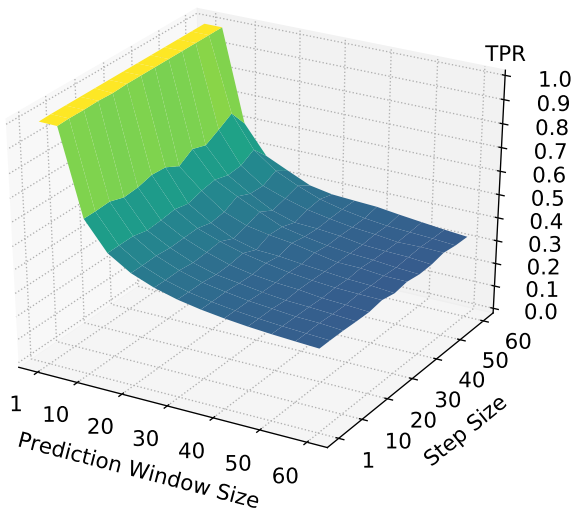
(b) TPR for the 10min observation windows.



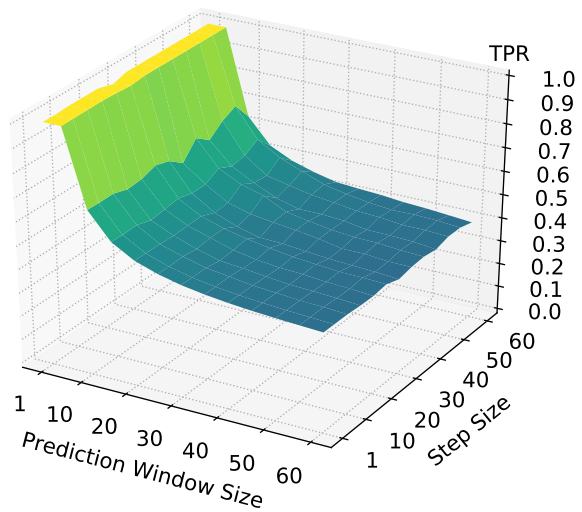
(c) TPR for the 15min observation windows.



(d) TPR for the 30min observation windows.

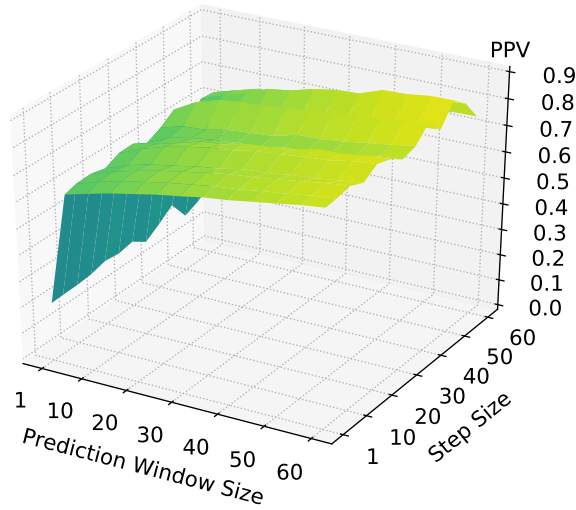


(e) TPR for the 45min observation windows.

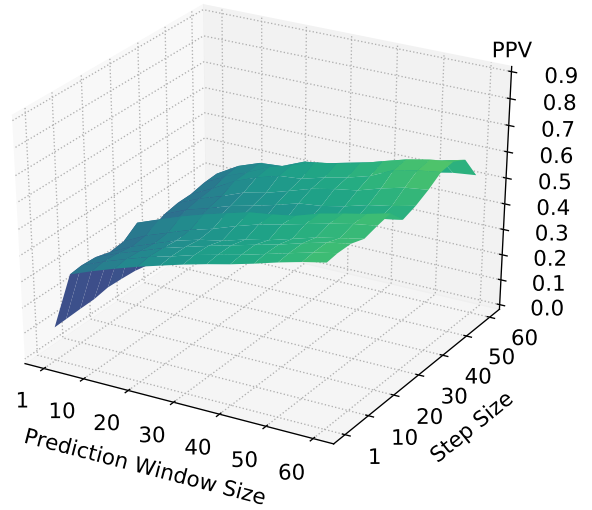


(f) TPR for the 60min observation windows.

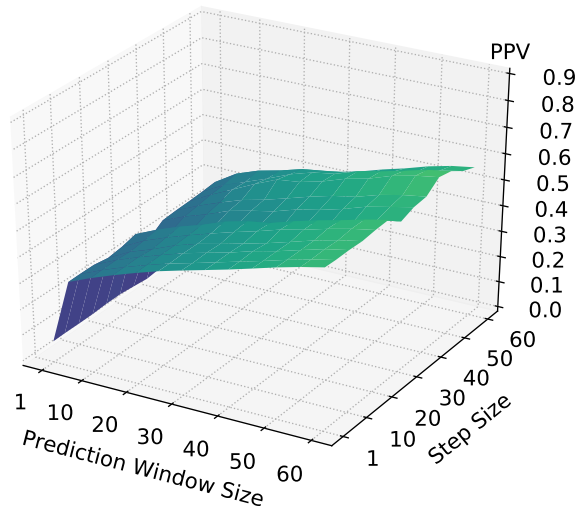
Figure C.9: TPR for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes. Yellow tones indicate better values, purple worse values.



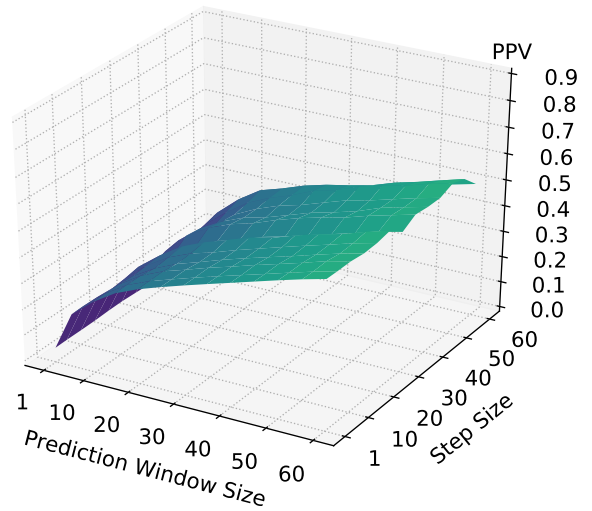
(a) PPV for the 5min observation windows.



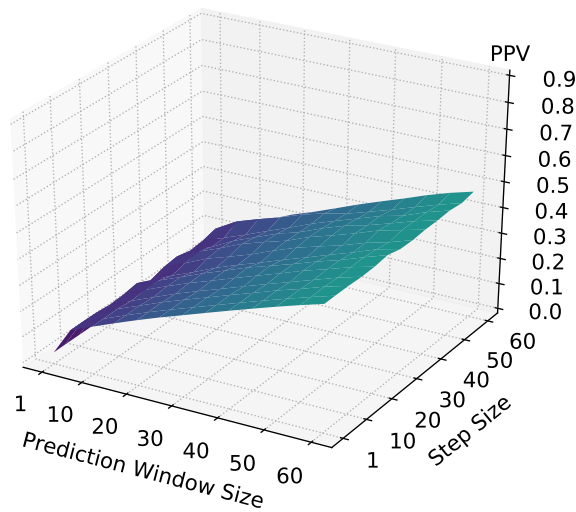
(b) PPV for the 10min observation windows.



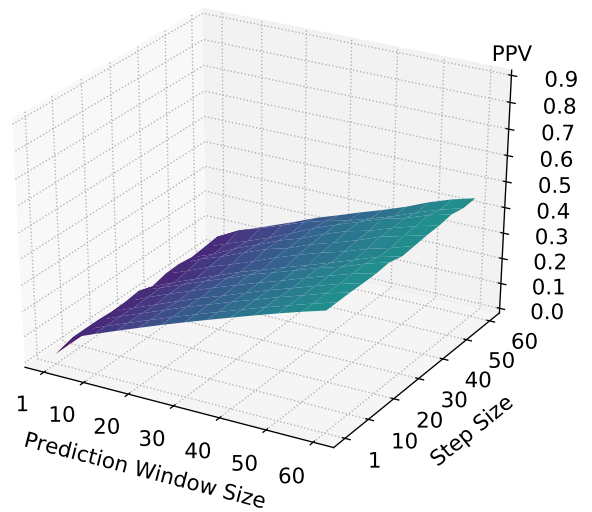
(c) PPV for the 15min observation windows.



(d) PPV for the 30min observation windows.

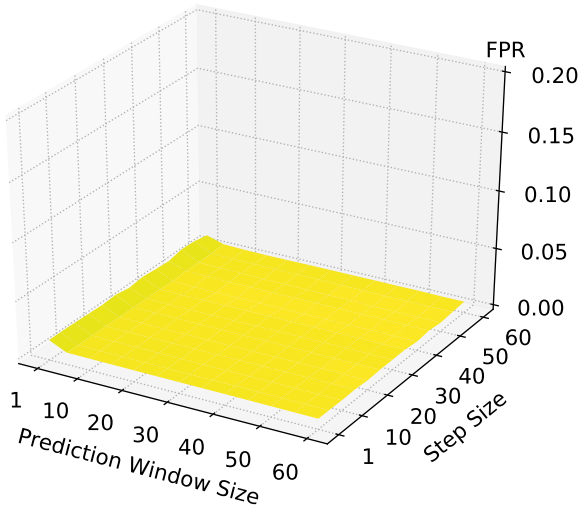


(e) PPV for the 45min observation windows.

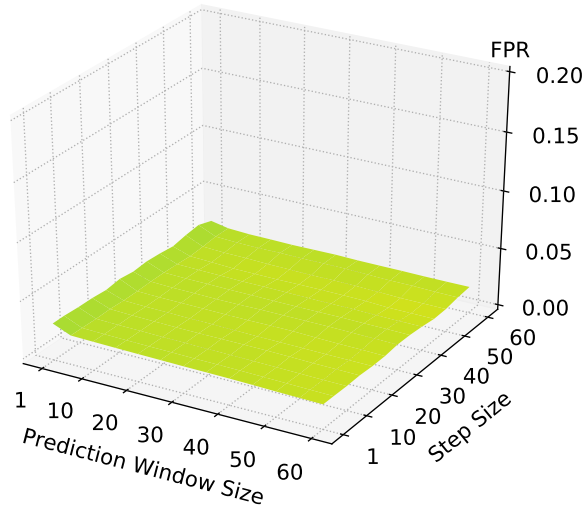


(f) PPV for the 60min observation windows.

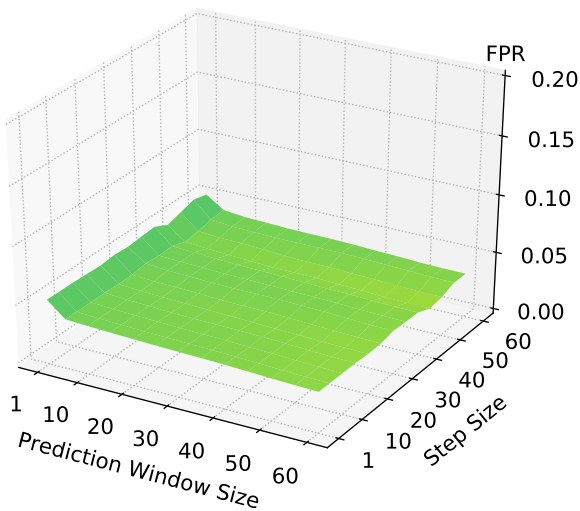
Figure C.10: PPV for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes. Yellow tones indicate better values, purple worse values.



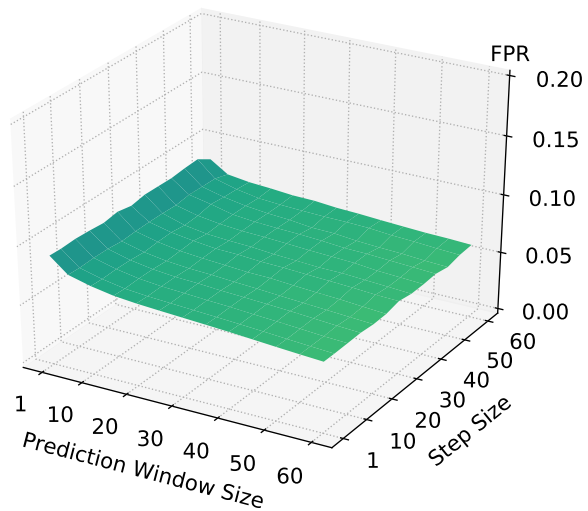
(a) FPR for the 5min observation windows.



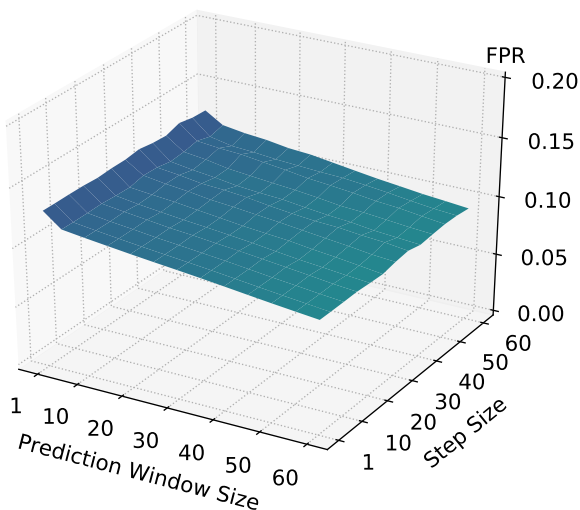
(b) FPR for the 10min observation windows.



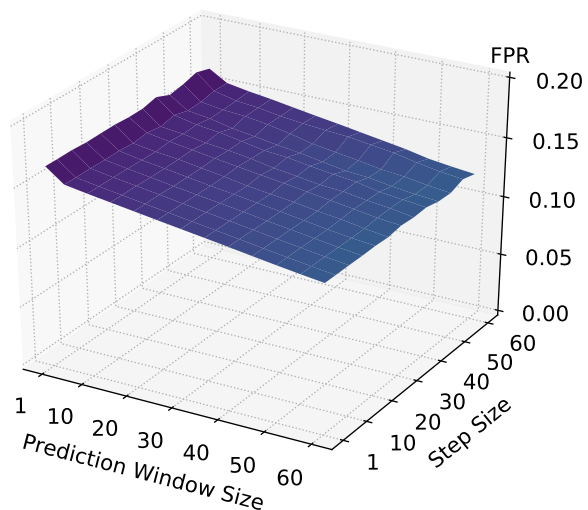
(c) FPR for the 15min observation windows.



(d) FPR for the 30min observation windows.

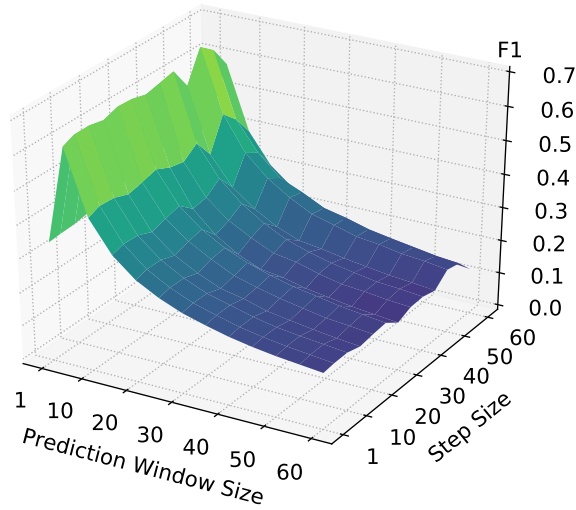


(e) FPR for the 45min observation windows.

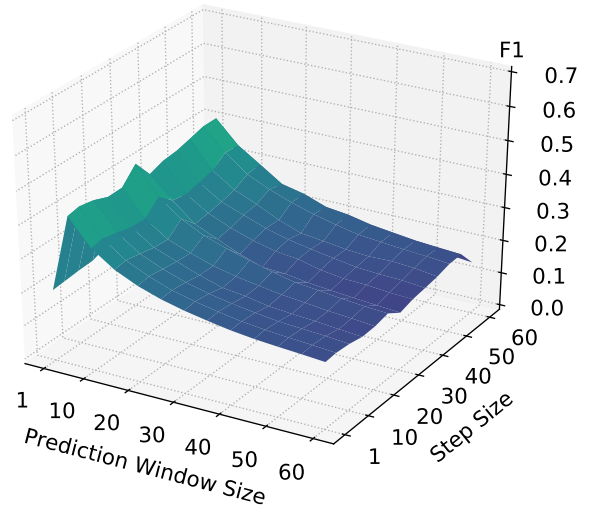


(f) FPR for the 60min observation windows.

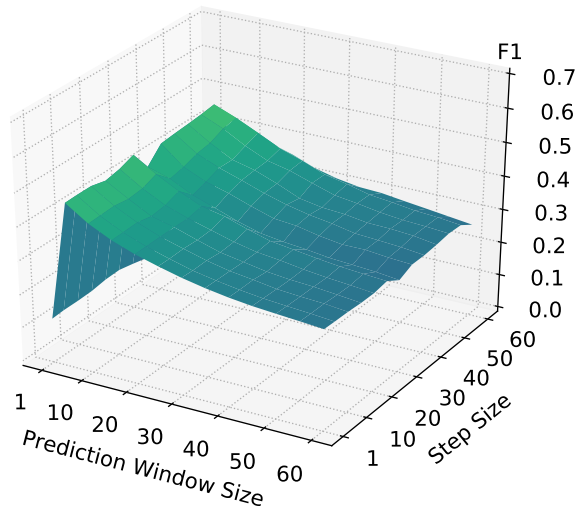
Figure C.11: FPR for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes. Yellow tones indicate better values, purple worse values.



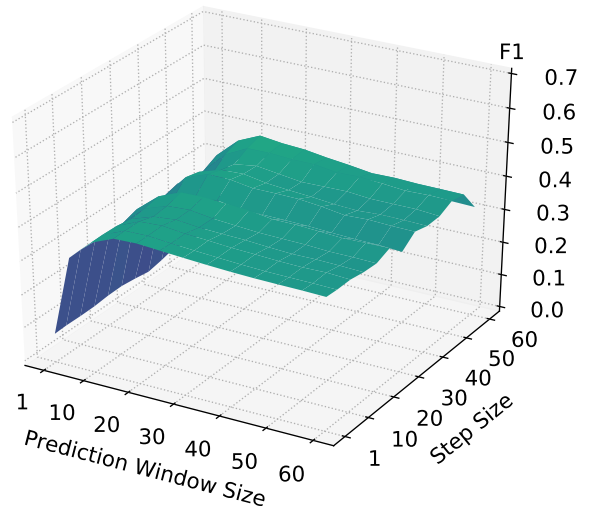
(a) F1 for the 5min observation windows.



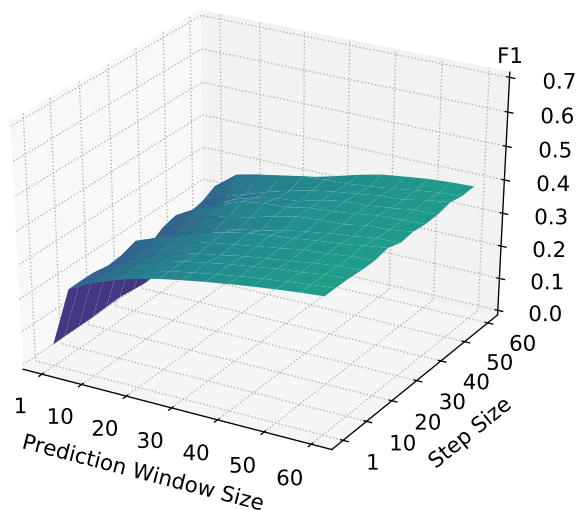
(b) F1 for the 10min observation windows.



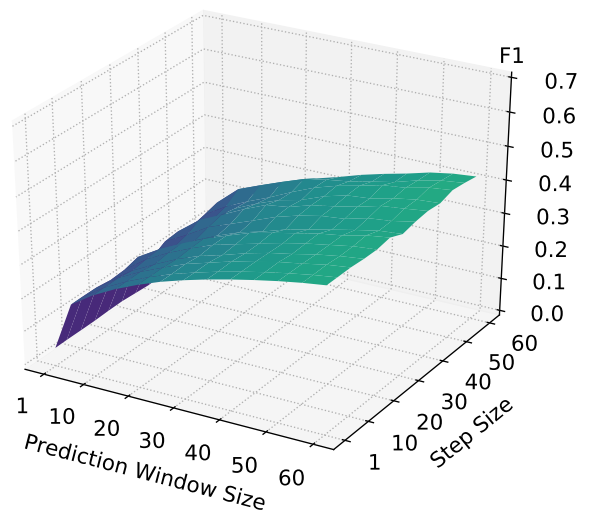
(c) F1 for the 15min observation windows.



(d) F1 for the 30min observation windows.

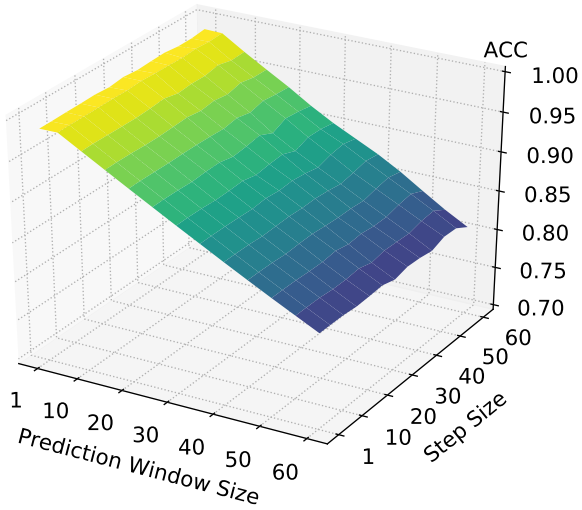


(e) F1 for the 45min observation windows.

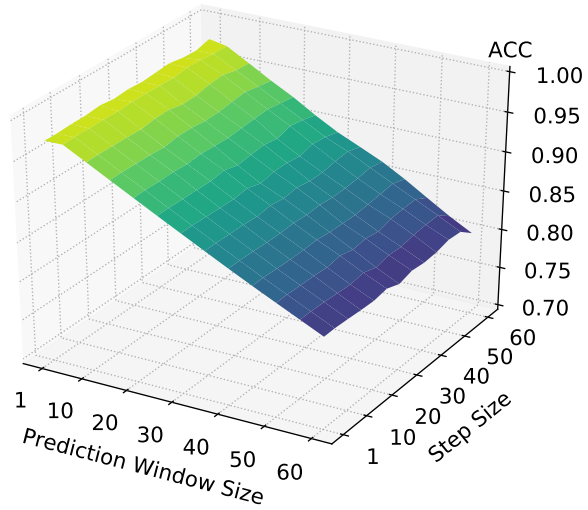


(f) F1 for the 60min observation windows.

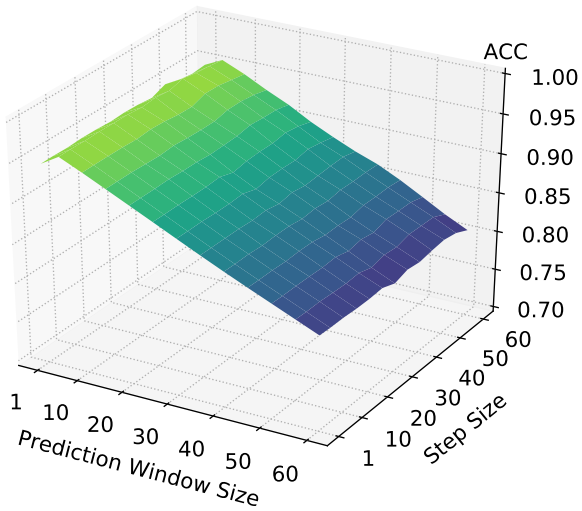
Figure C.12: F1 for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes. Yellow tones indicate better values, purple worse values.



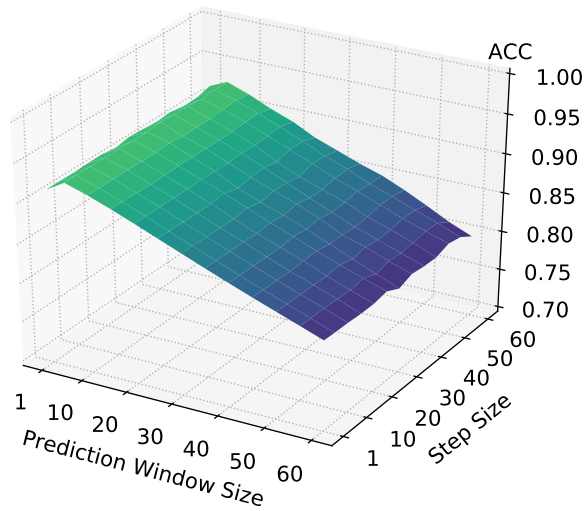
(a) ACC for the 5min observation windows.



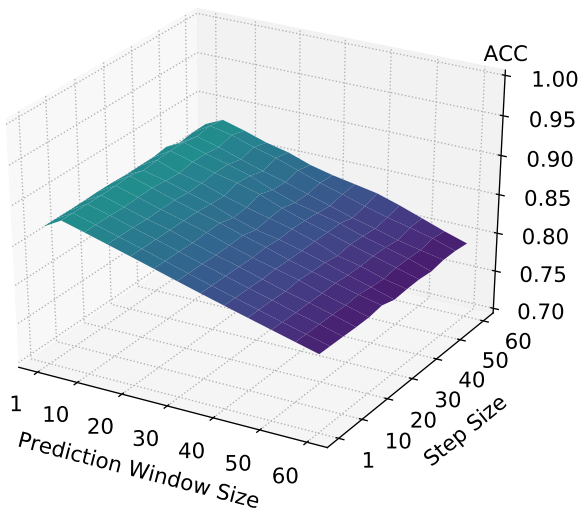
(b) ACC for the 10min observation windows.



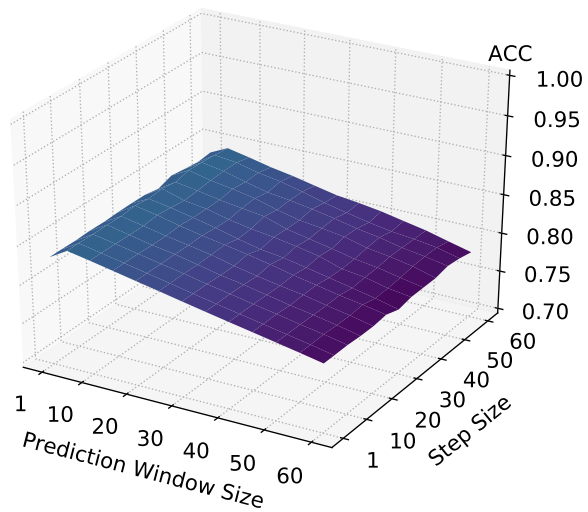
(c) ACC for the 15min observation windows.



(d) ACC for the 30min observation windows.

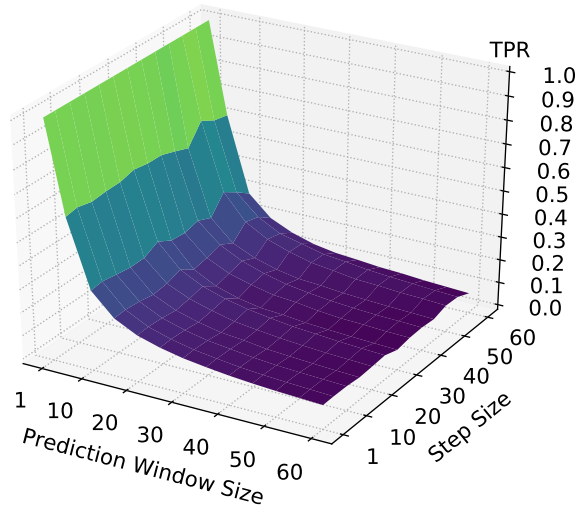


(e) ACC for the 45min observation windows.

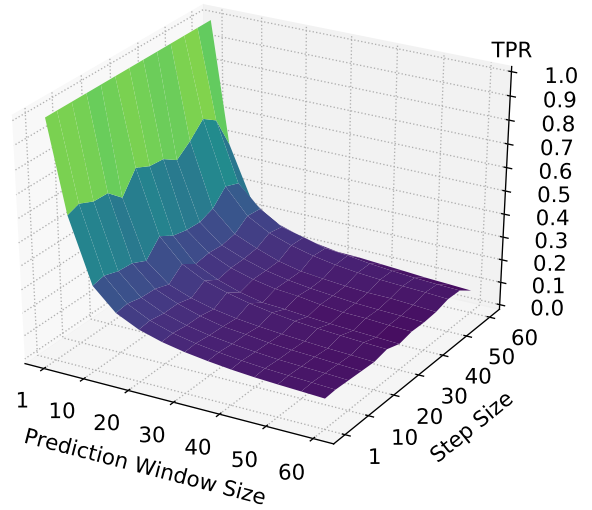


(f) ACC for the 60min observation windows.

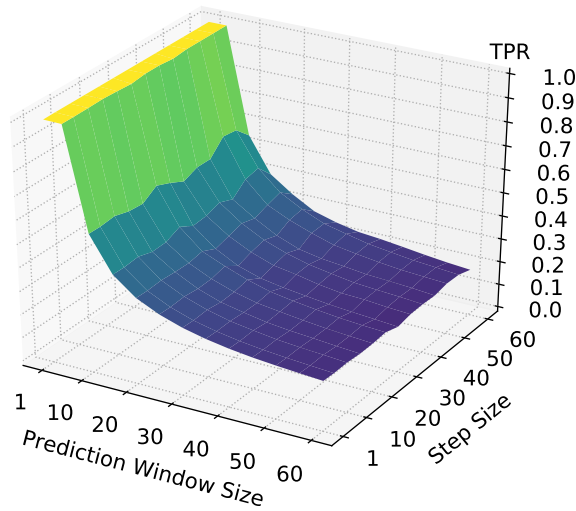
Figure C.13: 5-times augmented data: ACC for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.



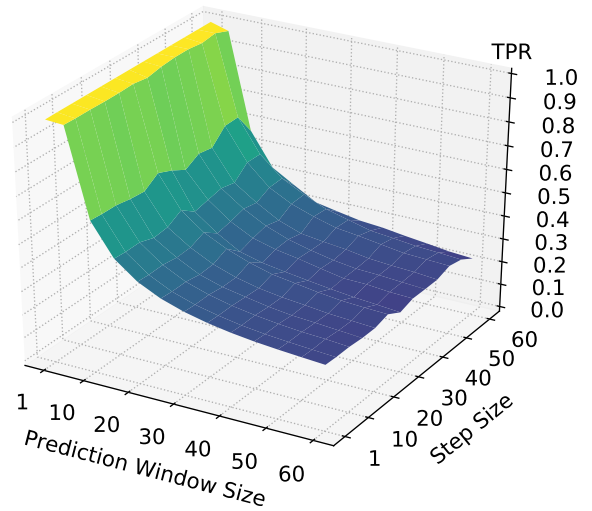
(a) TPR for the 5min observation windows.



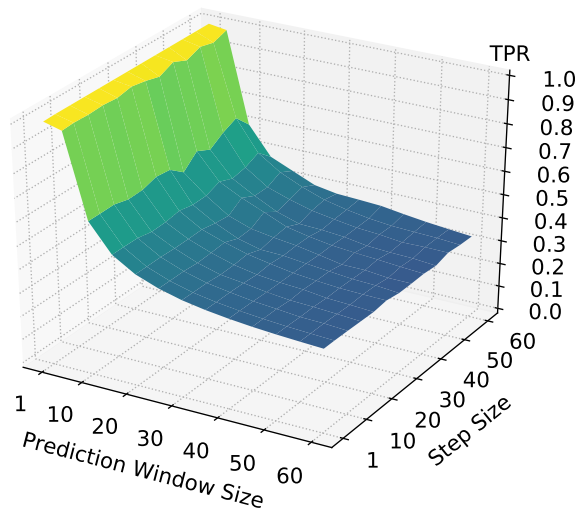
(b) TPR for the 10min observation windows.



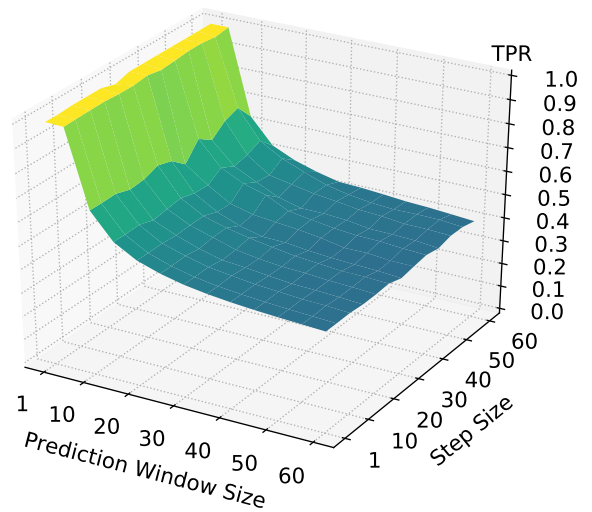
(c) TPR for the 15min observation windows.



(d) TPR for the 30min observation windows.

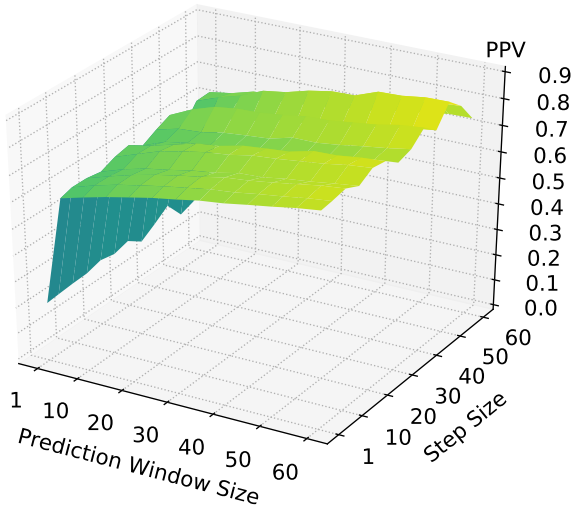


(e) TPR for the 45min observation windows.

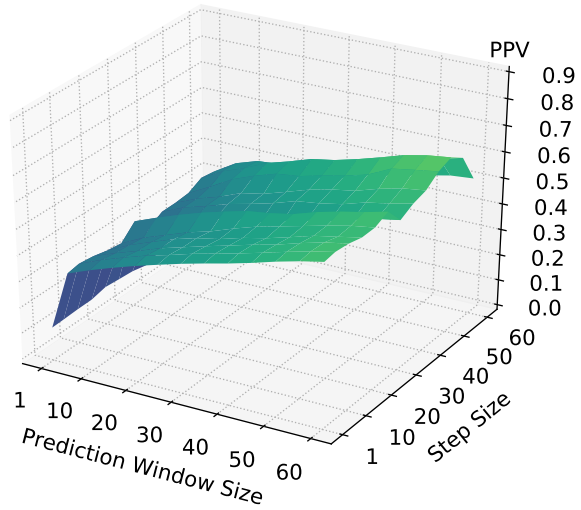


(f) TPR for the 60min observation windows.

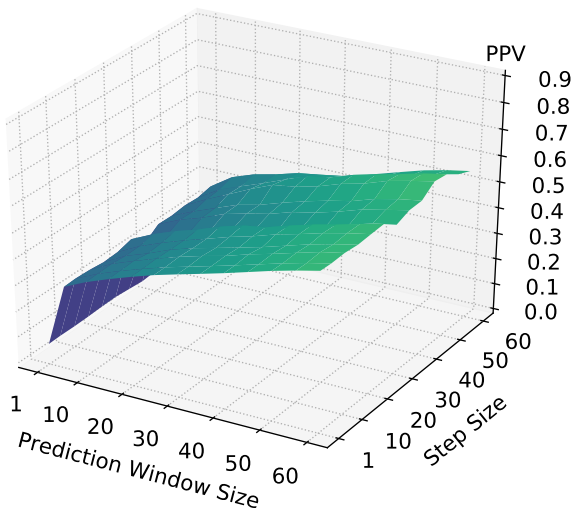
Figure C.14: 5-times augmented data: TPR for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.



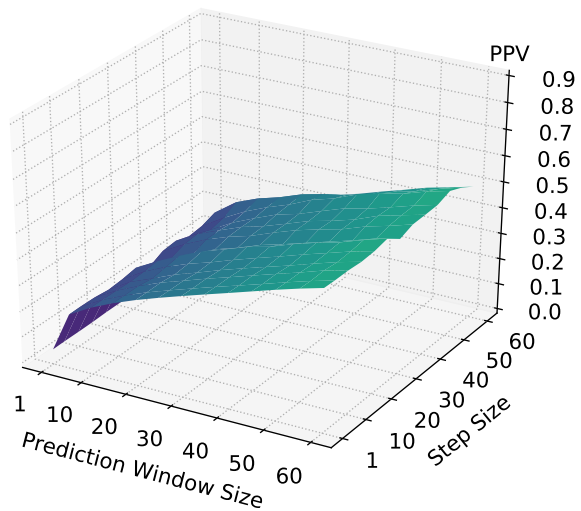
(a) PPV for the 5min observation windows.



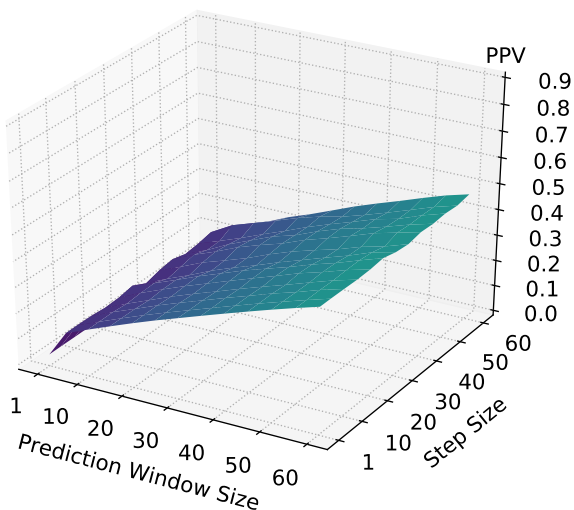
(b) PPV for the 10min observation windows.



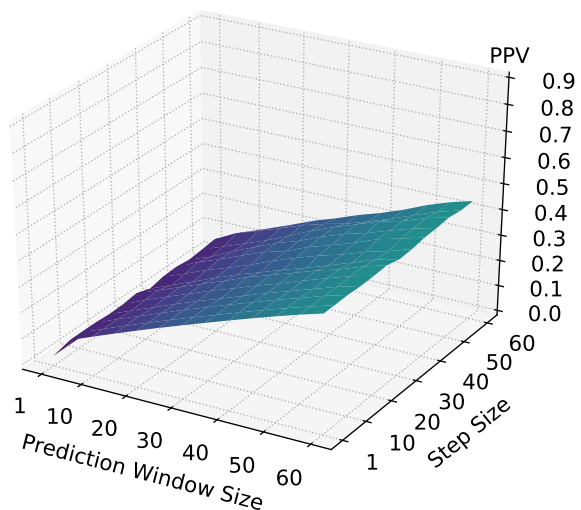
(c) PPV for the 15min observation windows.



(d) PPV for the 30min observation windows.

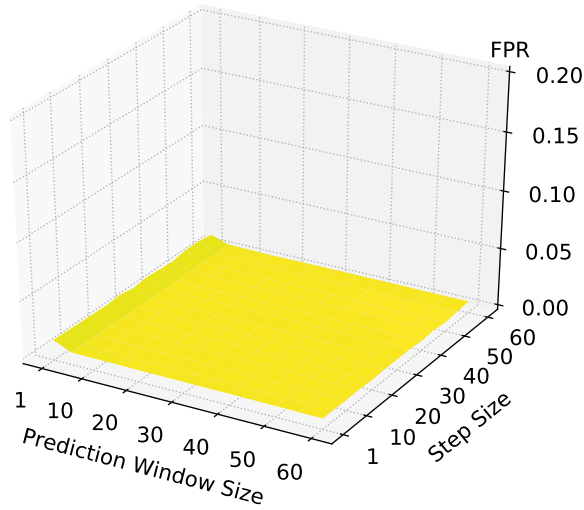


(e) PPV for the 45min observation windows.

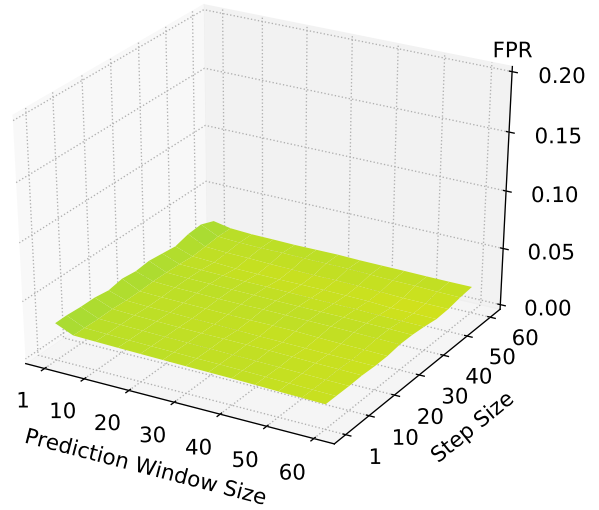


(f) PPV for the 60min observation windows.

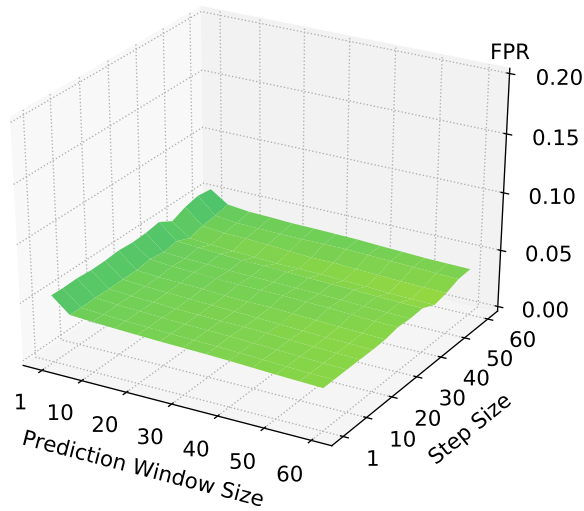
Figure C.15: 5-times augmented data: PPV for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.



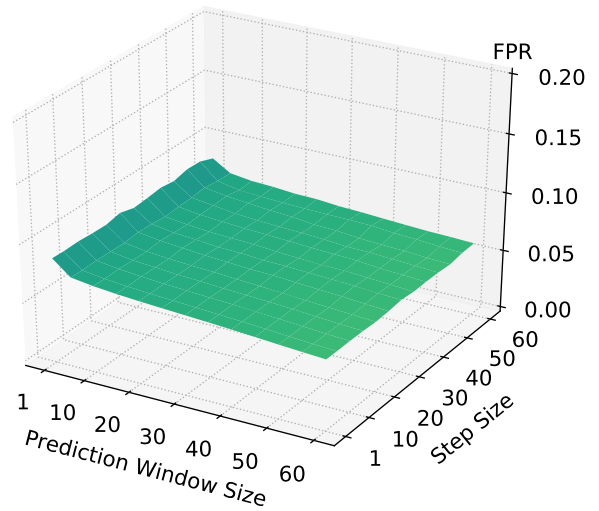
(a) FPR for the 5min observation windows.



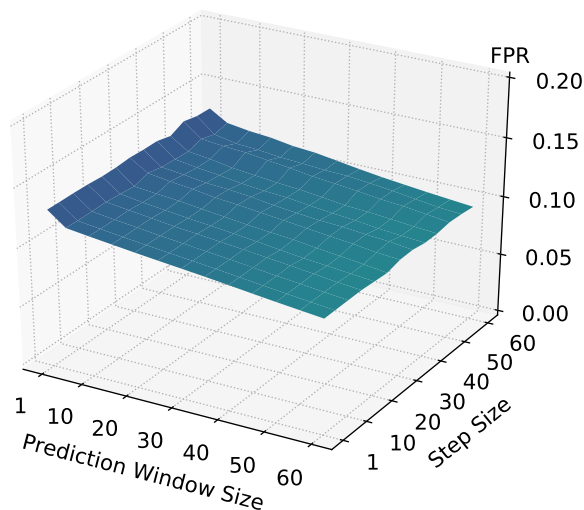
(b) FPR for the 10min observation windows.



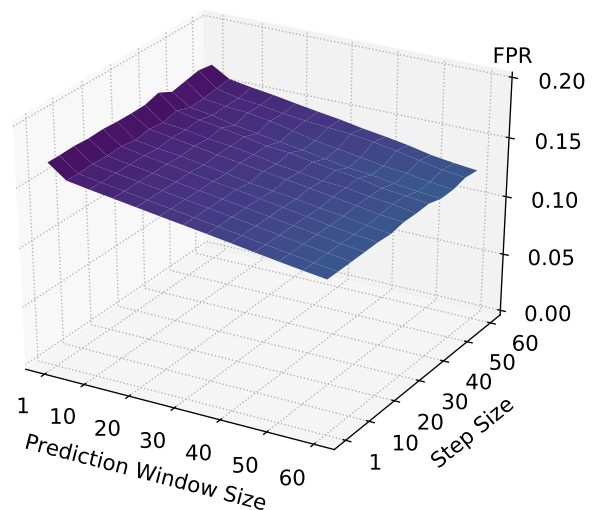
(c) FPR for the 15min observation windows.



(d) FPR for the 30min observation windows.

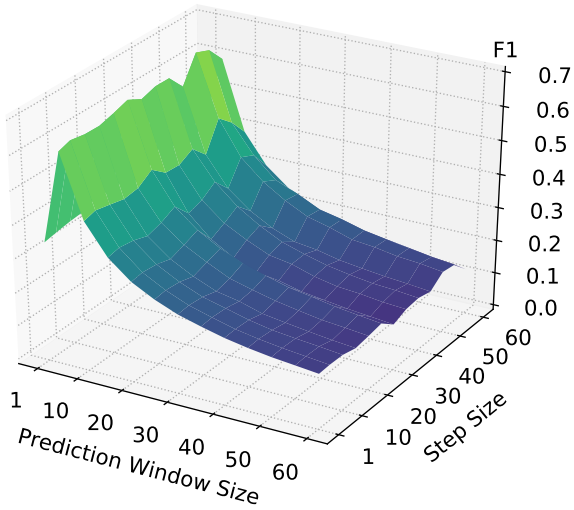


(e) FPR for the 45min observation windows.

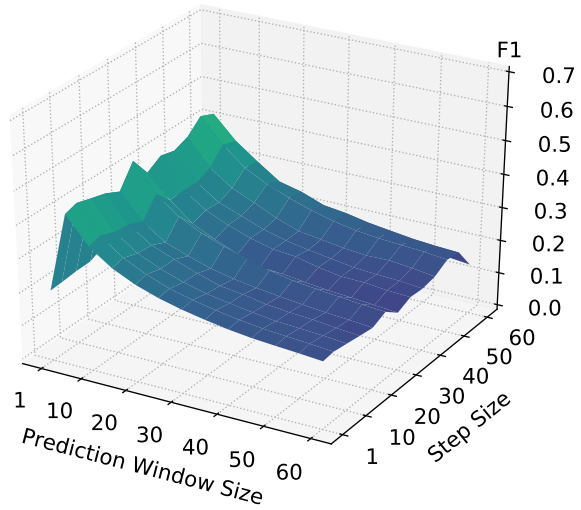


(f) FPR for the 60min observation windows.

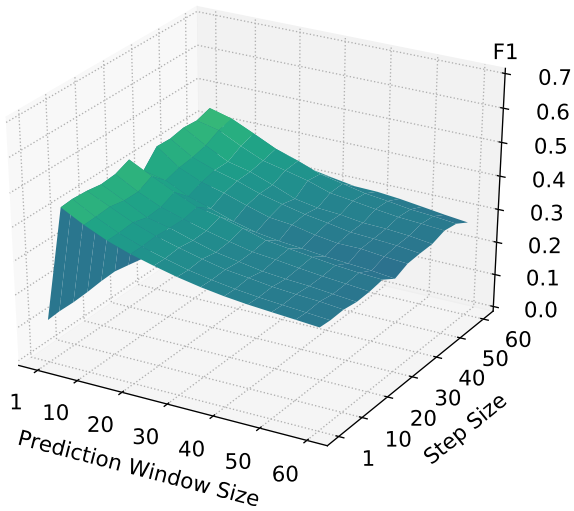
Figure C.16: 5-times augmented data: FPR for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.



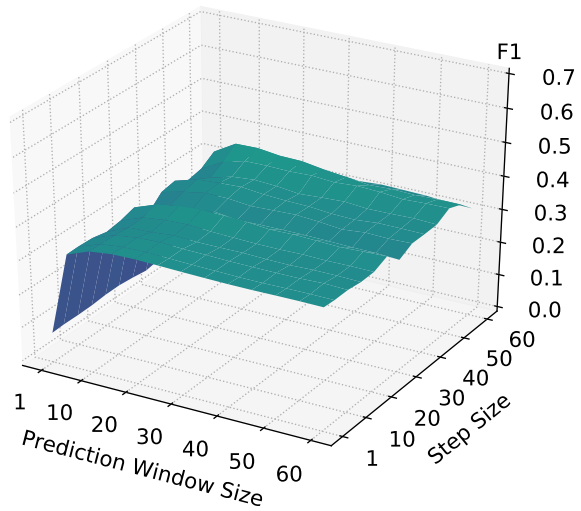
(a) F1 for the 5min observation windows.



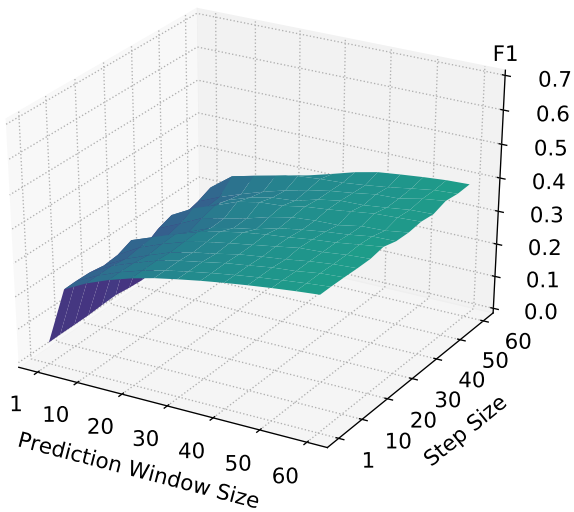
(b) F1 for the 10min observation windows.



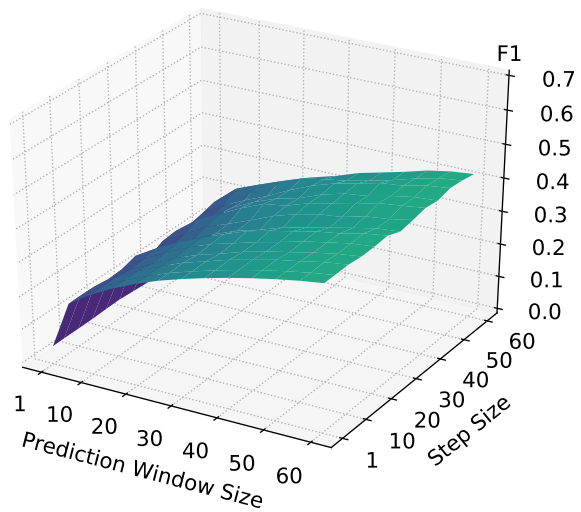
(c) F1 for the 15min observation windows.



(d) F1 for the 30min observation windows.

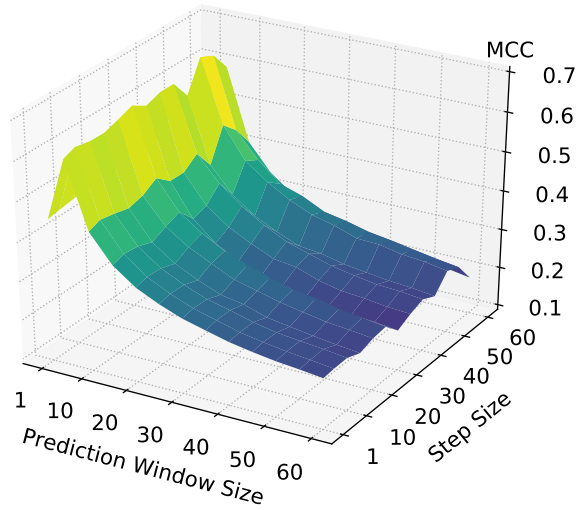


(e) F1 for the 45min observation windows.

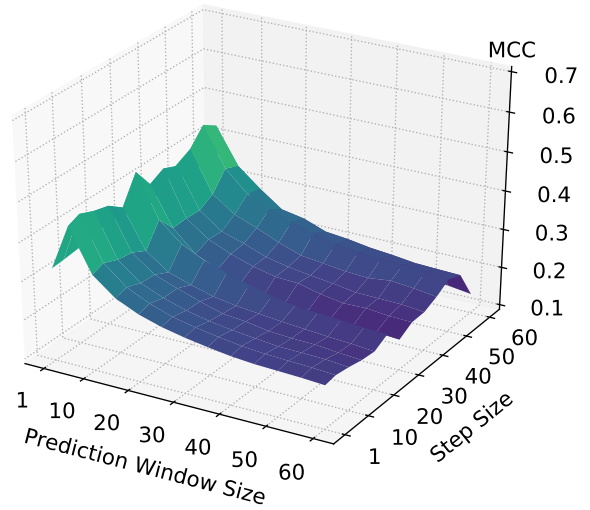


(f) F1 for the 60min observation windows.

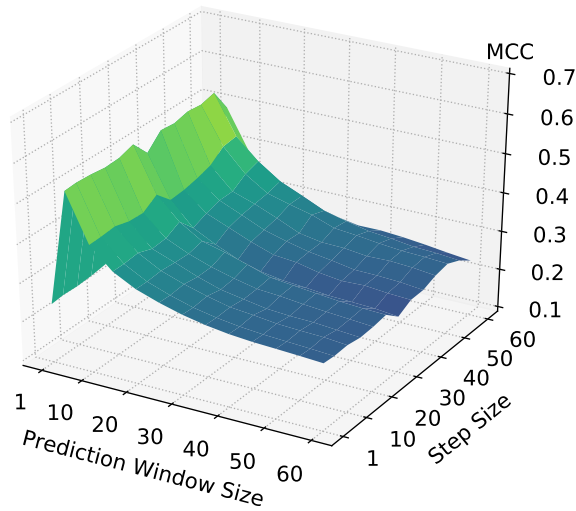
Figure C.17: 5-times augmented data: F1 for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.



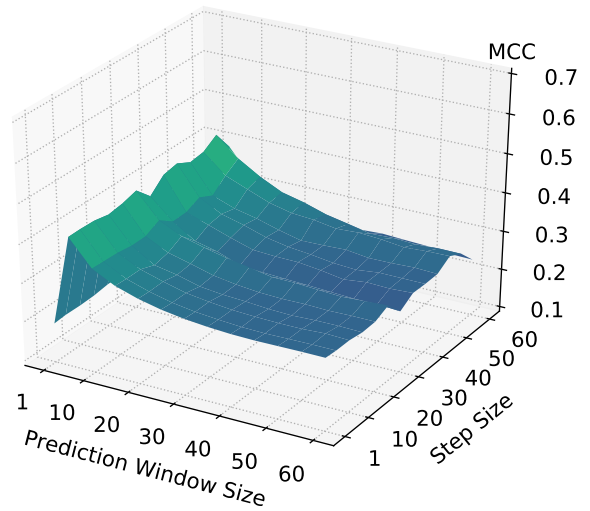
(a) MCC for the 5min observation windows.



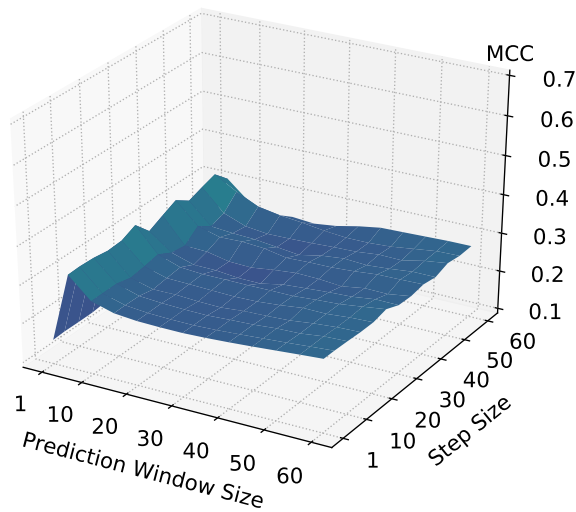
(b) MCC for the 10min observation windows.



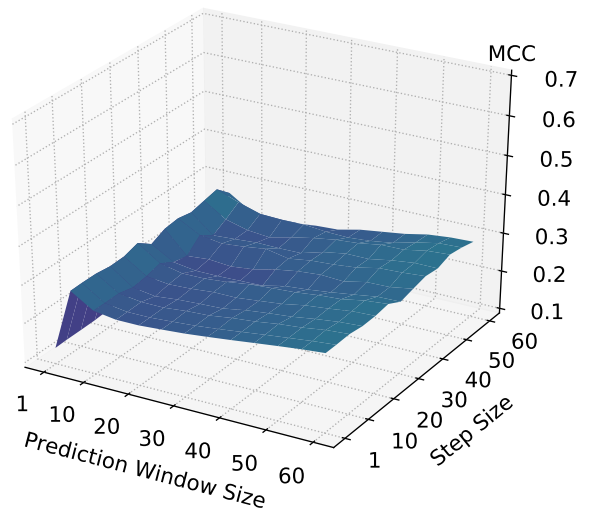
(c) MCC for the 15min observation windows.



(d) MCC for the 30min observation windows.

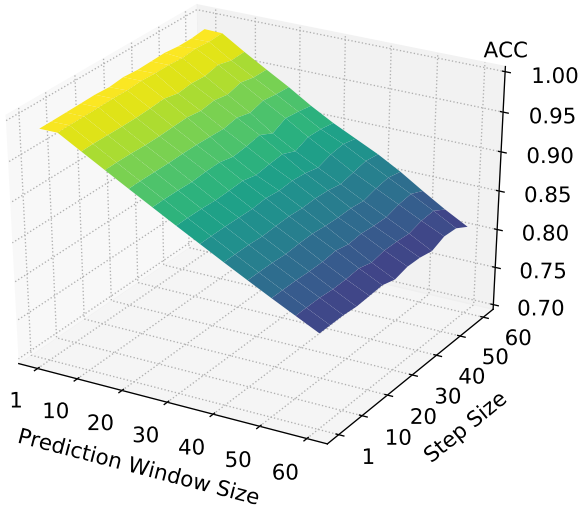


(e) MCC for the 45min observation windows.

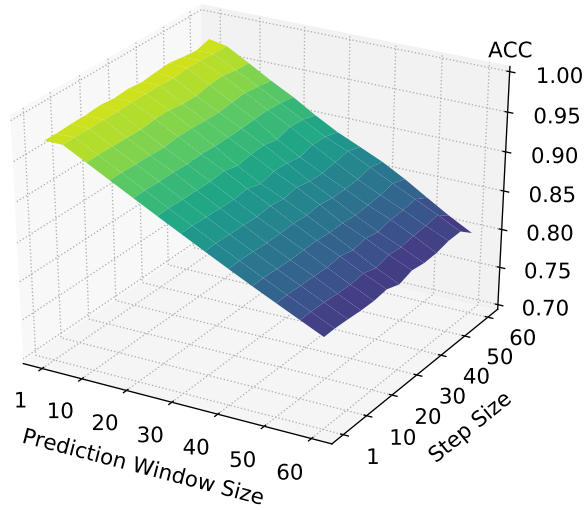


(f) MCC for the 60min observation windows.

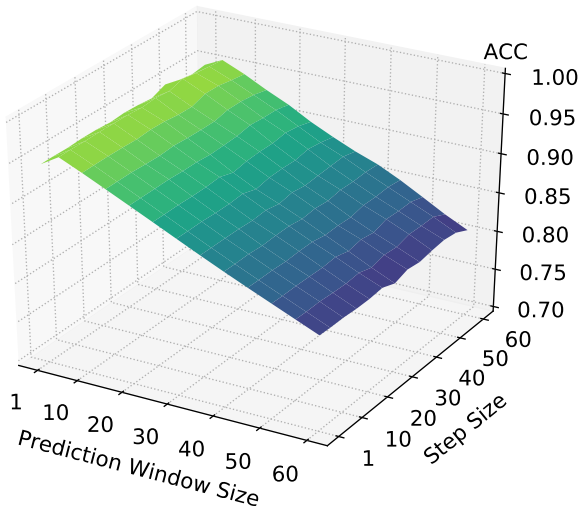
Figure C.18: 5-times augmented data: MCC for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.



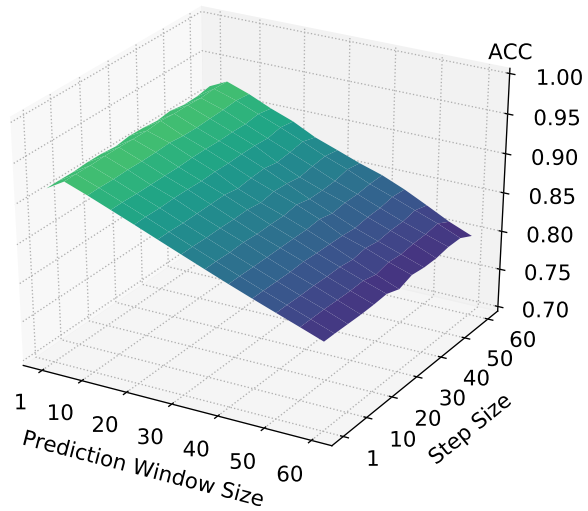
(a) ACC for the 5min observation windows.



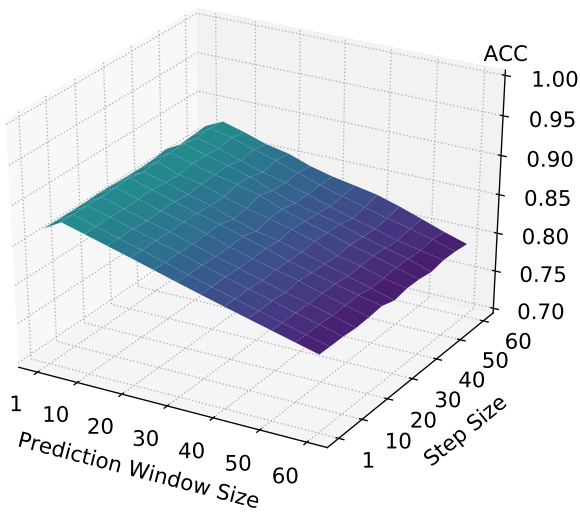
(b) ACC for the 10min observation windows.



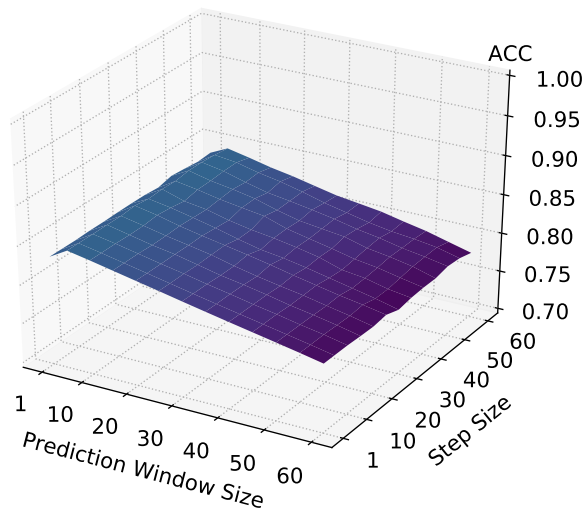
(c) ACC for the 15min observation windows.



(d) ACC for the 30min observation windows.

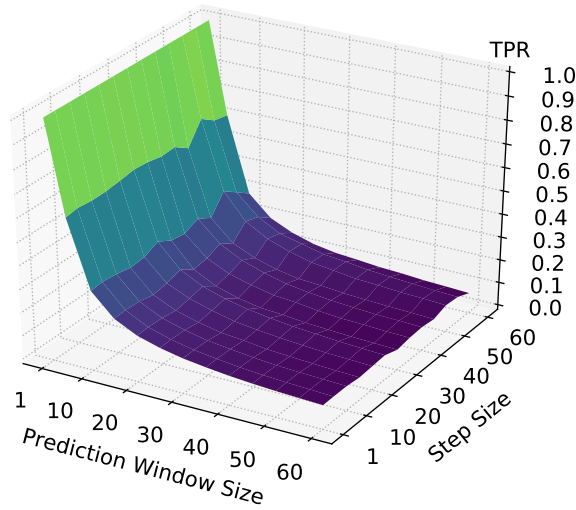


(e) ACC for the 45min observation windows.

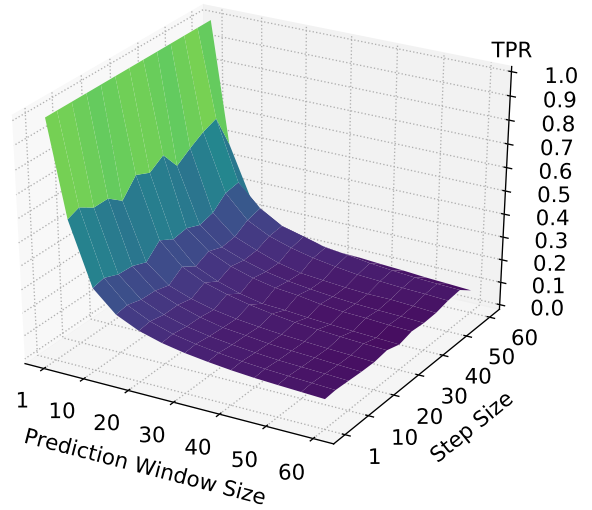


(f) ACC for the 60min observation windows.

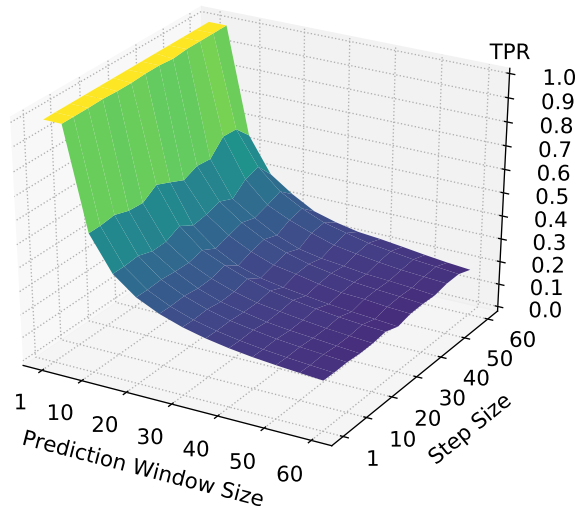
Figure C.19: 10-times augmented data: ACC for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.



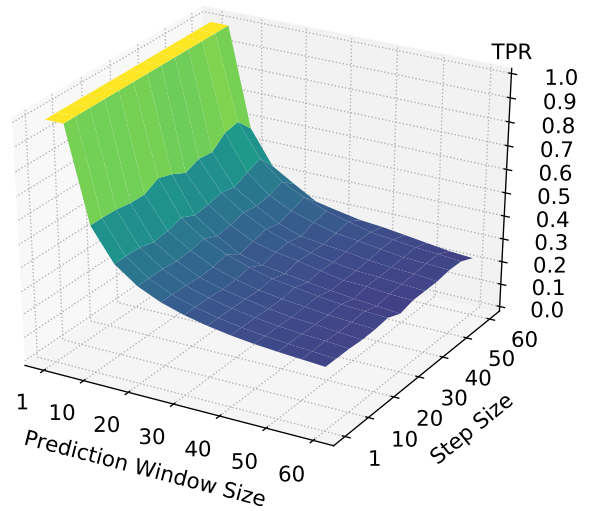
(a) TPR for the 5min observation windows.



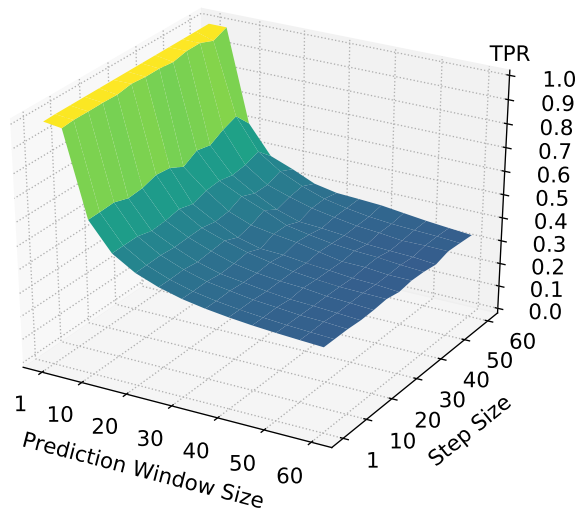
(b) TPR for the 10min observation windows.



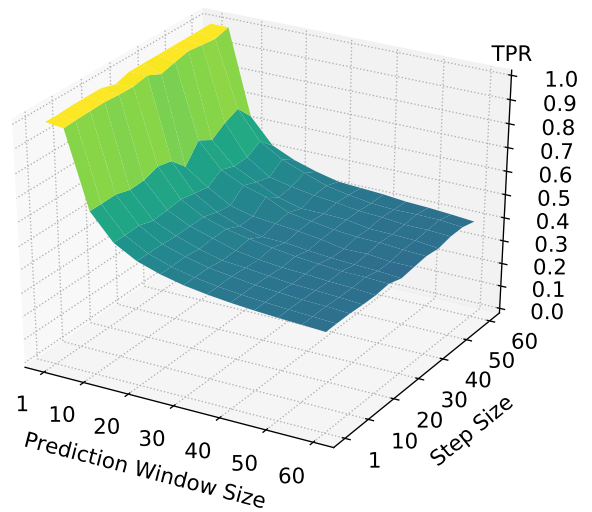
(c) TPR for the 15min observation windows.



(d) TPR for the 30min observation windows.

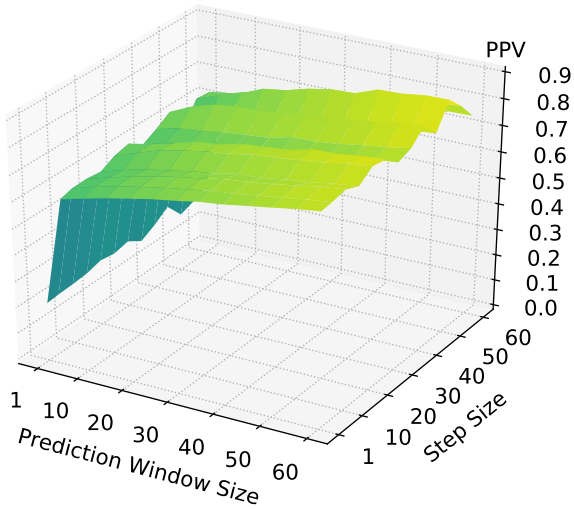


(e) TPR for the 45min observation windows.

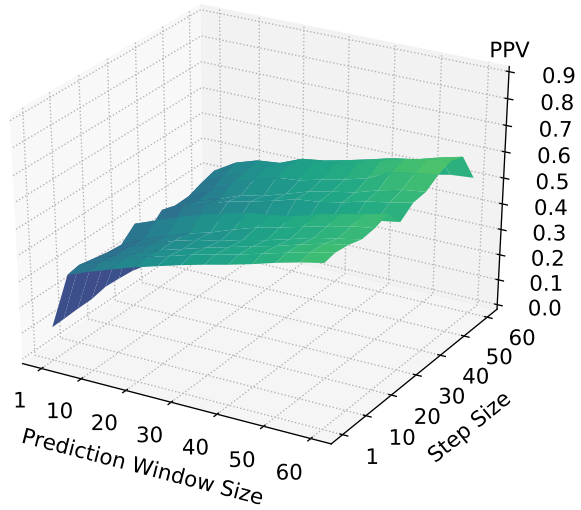


(f) TPR for the 60min observation windows.

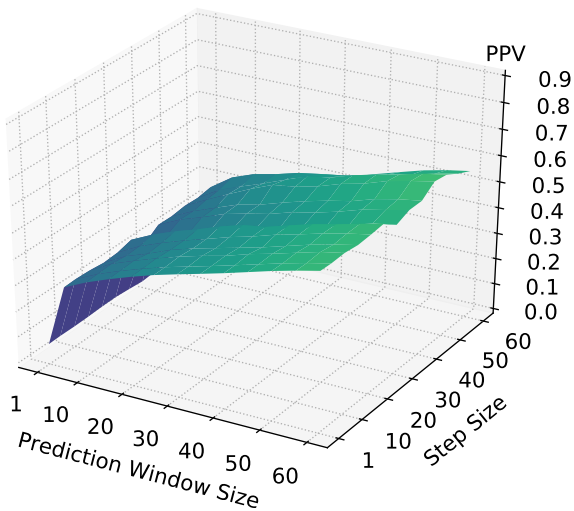
Figure C.20: 10-times augmented data: TPR for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.



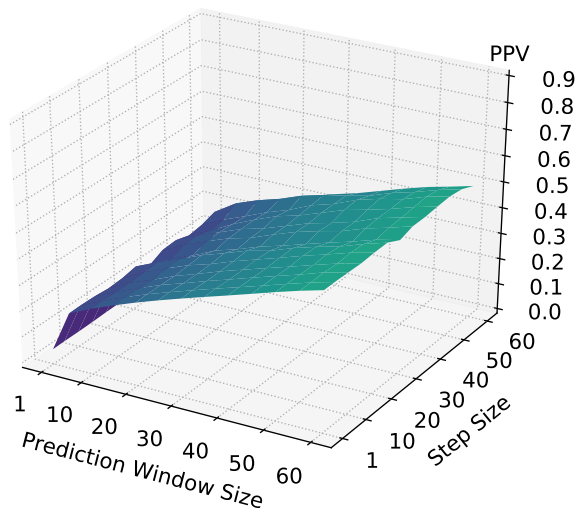
(a) PPV for the 5min observation windows.



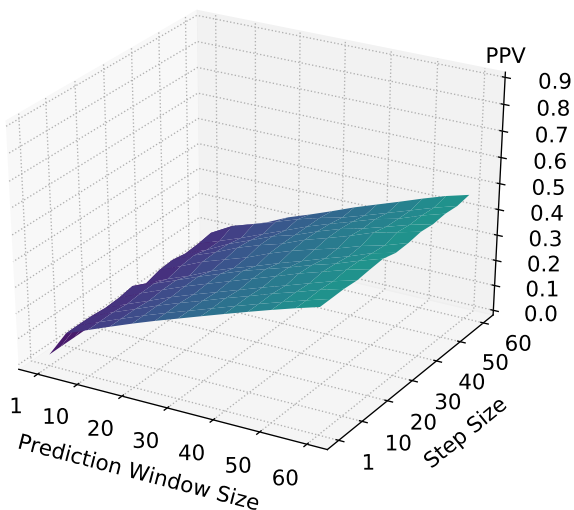
(b) PPV for the 10min observation windows.



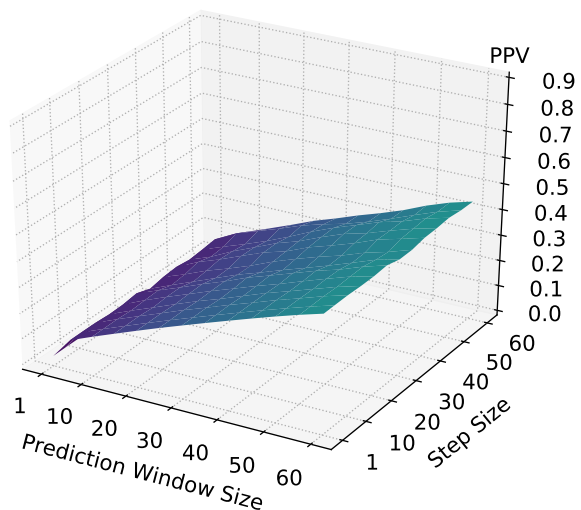
(c) PPV for the 15min observation windows.



(d) PPV for the 30min observation windows.

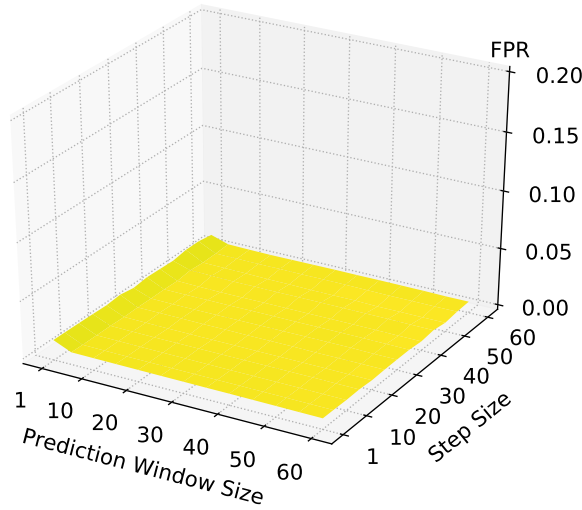


(e) PPV for the 45min observation windows.

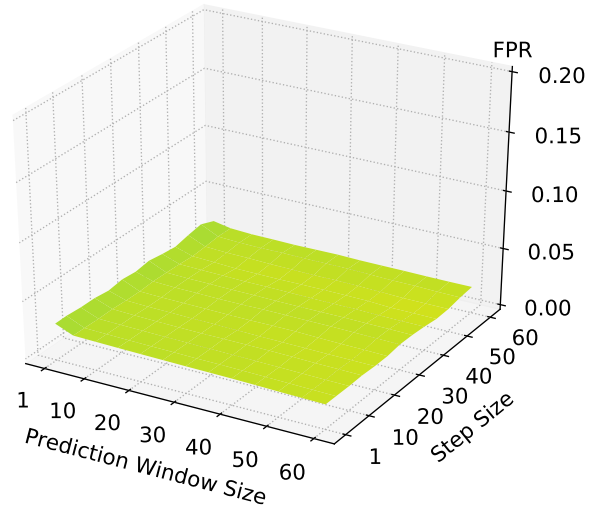


(f) PPV for the 60min observation windows.

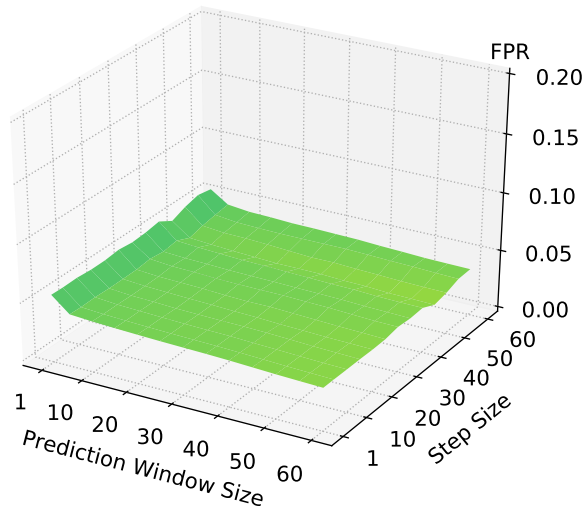
Figure C.21: 10-times augmented data: PPV for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.



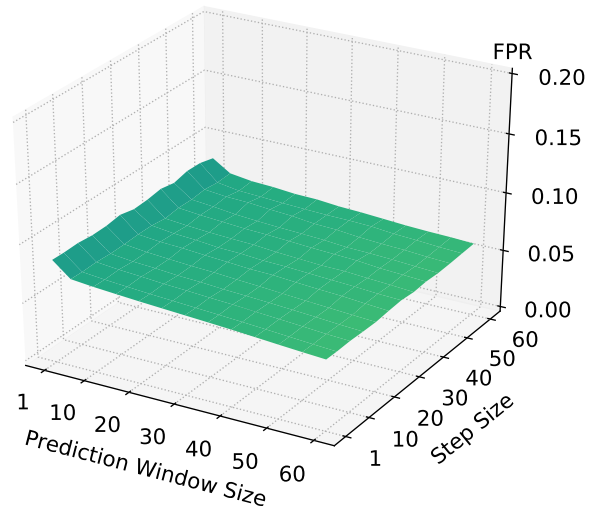
(a) FPR for the 5min observation windows.



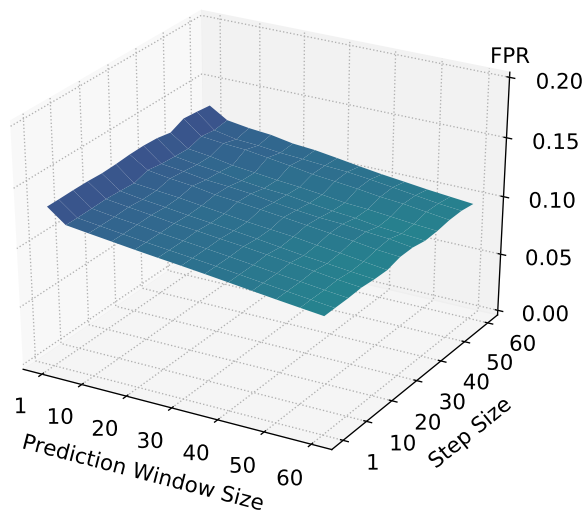
(b) FPR for the 10min observation windows.



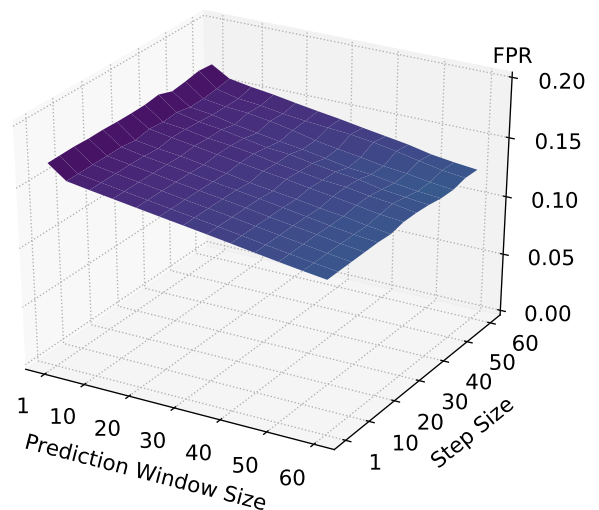
(c) FPR for the 15min observation windows.



(d) FPR for the 30min observation windows.

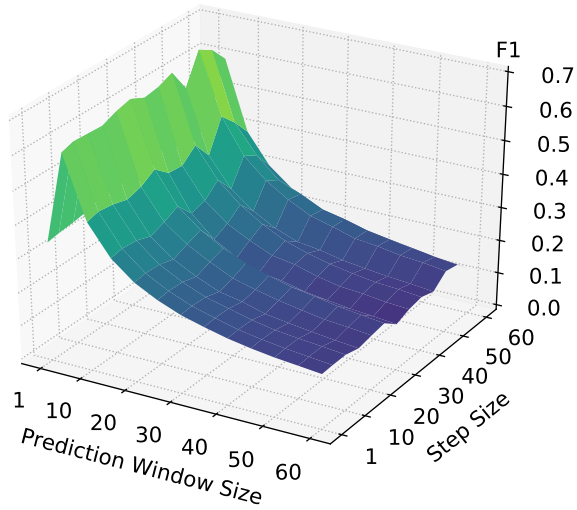


(e) FPR for the 45min observation windows.

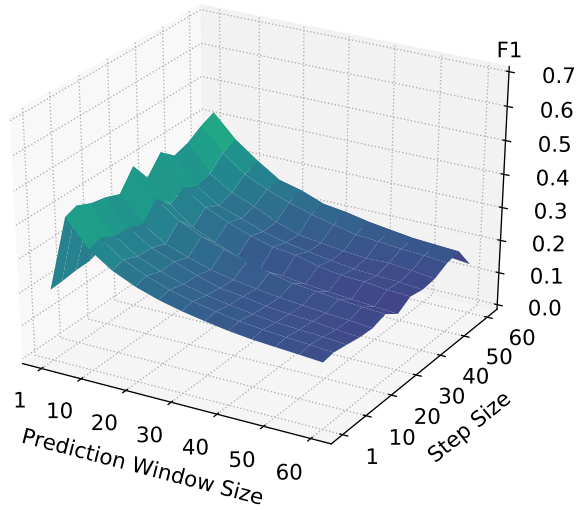


(f) FPR for the 60min observation windows.

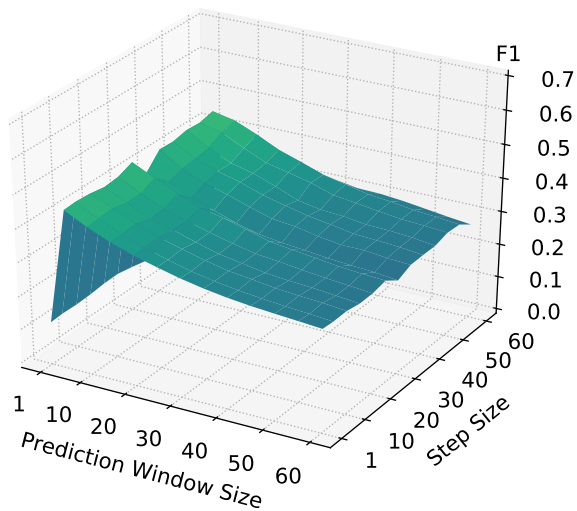
Figure C.22: 10-times augmented data: FPR for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.



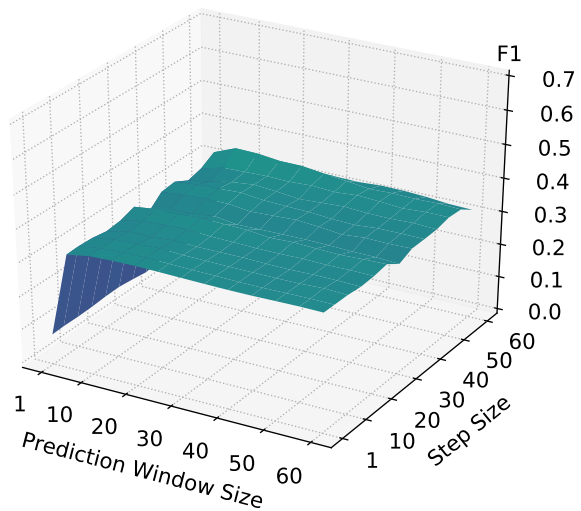
(a) F1 for the 5min observation windows.



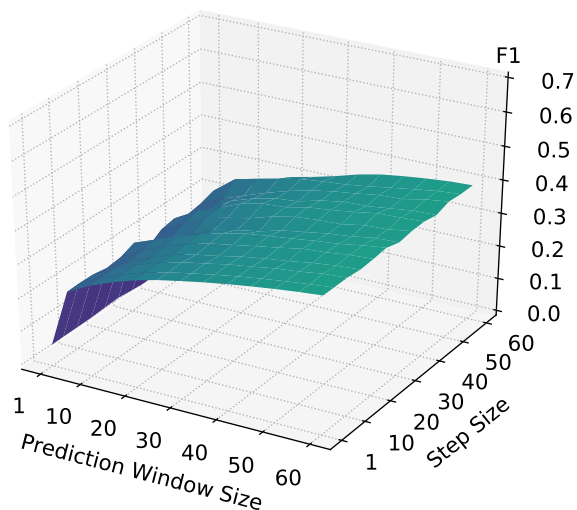
(b) F1 for the 10min observation windows.



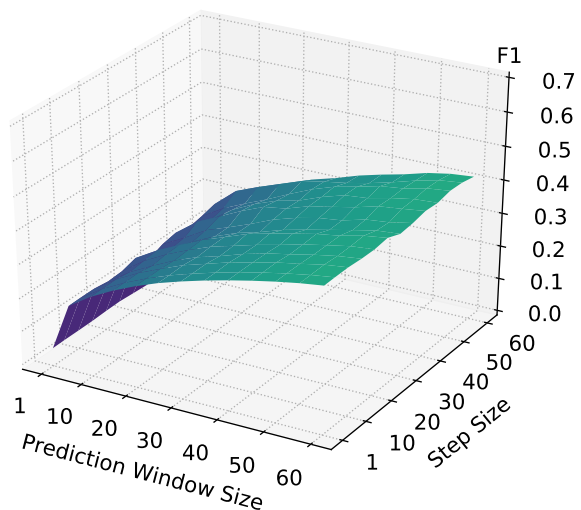
(c) F1 for the 15min observation windows.



(d) F1 for the 30min observation windows.

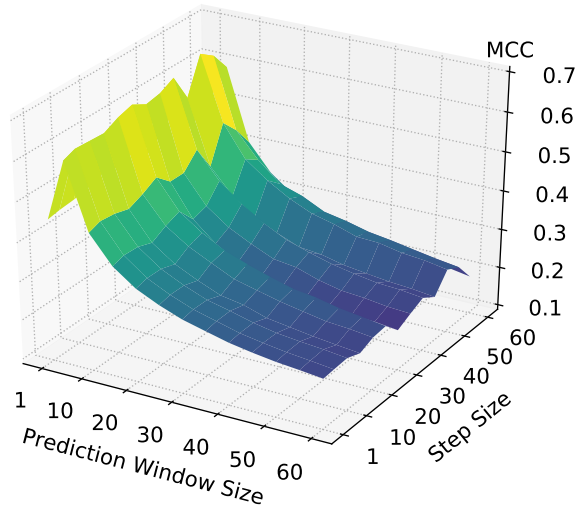


(e) F1 for the 45min observation windows.

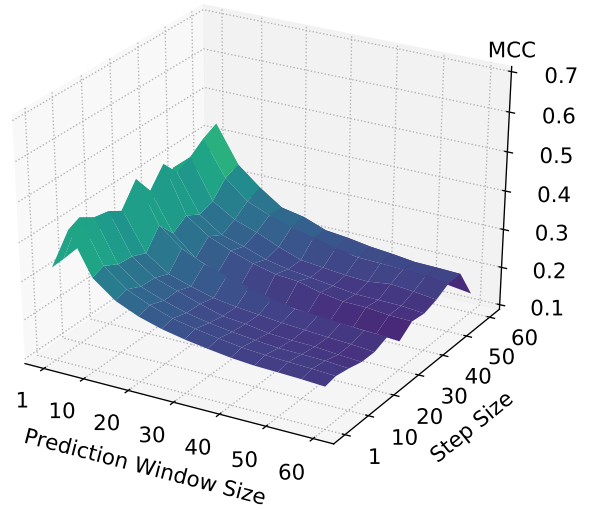


(f) F1 for the 60min observation windows.

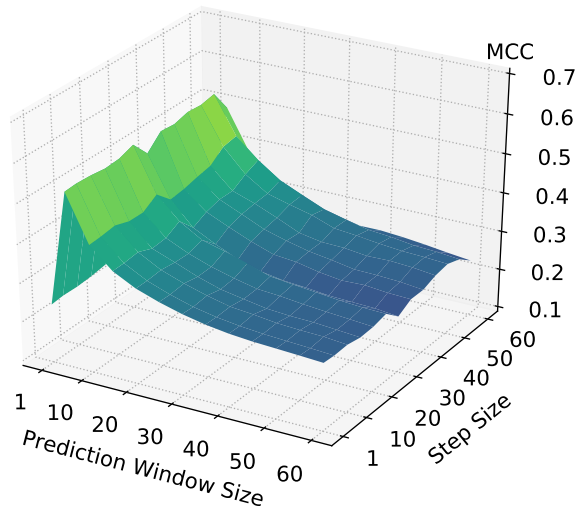
Figure C.23: 10-times augmented data: F1 for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.



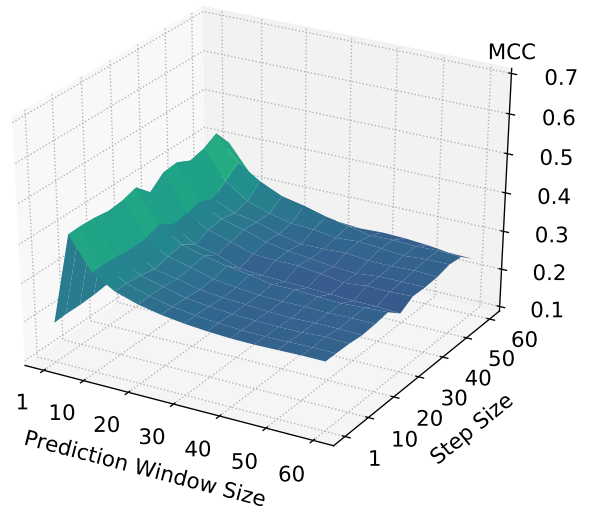
(a) MCC for the 5min observation windows.



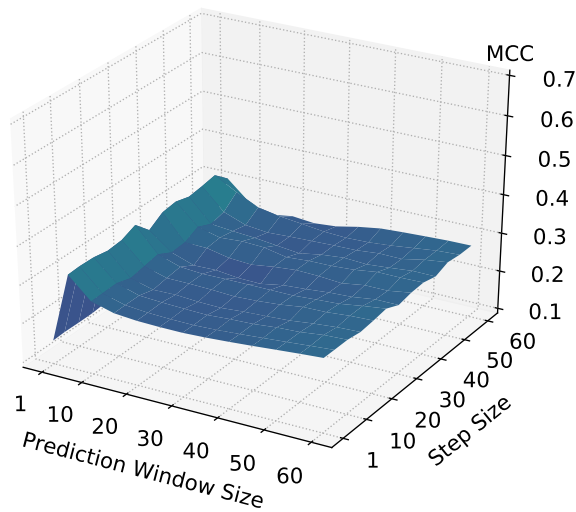
(b) MCC for the 10min observation windows.



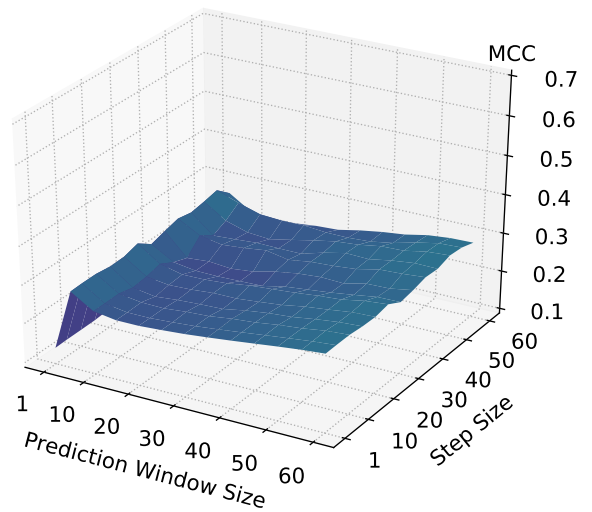
(c) MCC for the 15min observation windows.



(d) MCC for the 30min observation windows.



(e) MCC for the 45min observation windows.



(f) MCC for the 60min observation windows.

Figure C.24: 10-times augmented data: MCC for different observation window sizes after running slide-through sampling testing configurations with varying step sizes and prediction window sizes with models. Yellow tones indicate better values, purple worse values.

Appendix D

Time Series Clustering

In this appendix chapter, we provide additional results and figures that allow more detailed insights into the data, the approach and the evaluation presented in [Chapter 5 on p. 133](#).

D.1 Permutation Analysis Feature

One of our time series characteristics is a custom feature that is not taken from related work, namely `permutation_analysis`, which is a custom feature that calculates a score based on permutation. If a time series has certain temporal patterns (e.g., it has a trend or is periodic), the idea is that we destroy these patterns via permutation, whereas time series without temporal patterns are not/less affected by the permutation. The score is calculated by creating equally sized percentile-based thresholds of the time series, where the number of thresholds can be specified by the user. Then, the data is randomly permuted, and we count how often continuous signal segments are above these thresholds, which is repeated multiple times to get a confidence interval.

The example presented in [Figure D.1](#) demonstrates this confidence interval calculation based on a sinus-shaped time series with 100 data points (cf. [Figure D.1a](#)), i.e., a time series with a distinct temporal pattern. In the example, we set the number of percentile-based thresholds to 9, which results in the thresholds (10, 20, ..., 90), where each one represents the $i\%$ percentile of the time series data.¹ For instance, [Figure D.1b](#) shows the 30% percentile threshold for the original data. The next step is to count how many continuous time series segments are above this threshold. In the example, we have two such segments: the start segment of the time series (up until timestamp 15) and the second big segment at the end (starting from timestamp 40). We repeat this counting for every threshold. Afterwards, we randomly permute the original time series as shown in [Figure D.1c](#) and count again how many continuous time series segments are above the same thresholds. In the example permutation, there are a total of 19 such segments. Again, this is done for every threshold. Furthermore, we repeat the permutation procedure multiple times, which results in multiple counts per threshold, for which we can then calculate a confidence interval as visualized in [Figure D.1d](#). The lower and upper bounds of this interval are determined based on the α and $1 - \alpha$ percentiles of the counts, respectively, where α can be specified by the user. In the example, we repeat the permutation ten times (ten counts per threshold) and use $\alpha = 0.01$, i.e., the lower bound is the 1% percentile of the ten counts and the upper bound the 99% percentile of the ten counts.

¹The 0% and 100% thresholds are not used because the entire data either is above or below these two thresholds, respectively.

The last step is then to combine the counts of the original time series with the confidence interval of the permutation-based counts and calculate our final score, which is shown in [Figure D.1e](#). The score is the fraction of points outside the confidence interval, where values towards 1 indicate the presence of temporal patterns. In the example, the nice counts of the sinus-shaped time series fall zero times into the corresponding confidence interval counts, i.e., $\frac{9}{9}$ are outside the interval, which yields a score of 1.

We also provide a second example in [Figure D.2](#), where we use a random time series signal (cf. [Figure D.2a](#)) instead of the sinus-shape one, i.e., a time series without any temporal patterns. The new confidence interval and the counts of this original, random time series are shown in [Figure D.2b](#), where we can see that only a single count (at the 50% threshold) is outside the interval, which results in a score of $\frac{1}{9} \approx 0.11$, indicating that there is indeed no presence of temporal patterns.

D.2 Data Exploration

This section covers additional figures for the raw data we had at our disposal for evaluating our time series clustering approach.

In [Table D.1](#), all 128 UCR datasets are listed, including their name, the type of the time series as well as their length, the number of samples in the training set and test set, and the number of classes/labels. A detailed description of each dataset and its domain can be found in the official documentation [\[45\]](#). Moreover, for each training and test set, we specify whether the samples are approximately equally distributed among the different classes, i.e., whether the classes are balanced or unbalanced. ‘‘Approximately’’ means that we use a threshold t to check if the lowest class size similarity s is within this threshold to still consider the dataset balanced (b) or to treat it as unbalanced (u), which is defined in [Equation D.1](#):

$$\text{class balance} = \begin{cases} \text{balanced (b)} & \text{if } s \geq t \\ \text{unbalanced (u)} & \text{if } s < t \end{cases} \quad \text{with } s = \frac{\min(\text{class sizes})}{\max(\text{class sizes})} \quad (\text{D.1})$$

We chose a threshold of $t = 0.9$, meaning that a dataset is considered to be balanced if the smallest class still has at least 90% of the number of samples of the largest class. For example, a dataset with three class sizes $\{29, 32, 31\}$ would be balanced ($\frac{29}{32} \approx 0.91 \geq t = 0.9$), whereas if it had the class sizes $\{20, 32, 31\}$, it would be unbalanced ($\frac{20}{32} \approx 0.63 < t = 0.9$). 63 (49%) of the 128 UCR datasets thus have a balanced training set and 59 (46%) a balanced test set.

Name	Type	#Train (b/u)	#Test (b/u)	#C	Length
ACSF1	Device	100 (b)	100 (b)	10	1460
Adiac	Image	390 (u)	391 (u)	37	176
AllGestureWiiimoteX	Sensor	300 (b)	700 (b)	10	500
AllGestureWiiimoteY	Sensor	300 (b)	700 (b)	10	500
AllGestureWiiimoteZ	Sensor	300 (b)	700 (b)	10	500
ArrowHead	Image	36 (b)	175 (u)	3	251
Beef	Spectro	30 (b)	30 (b)	5	470
BeetleFly	Image	20 (b)	20 (b)	2	512
BirdChicken	Image	20 (b)	20 (b)	2	512
BME	Simulated	30 (b)	150 (b)	3	128
Car	Sensor	60 (u)	60 (u)	4	577
CBF	Simulated	30 (u)	900 (b)	3	128

Name	Type	#Train (b/u)	#Test (b/u)	#C	Length
Chinatown	Traffic	20 (b)	343 (u)	2	24
ChlorineConcentration	Sensor	467 (u)	3840 (u)	3	166
CinCECGTorso	Sensor	40 (u)	1380 (b)	4	1639
Coffee	Spectro	28 (b)	28 (u)	2	286
Computers	Device	250 (b)	250 (b)	2	720
CricketX	Motion	390 (u)	390 (u)	12	300
CricketY	Motion	390 (u)	390 (u)	12	300
CricketZ	Motion	390 (u)	390 (u)	12	300
Crop	Image	7200 (b)	16800 (b)	24	46
DiatomSizeReduction	Image	16 (u)	306 (u)	4	345
DistalPhalanxOutlineAgeGroup	Image	400 (u)	139 (u)	3	80
DistalPhalanxOutlineCorrect	Image	600 (u)	276 (u)	2	80
DistalPhalanxTW	Image	400 (u)	139 (u)	6	80
DodgerLoopDay	Sensor	78 (u)	80 (u)	7	288
DodgerLoopGame	Sensor	20 (b)	138 (b)	2	288
DodgerLoopWeekend	Sensor	20 (b)	138 (u)	2	288
Earthquakes	Sensor	322 (u)	139 (u)	2	512
ECG200	ECG	100 (u)	100 (u)	2	96
ECG5000	ECG	500 (u)	4500 (u)	5	140
ECGFiveDays	ECG	23 (u)	861 (b)	2	136
ElectricDevices	Device	8926 (u)	7711 (u)	7	96
EOGHorizontalSignal	EOG	362 (b)	362 (b)	12	1250
EOGVerticalSignal	EOG	362 (b)	362 (b)	12	1250
EthanolLevel	Spectro	504 (b)	500 (b)	4	1751
FaceAll	Image	560 (b)	1690 (u)	14	131
FaceFour	Image	24 (u)	88 (u)	4	350
FacesUCR	Image	200 (u)	2050 (u)	14	131
FiftyWords	Image	450 (u)	455 (u)	50	270
Fish	Image	175 (u)	175 (u)	7	463
FordA	Sensor	3601 (b)	1320 (b)	2	500
FordB	Sensor	3636 (b)	810 (b)	2	500
FreezerRegularTrain	Sensor	150 (b)	2850 (b)	2	301
FreezerSmallTrain	Sensor	28 (b)	2850 (b)	2	301
Fungi	HRM	18 (b)	186 (u)	18	201
GestureMidAirD1	Trajectory	208 (b)	130 (b)	26	360
GestureMidAirD2	Trajectory	208 (b)	130 (b)	26	360
GestureMidAirD3	Trajectory	208 (b)	130 (b)	26	360
GesturePebbleZ1	Sensor	132 (u)	172 (u)	6	455
GesturePebbleZ2	Sensor	146 (u)	158 (u)	6	455
GunPoint	Motion	50 (b)	150 (b)	2	150
GunPointAgeSpan	Motion	135 (b)	316 (b)	2	150
GunPointMaleVersusFemale	Motion	135 (b)	316 (b)	2	150
GunPointOldVersusYoung	Motion	136 (b)	315 (b)	2	150
Ham	Spectro	109 (b)	105 (b)	2	431
HandOutlines	Image	1000 (u)	370 (u)	2	2709
Haptics	Motion	155 (u)	308 (u)	5	1092
Herring	Image	64 (u)	64 (u)	2	512
HouseTwenty	Device	40 (b)	119 (u)	2	2000
InlineSkate	Motion	100 (u)	550 (u)	7	1882
InsectEPGRegularTrain	EPG	62 (u)	249 (u)	3	601
InsectEPGSmallTrain	EPG	17 (u)	249 (u)	3	601
InsectWingbeatSound	Sensor	220 (b)	1980 (b)	11	256
ItalyPowerDemand	Sensor	67 (b)	1029 (b)	2	24
LargeKitchenAppliances	Device	375 (b)	375 (b)	3	720

Name	Type	#Train (b/u)	#Test (b/u)	#C	Length
Lightning2	Sensor	60 (u)	61 (u)	2	637
Lightning7	Sensor	70 (u)	73 (u)	7	319
Mallat	Simulated	55 (u)	2345 (b)	8	1024
Meat	Spectro	60 (b)	60 (b)	3	448
MedicalImages	Image	381 (u)	760 (u)	10	99
MelbournePedestrian	Traffic	1194 (b)	2439 (b)	10	24
MiddlePhalanxOutlineAgeGroup	Image	400 (u)	154 (u)	3	80
MiddlePhalanxOutlineCorrect	Image	600 (u)	291 (u)	2	80
MiddlePhalanxTW	Image	399 (u)	154 (u)	6	80
MixedShapesRegularTrain	Image	500 (b)	2425 (u)	5	1024
MixedShapesSmallTrain	Image	100 (b)	2425 (u)	5	1024
MoteStrain	Sensor	20 (b)	1252 (u)	2	84
NonInvasiveFetalECGThorax1	ECG	1800 (u)	1965 (u)	42	750
NonInvasiveFetalECGThorax2	ECG	1800 (u)	1965 (u)	42	750
OliveOil	Spectro	30 (u)	30 (u)	4	570
OSULeaf	Image	200 (u)	242 (u)	6	427
PhalangesOutlinesCorrect	Image	1800 (u)	858 (u)	2	80
Phoneme	Sensor	214 (u)	1896 (u)	39	1024
PickupGestureWiimoteZ	Sensor	50 (b)	50 (b)	10	361
PigAirwayPressure	Hemodynamics	104 (b)	208 (b)	52	2000
PigArtPressure	Hemodynamics	104 (b)	208 (b)	52	2000
PigCVP	Hemodynamics	104 (b)	208 (b)	52	2000
PLAID	Device	537 (u)	537 (u)	11	1344
Plane	Sensor	105 (u)	105 (u)	7	144
PowerCons	Power	180 (b)	180 (b)	2	144
ProximalPhalanxOutlineAgeGroup	Image	400 (u)	205 (u)	3	80
ProximalPhalanxOutlineCorrect	Image	600 (u)	291 (u)	2	80
ProximalPhalanxTW	Image	400 (u)	205 (u)	6	80
RefrigerationDevices	Device	375 (b)	375 (b)	3	720
Rock	Spectrum	20 (b)	50 (u)	4	2844
ScreenType	Device	375 (b)	375 (b)	3	720
SemgHandGenderCh2	Spectrum	300 (b)	600 (u)	2	1500
SemgHandMovementCh2	Spectrum	450 (b)	450 (b)	6	1500
SemgHandSubjectCh2	Spectrum	450 (b)	450 (b)	5	1500
ShakeGestureWiimoteZ	Sensor	50 (b)	50 (b)	10	385
ShapeletSim	Simulated	20 (b)	180 (b)	2	500
ShapesAll	Image	600 (b)	600 (b)	60	512
SmallKitchenAppliances	Device	375 (b)	375 (b)	3	720
SmoothSubspace	Simulated	150 (b)	150 (b)	3	15
SonyAIBORobotSurface1	Sensor	20 (u)	601 (u)	2	70
SonyAIBORobotSurface2	Sensor	27 (u)	953 (u)	2	65
StarLightCurves	Sensor	1000 (u)	8236 (u)	3	1024
Strawberry	Spectro	613 (u)	370 (u)	2	235
SwedishLeaf	Image	500 (u)	625 (u)	15	128
Symbols	Image	25 (u)	995 (u)	6	398
SyntheticControl	Simulated	300 (b)	300 (b)	6	60
ToeSegmentation1	Motion	40 (b)	228 (b)	2	277
ToeSegmentation2	Motion	36 (b)	130 (u)	2	343
Trace	Sensor	100 (u)	100 (u)	4	275
TwoLeadECG	ECG	23 (b)	1139 (b)	2	82
TwoPatterns	Simulated	1000 (u)	4000 (b)	4	128
UMD	Simulated	36 (b)	144 (b)	3	150
UWaveGestureLibraryAll	Motion	896 (u)	3582 (b)	8	945
UWaveGestureLibraryX	Motion	896 (u)	3582 (b)	8	315

Name	Type	#Train (b/u)	#Test (b/u)	#C	Length
UWaveGestureLibraryY	Motion	896 (u)	3582 (b)	8	315
UWaveGestureLibraryZ	Motion	896 (u)	3582 (b)	8	315
Wafer	Sensor	1000 (u)	6164 (u)	2	152
Wine	Spectro	57 (b)	54 (b)	2	234
WordSynonyms	Image	267 (u)	638 (u)	25	270
Worms	Motion	181 (u)	77 (u)	5	900
WormsTwoClass	Motion	181 (u)	77 (u)	2	900
Yoga	Image	300 (u)	3000 (u)	2	426

Table D.1: Detailed information on the 128 UCR datasets. $\#Train/Test$ (b/u) represents the number of train/test samples and whether their distribution among the $\#C$ different classes is balanced (b) or unbalanced (u). *Length* indicates the time series length.

D.3 Evaluation Results

We present the complete variant differences for all models and datasets, and then we proceed with the clustering results for all unlabeled datasets that we did not show in the evaluation.

D.3.1 Variant Differences

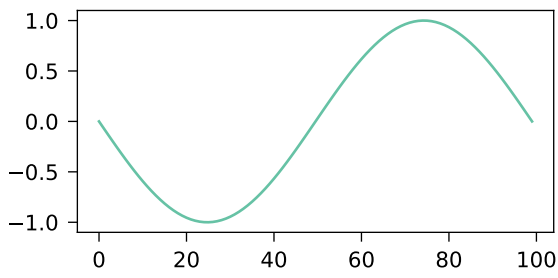
This section covers all variant differences, grouped by the four datasets and by the evaluated feature sets. [Figure D.3](#) shows the variant differences for the UCR dataset, [Figure D.4](#) for the UCR-merged dataset, [Figure D.5](#) for the IMTS₁ dataset and [Figure D.6](#) for the IMTS₂ dataset. In all figures, the column variants are omitted to create more compact matrix representations. However, they are identical to the row variants, since the matrices are symmetric in this regard, i.e., the omitted column variants are identical to the corresponding row variants.

D.3.2 Clustering Unlabeled Data

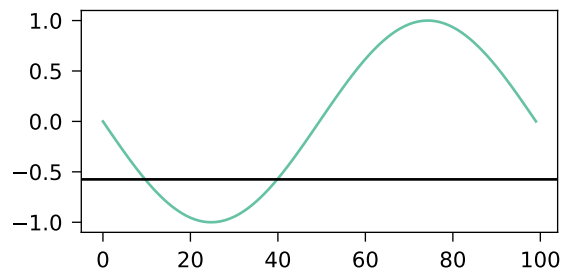
In this section, we present all results on the remaining unlabeled datasets (cf. [Table 5.5 on p. 158](#)) we did not cover in the evaluation, which, unless explicitly stated otherwise, were all obtained by using our selected clustering method `linkage|distributional|clip01_drop`. Furthermore, we include figures for additional information, which are t-SNE visualizations of the *distributional* feature set and the cluster feature values showing all TSC (sub)group features².

[Figure D.7](#) shows the t-SNE visualization and cluster feature values for the CPU Idle (H-01) metric of the IMTS₁ dataset. [Figure D.8](#) also shows these two figures as well as the Venn diagram for the original two clusters within the Memory Available % (H-07) metric of the IMTS₁ dataset, and [Figure D.9](#) displays the t-SNE visualization and cluster feature values when partitioning the same data into six new clusters. In [Figure D.10](#), [Figure D.11](#) and [Figure D.12](#), we present all clustering results for the Page Faults (H-06), Disk Available % (D-03) and Bytes Received (N-01) metrics of the IMTS₁ dataset, respectively.

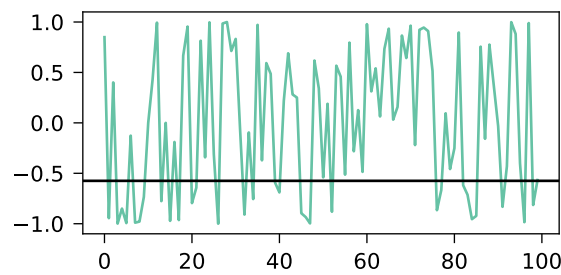
²The clusters were still obtained with the *distributional* features, the TSC feature values were then calculated afterwards for each resulting cluster.



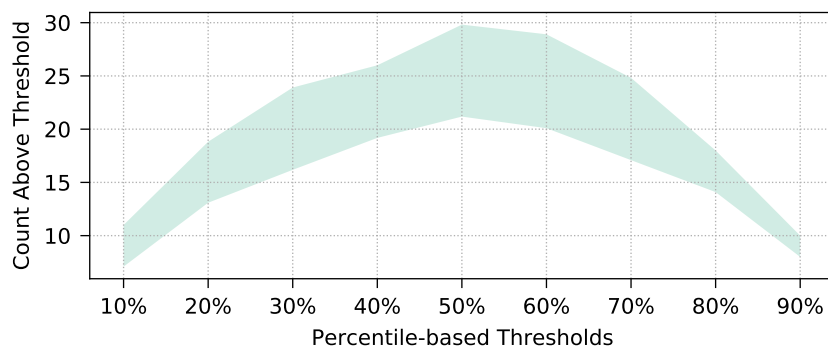
(a) Sinus-shaped time series with 100 data points.



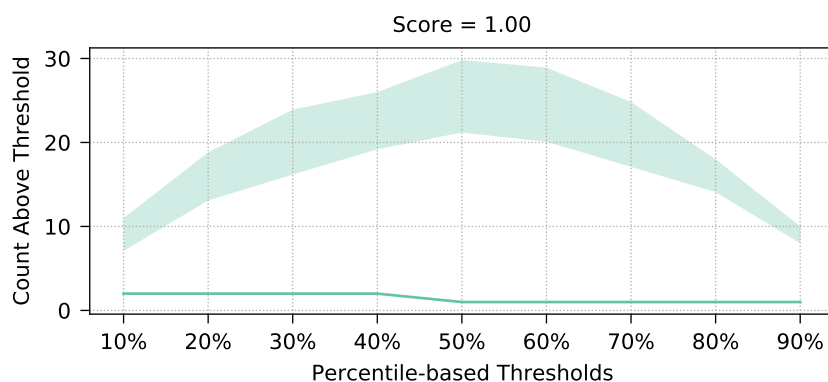
(b) The 30% percentile threshold for the sinus-shaped time series, where two continuous segments are above this threshold.



(c) A random permutation of the sinus-shaped time series with the same 30% percentile threshold as shown in [Figure D.1b](#). Here, 19 continuous segments are above this threshold.

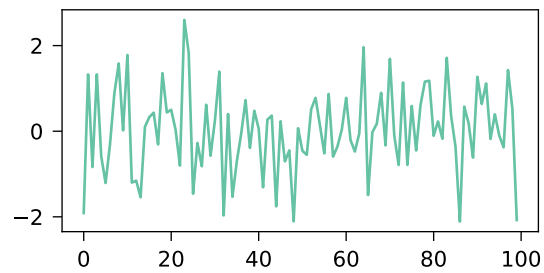


(d) The confidence interval of the counts obtained after repeating the permutation step ten times, where *Count Above Threshold* indicates the number of continuous time series segments that are above the corresponding threshold.

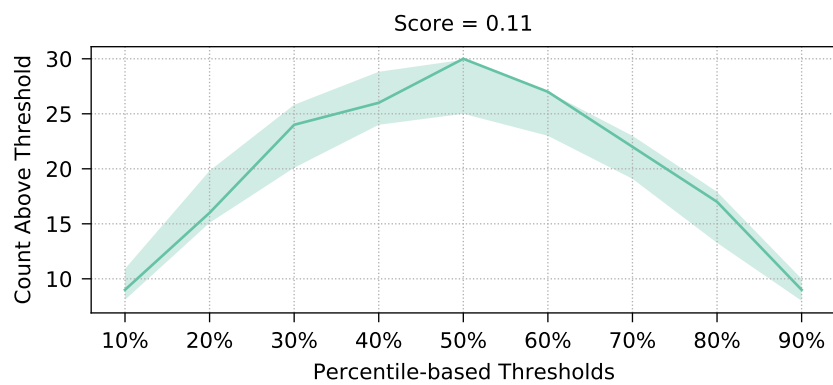


(e) The counts of the original, sinus-shaped time series compared to the confidence interval of the permutation-based counts. Here, the counts do not overlap with the interval, which results in a score of 1, thus indicating the presence of temporal patterns.

Figure D.1: Example of the steps and the final score returned by our custom feature `permutation_analysis` (using nine thresholds) when applied on a sinus-shaped time series.

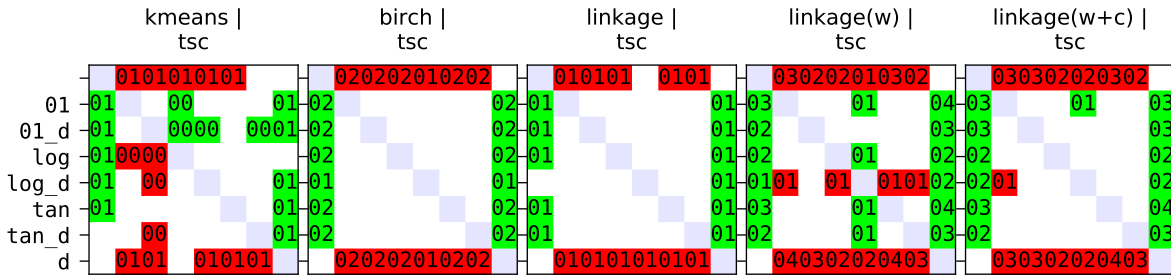


(a) Random time series with 100 data points.

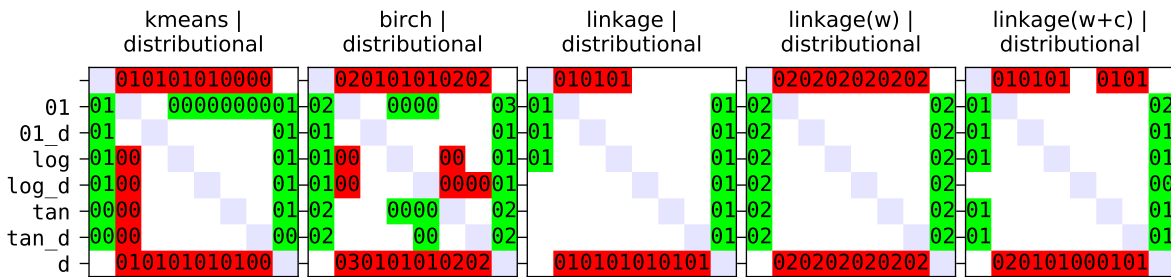


(b) The counts of the original, random time series compared to the confidence interval of the permutation-based counts. Here, most of the counts overlap with the interval (only the count at the 50% threshold is outside the interval), which results in a score of 0.11, thus indicating the absence of temporal patterns.

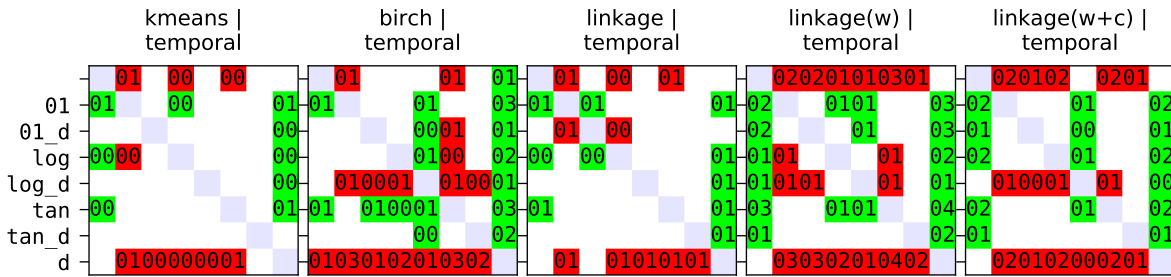
Figure D.2: Example of the final score returned by our custom feature `permutation_analysis` (using nine thresholds) when applied on a random time series.



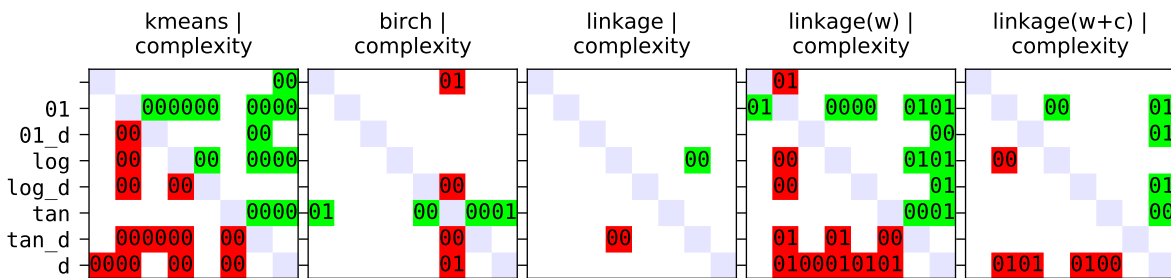
(a) Variant differences for dataset UCR and feature set *tsc*.



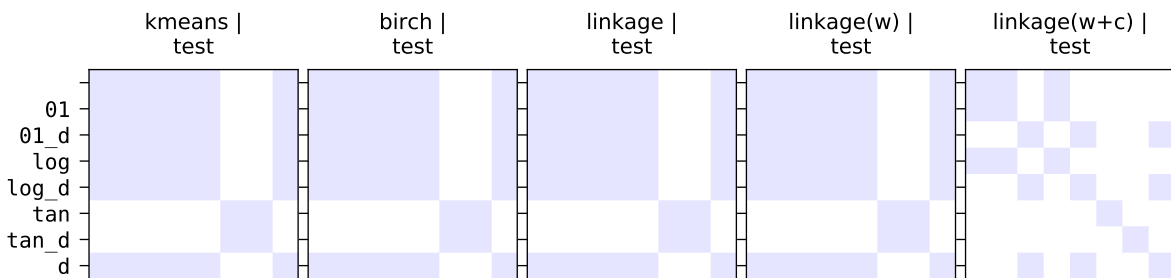
(b) Variant differences for dataset UCR and feature set *distributional*.



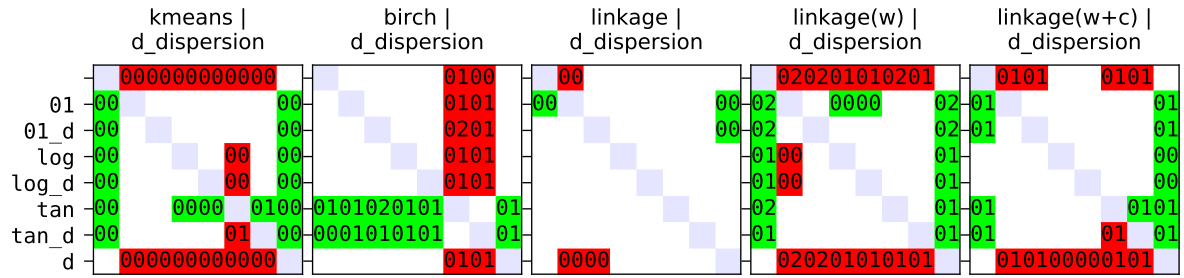
(c) Variant differences for dataset UCR and feature set *temporal*.



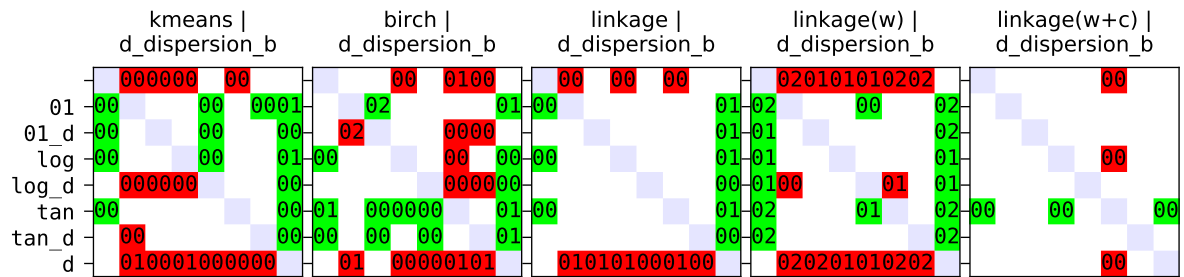
(d) Variant differences for dataset UCR and feature set *complexity*.



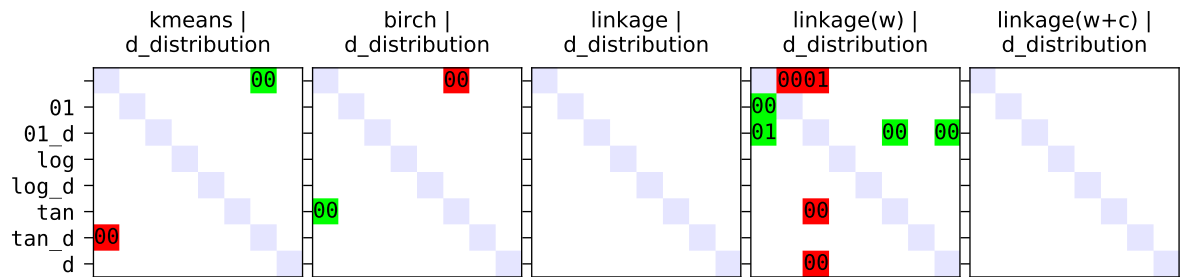
(e) Variant differences for dataset UCR and feature set *test*.



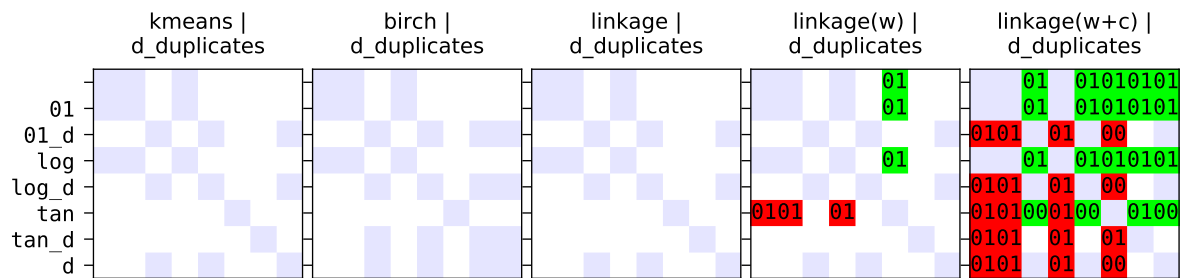
(f) Variant differences for dataset UCR and feature set $d_dispersion$.



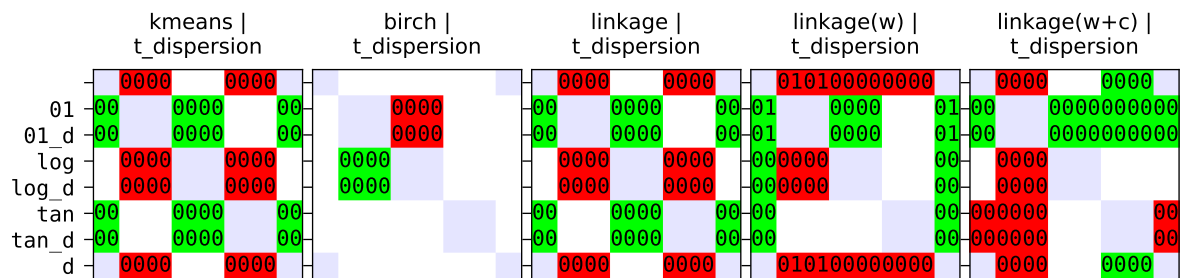
(g) Variant differences for dataset UCR and feature set $d_dispersion_b$.



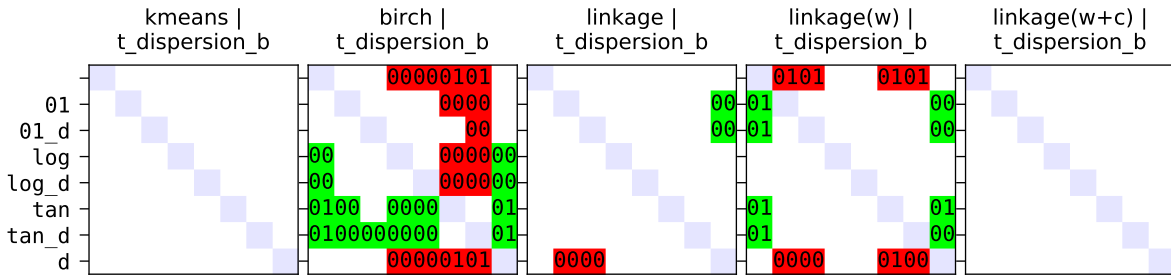
(h) Variant differences for dataset UCR and feature set $d_distribution$.



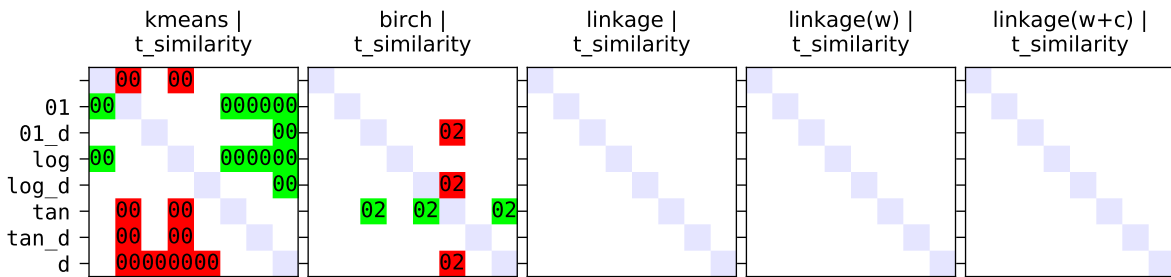
(i) Variant differences for dataset UCR and feature set $d_duplicates$.



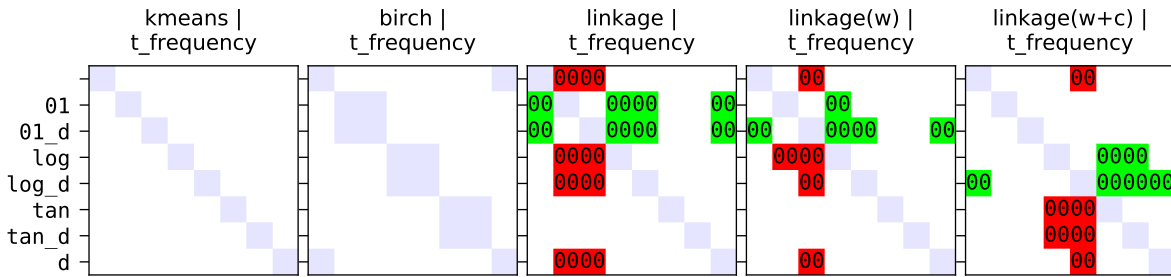
(j) Variant differences for dataset UCR and feature set $t_dispersion$.



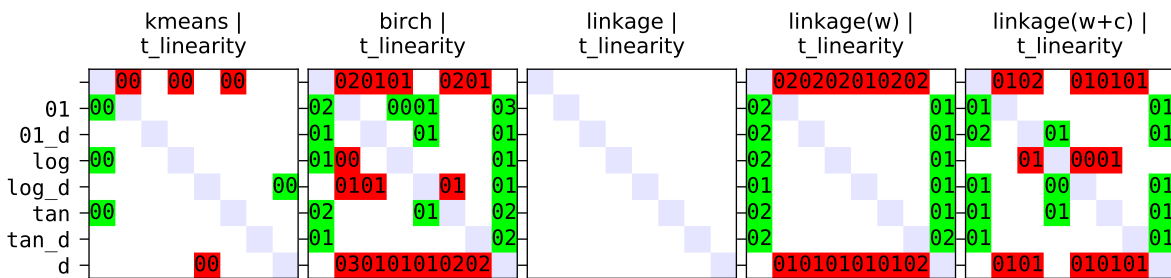
(k) Variant differences for dataset UCR and feature set $t_dispersion_b$.



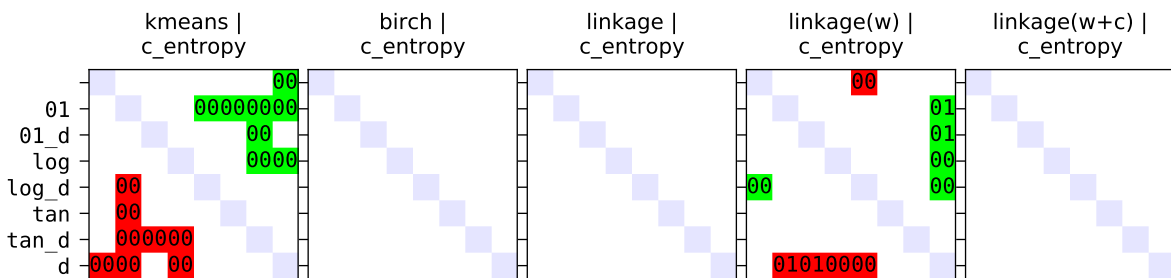
(l) Variant differences for dataset UCR and feature set $t_similarity$.



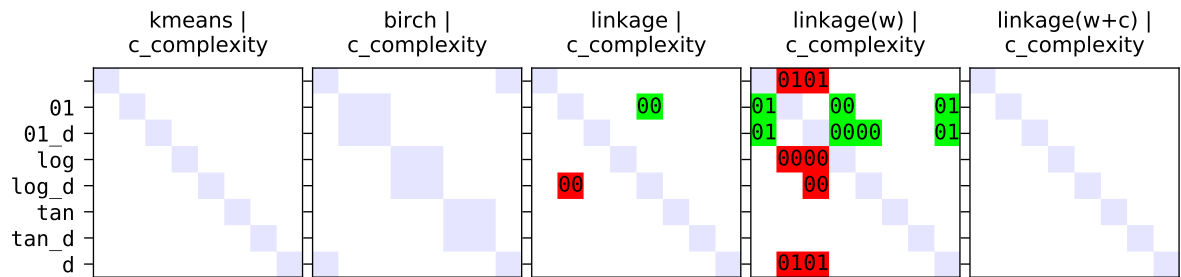
(m) Variant differences for dataset UCR and feature set $t_frequency$.



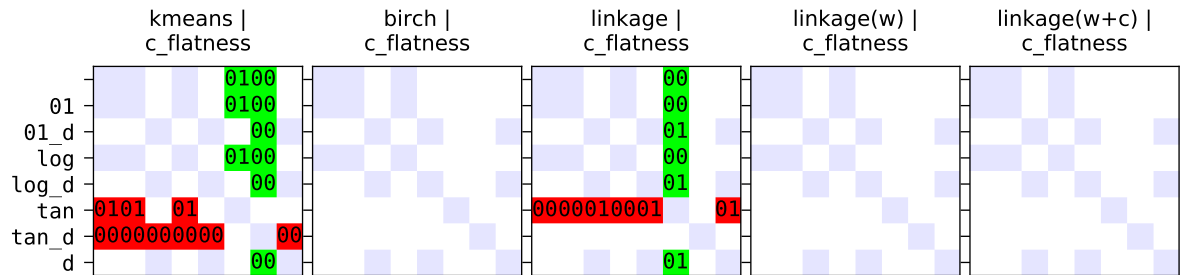
(n) Variant differences for dataset UCR and feature set $t_linearity$.



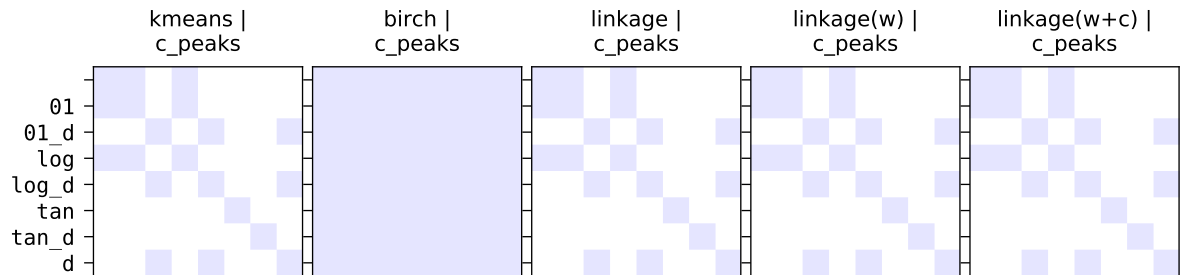
(o) Variant differences for dataset UCR and feature set $c_entropy$.



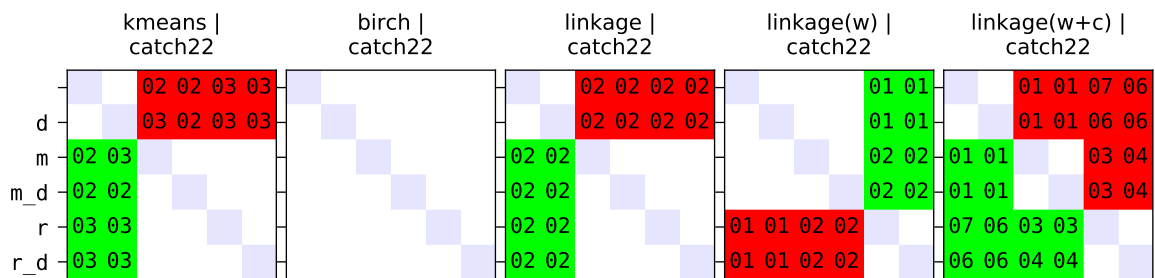
(p) Variant differences for dataset UCR and feature set *c_complexity*.



(q) Variant differences for dataset UCR and feature set *c_flatness*.

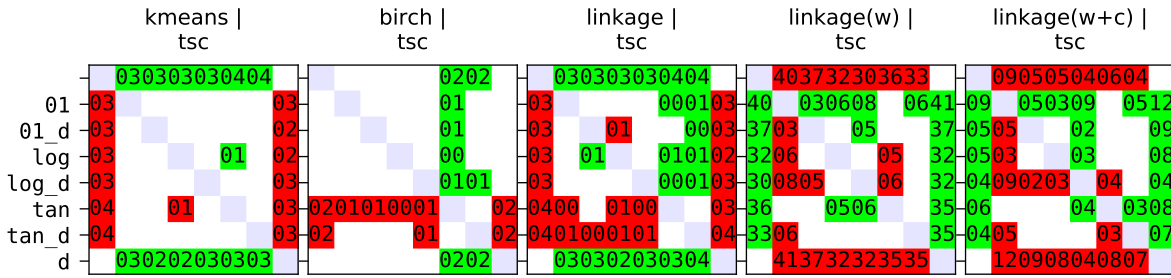


(r) Variant differences for dataset UCR and feature set *c_peaks*.

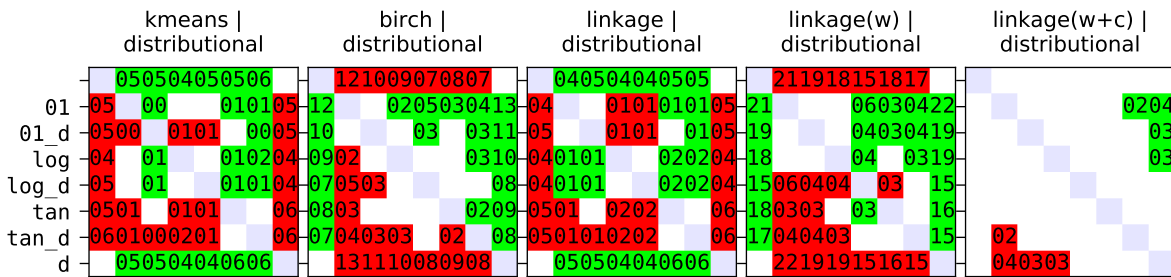


(s) Variant differences for dataset UCR and feature set *catch22*.

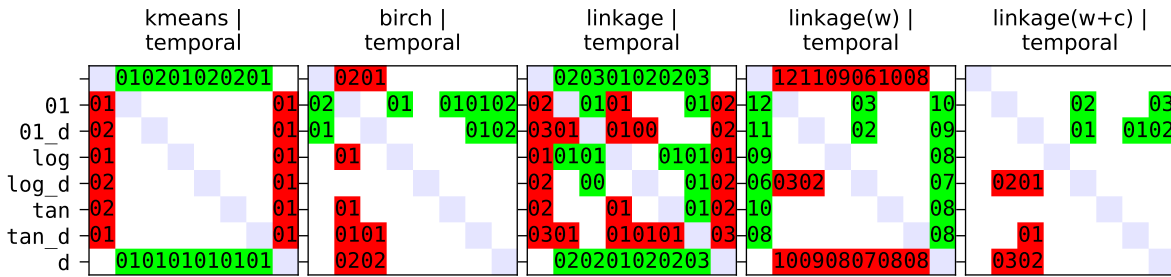
Figure D.3: Variant differences for the UCR dataset of the five models k-means (left), BIRCH (second), linkage (middle), linkage weighted (fourth) and linkage weighted cosine (right) in combination with all evaluated feature sets (TSC, four main groups, 13 subgroups, catch22). Abbreviations: empty = no post-processing, *01* = clip01, *tan* = clipTan, *log* = clipLog, *m* = minmax, *r* = robust, *d* = drop, *v_d* = variant with drop. Omitted column variants = row variants.



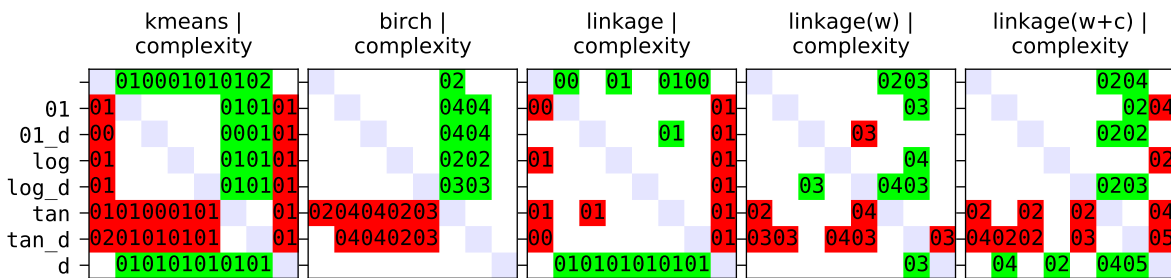
(a) Variant differences for dataset UCR-merged and feature set *tsc*.



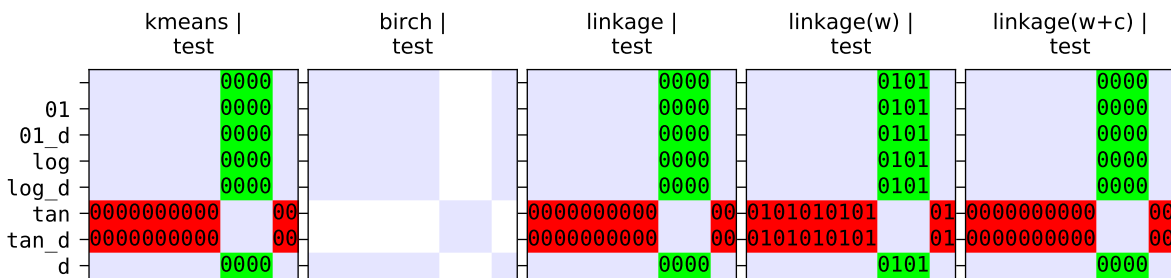
(b) Variant differences for dataset UCR-merged and feature set *distributional*.



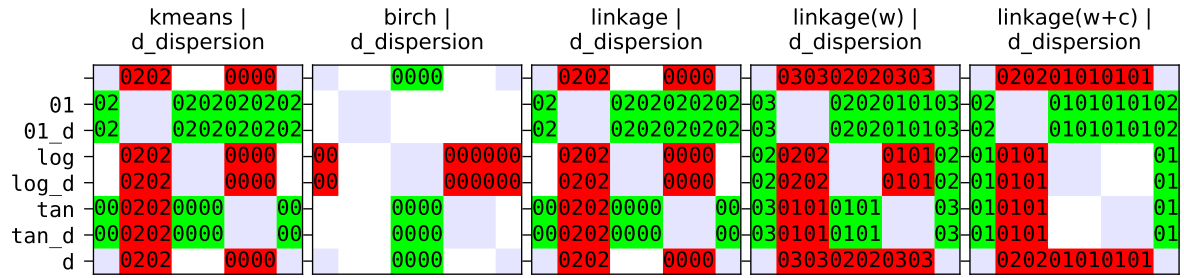
(c) Variant differences for dataset UCR-merged and feature set *temporal*.



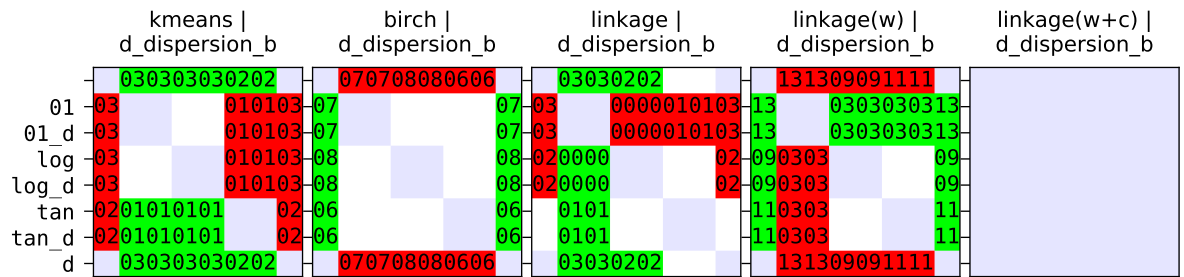
(d) Variant differences for dataset UCR-merged and feature set *complexity*.



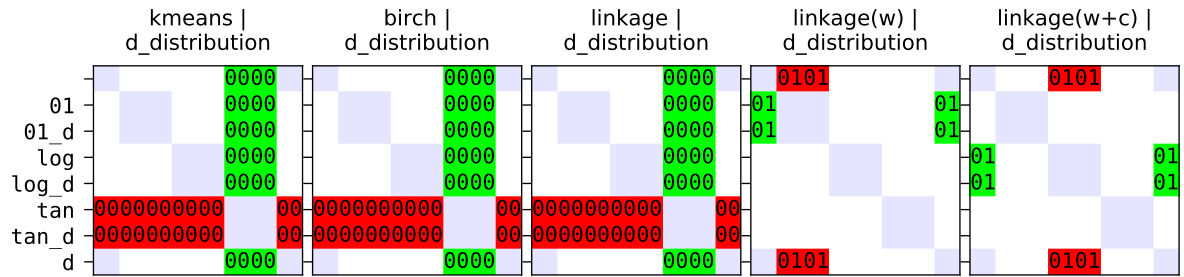
(e) Variant differences for dataset UCR-merged and feature set *test*.



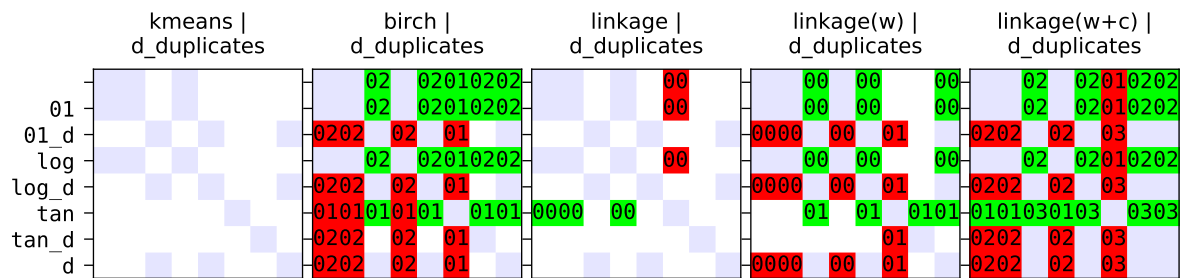
(f) Variant differences for dataset UCR-merged and feature set $d_dispersion$.



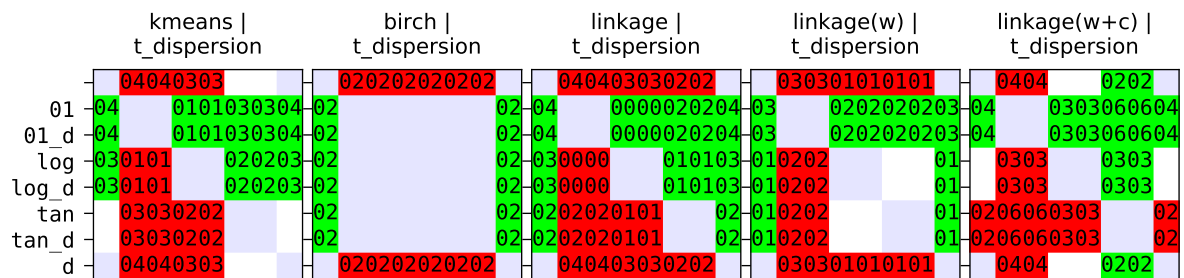
(g) Variant differences for dataset UCR-merged and feature set $d_dispersion_b$.



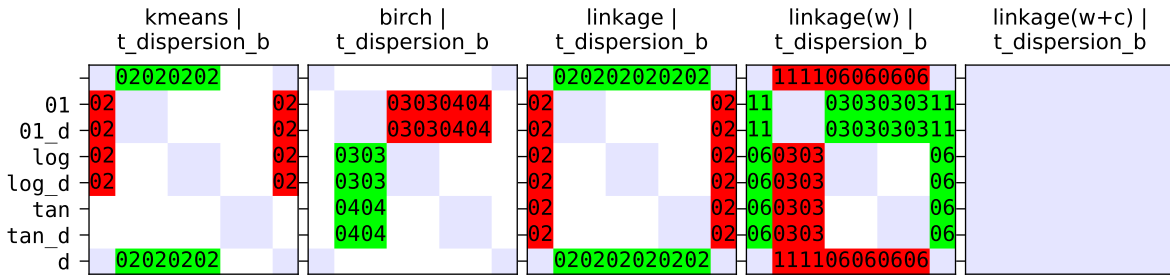
(h) Variant differences for dataset UCR-merged and feature set $d_distribution$.



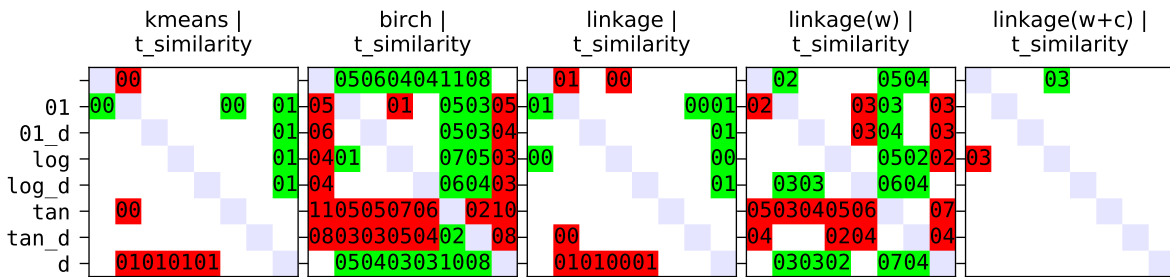
(i) Variant differences for dataset UCR-merged and feature set $d_duplicates$.



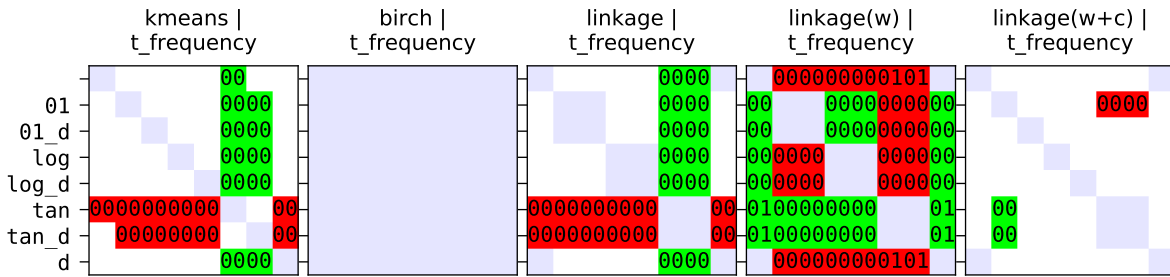
(j) Variant differences for dataset UCR-merged and feature set $t_dispersion$.



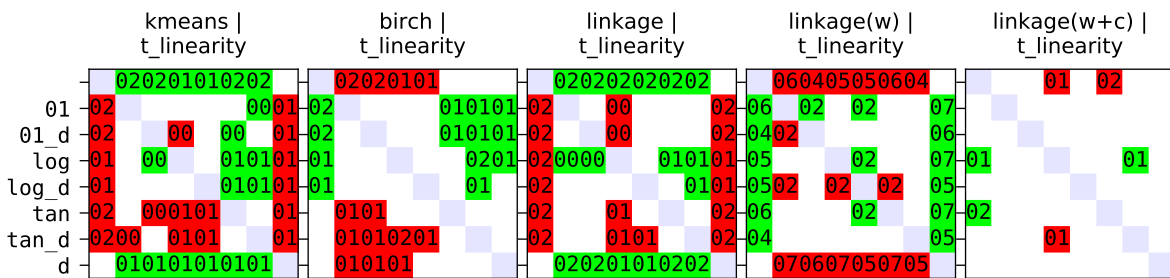
(k) Variant differences for dataset UCR-merged and feature set $t_dispersion_b$.



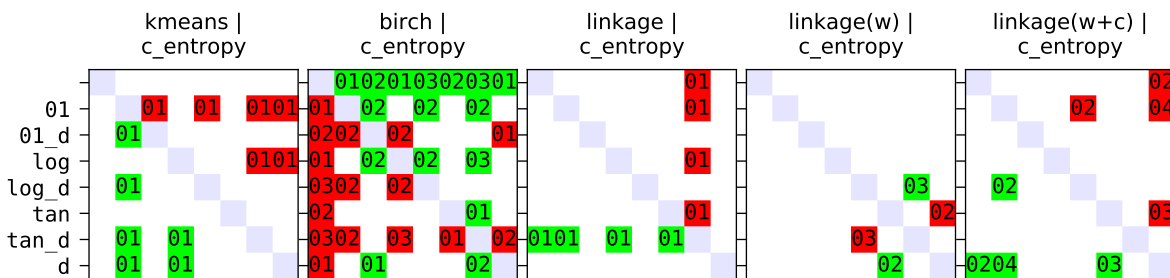
(l) Variant differences for dataset UCR-merged and feature set $t_similarity$.



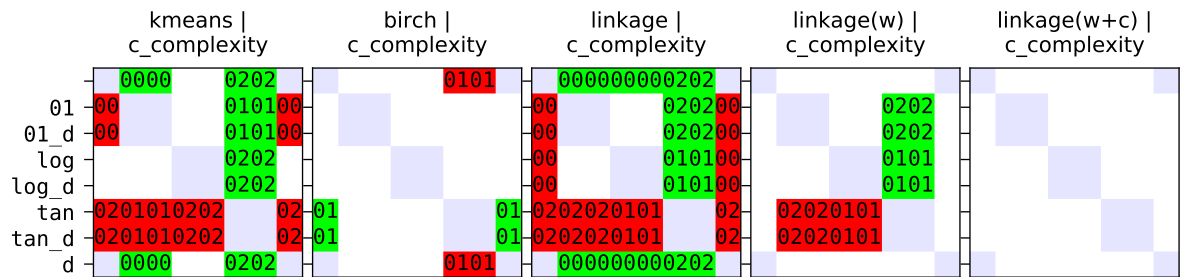
(m) Variant differences for dataset UCR-merged and feature set $t_frequency$.



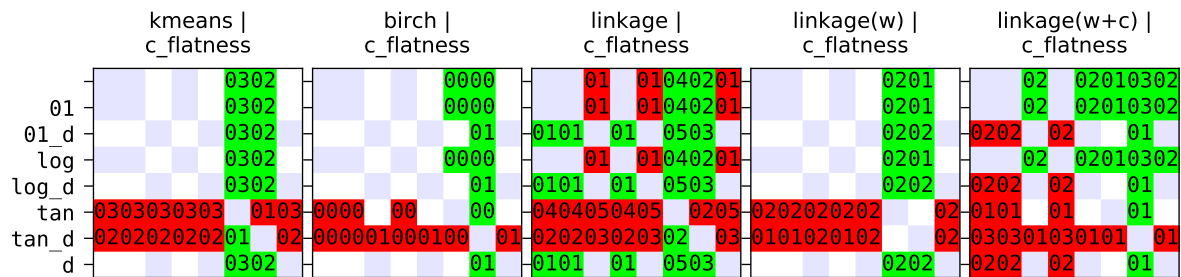
(n) Variant differences for dataset UCR-merged and feature set $t_linearity$.



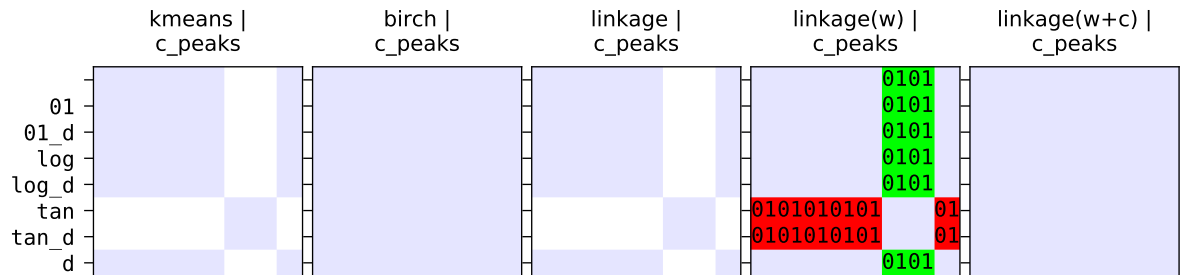
(o) Variant differences for dataset UCR-merged and feature set $c_entropy$.



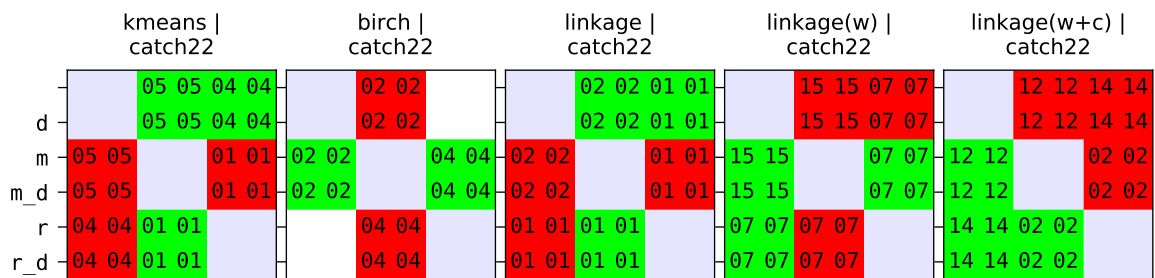
(p) Variant differences for dataset UCR-merged and feature set *c_complexity*.



(q) Variant differences for dataset UCR-merged and feature set *c_flatness*.

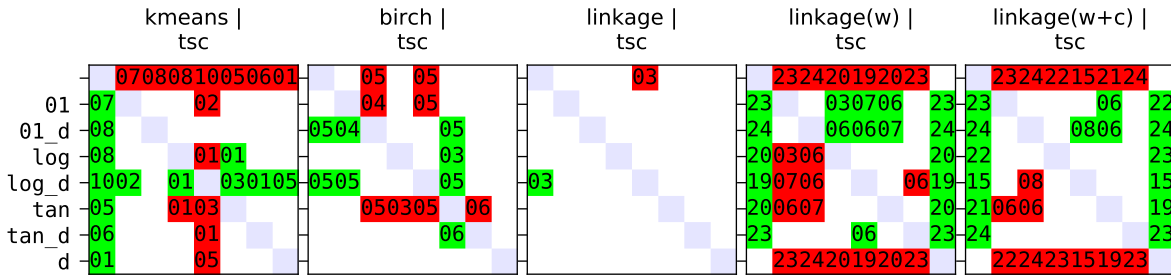


(r) Variant differences for dataset UCR-merged and feature set *c_peaks*.

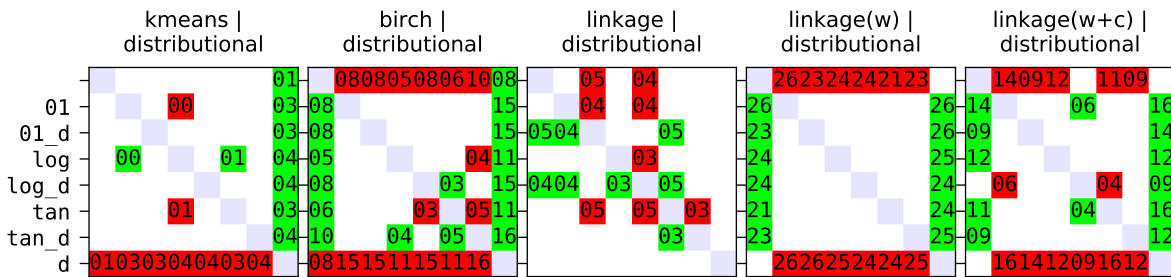


(s) Variant differences for dataset UCR-merged and feature set *catch22*.

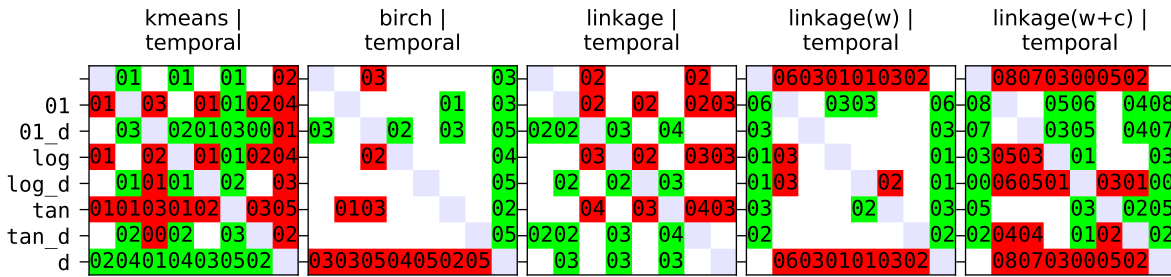
Figure D.4: Variant differences for the UCR-merged dataset of the five models k-means (left), BIRCH (second), linkage (middle), linkage weighted (fourth) and linkage weighted cosine (right) in combination with all evaluated feature sets (TSC, four main groups, 13 subgroups, catch22). Abbreviations: empty = no post-processing, *01* = clip01, *tan* = clipTan, *log* = clipLog, *m* = minmax, *r* = robust, *d* = drop, *v_d* = variant with drop. Omitted column variants = row variants.



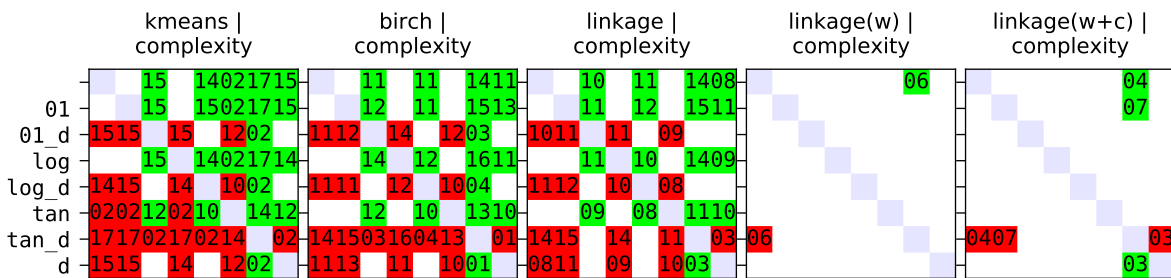
(a) Variant differences for dataset $IMTS_1$ and feature set *tsc*.



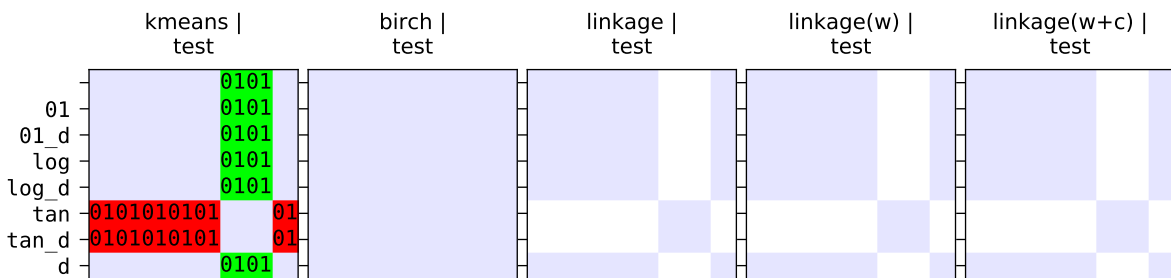
(b) Variant differences for dataset $IMTS_1$ and feature set *distributional*.



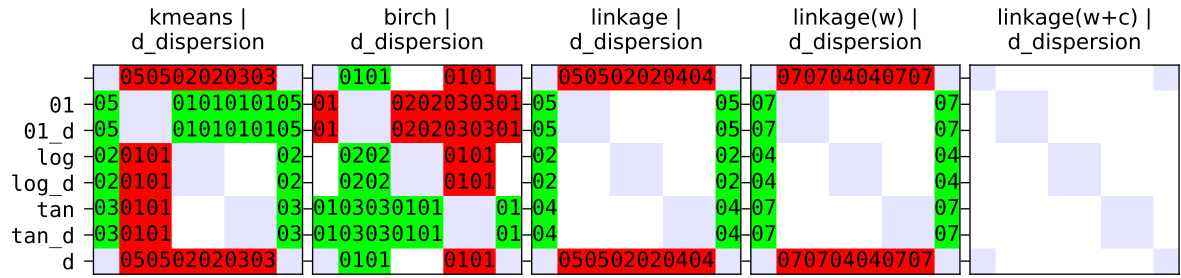
(c) Variant differences for dataset $IMTS_1$ and feature set *temporal*.



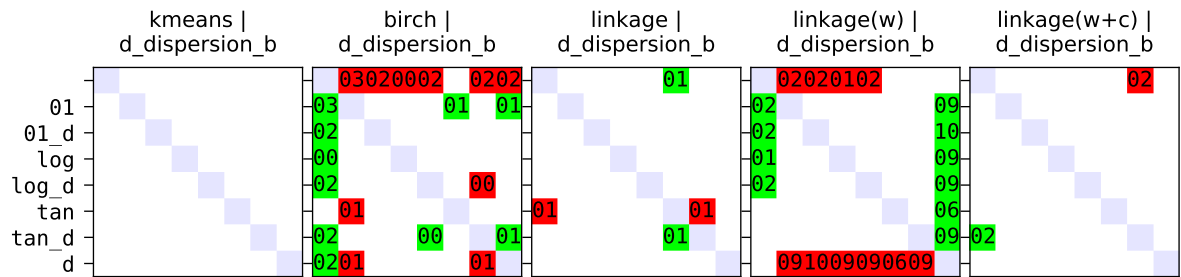
(d) Variant differences for dataset $IMTS_1$ and feature set *complexity*.



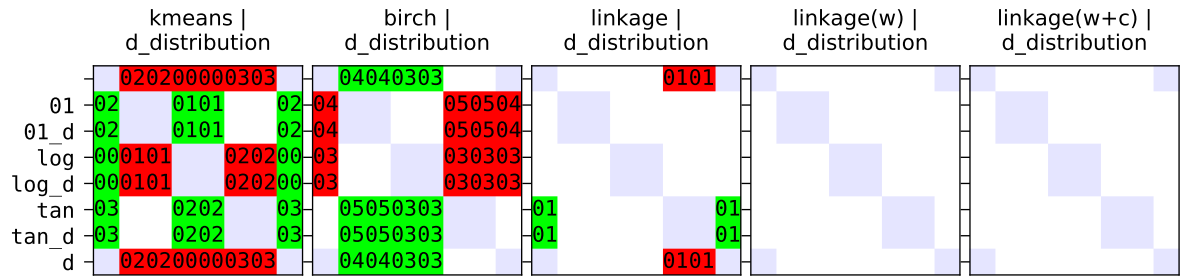
(e) Variant differences for dataset $IMTS_1$ and feature set *test*.



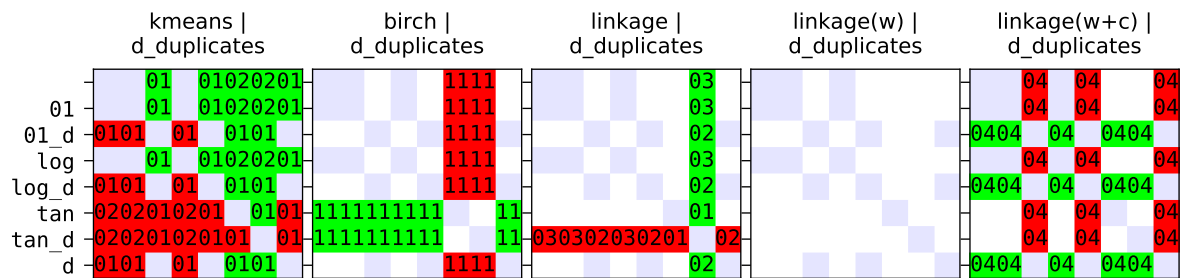
(f) Variant differences for dataset $IMTS_1$ and feature set $d_dispersion$.



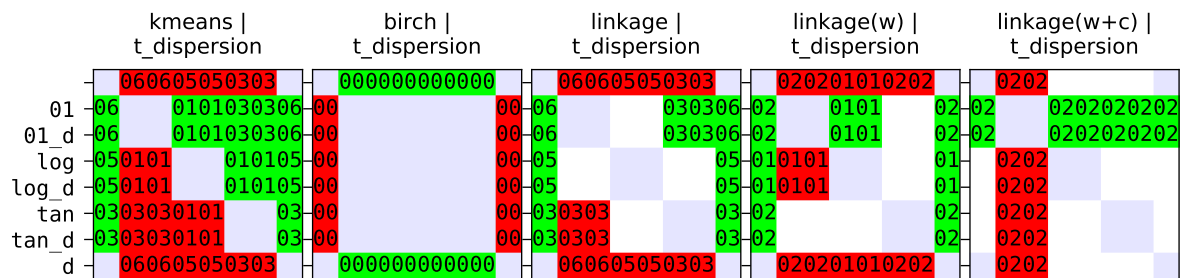
(g) Variant differences for dataset $IMTS_1$ and feature set $d_dispersion_b$.



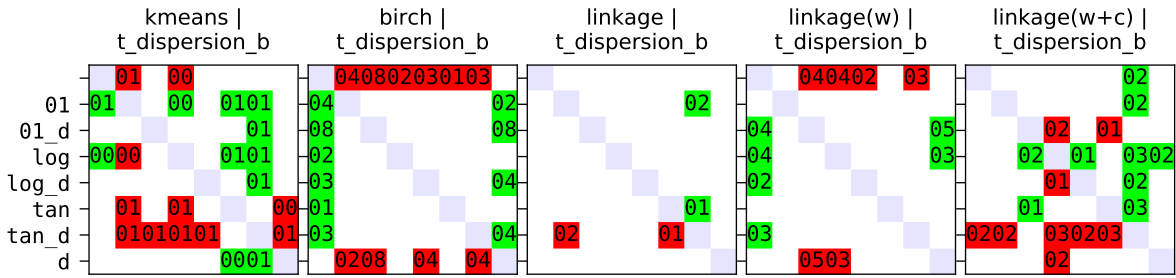
(h) Variant differences for dataset $IMTS_1$ and feature set $d_distribution$.



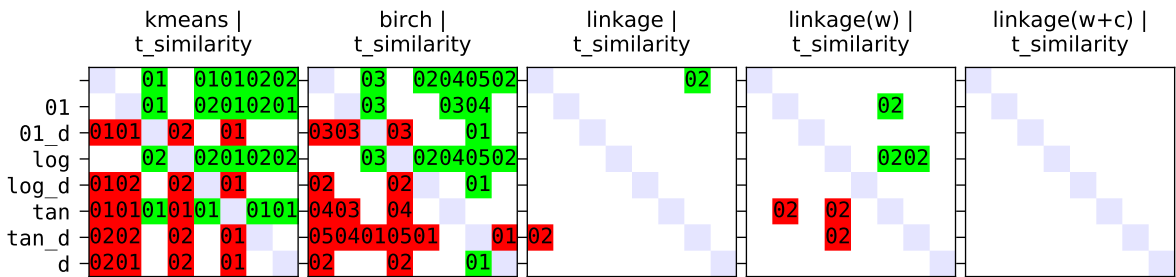
(i) Variant differences for dataset $IMTS_1$ and feature set $d_duplicates$.



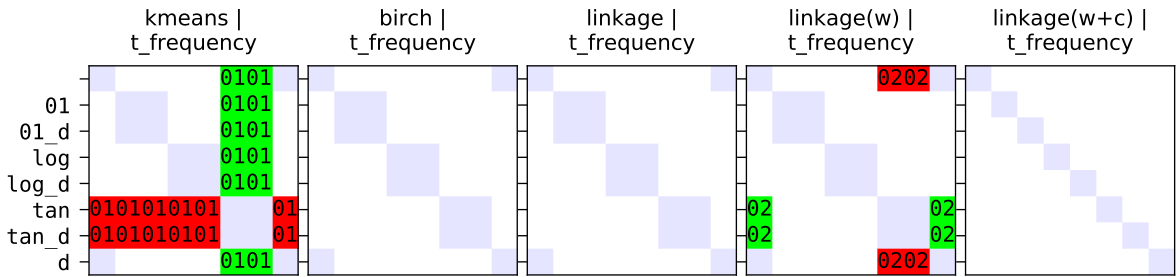
(j) Variant differences for dataset $IMTS_1$ and feature set $t_dispersion$.



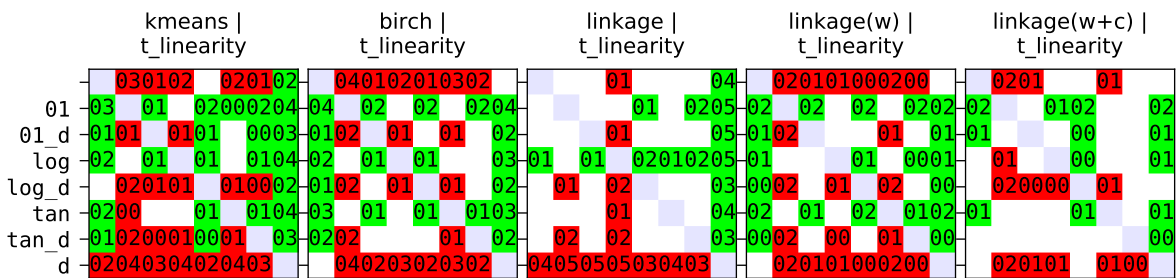
(k) Variant differences for dataset $IMTS_1$ and feature set $t_dispersion_b$.



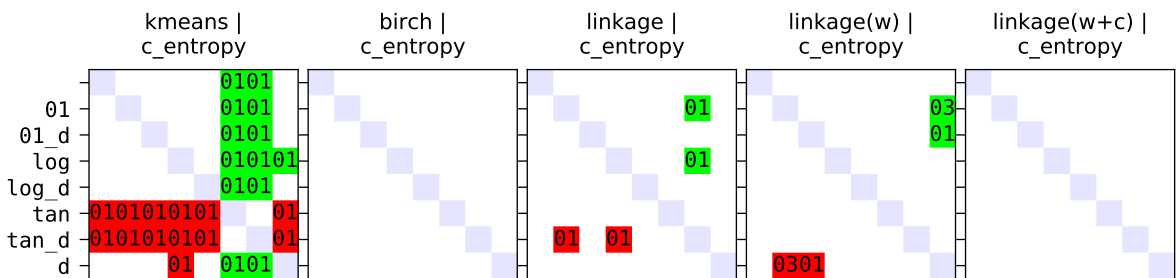
(l) Variant differences for dataset $IMTS_1$ and feature set $t_similarity$.



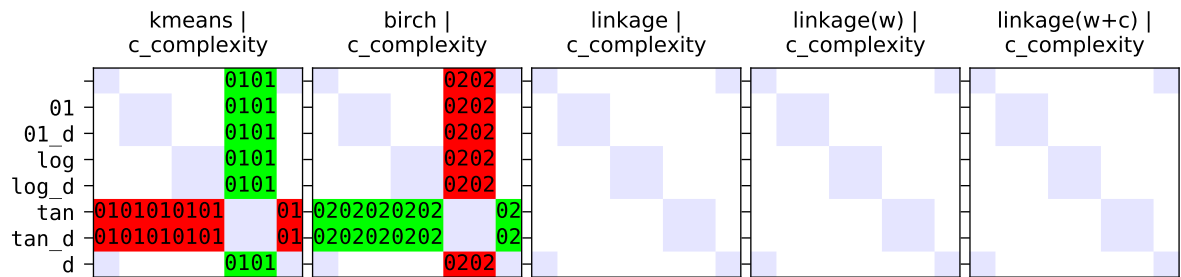
(m) Variant differences for dataset $IMTS_1$ and feature set $t_frequency$.



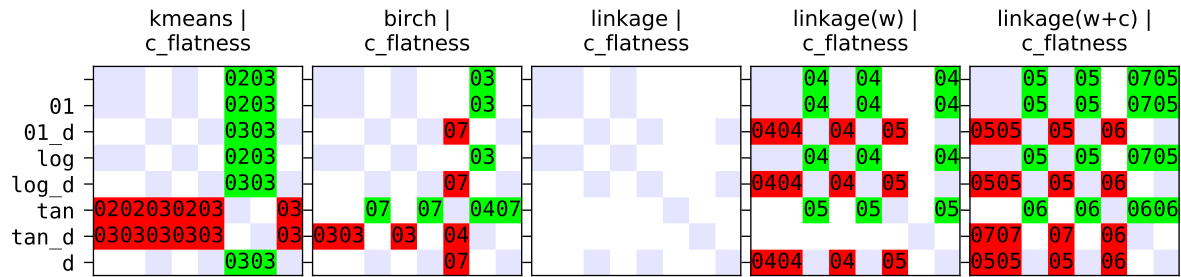
(n) Variant differences for dataset $IMTS_1$ and feature set $t_linearity$.



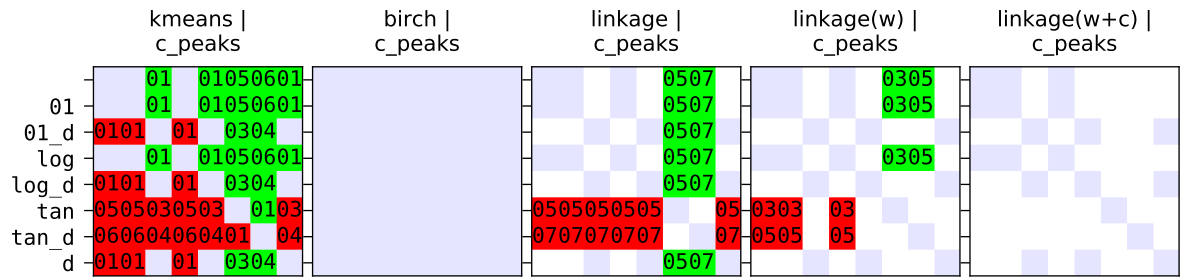
(o) Variant differences for dataset $IMTS_1$ and feature set $c_entropy$.



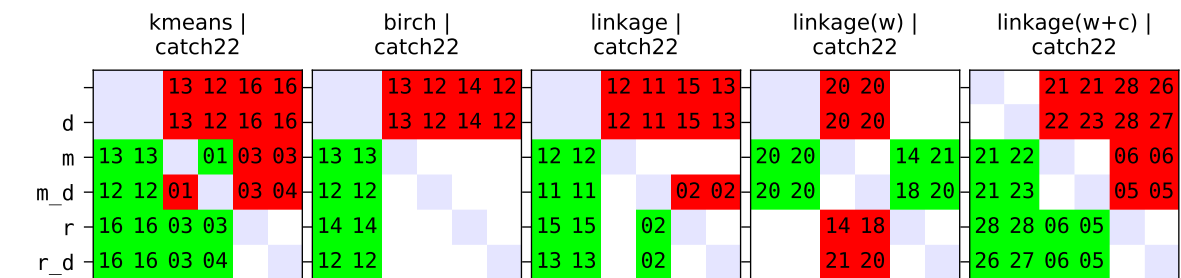
(p) Variant differences for dataset $IMTS_1$ and feature set $c_complexity$.



(q) Variant differences for dataset $IMTS_1$ and feature set $c_flatness$.

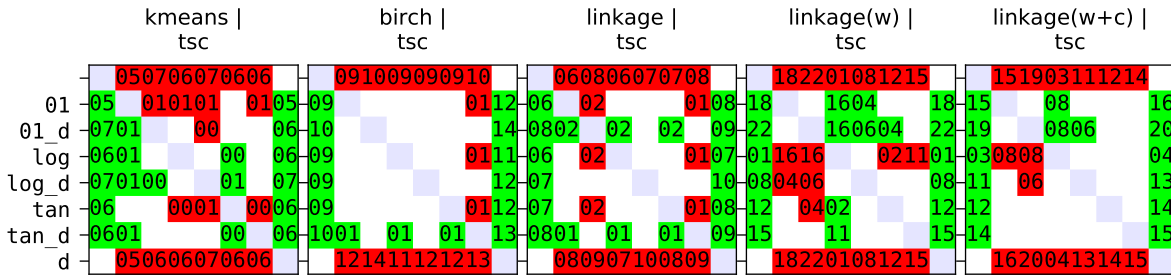


(r) Variant differences for dataset $IMTS_1$ and feature set c_peaks .

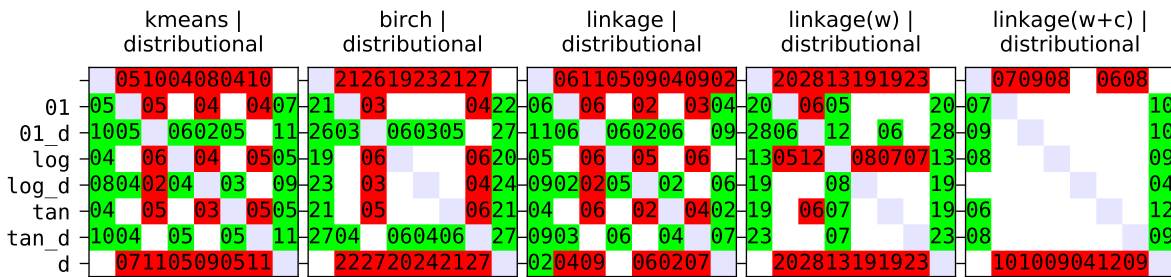


(s) Variant differences for dataset $IMTS_1$ and feature set $catch22$.

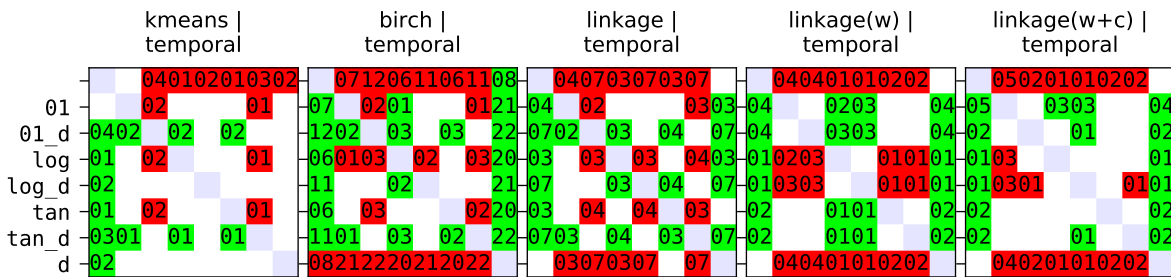
Figure D.5: Variant differences for the $IMTS_1$ dataset of the five models k-means (left), BIRCH (second), linkage (middle), linkage weighted (fourth) and linkage weighted cosine (right) in combination with all evaluated feature sets (TSC, four main groups, 13 subgroups, catch22). Abbreviations: empty = no post-processing, 01 = clip01, tan = clipTan, log = clipLog, m = minmax, r = robust, d = drop, v_d = variant with drop. Omitted column variants = row variants.



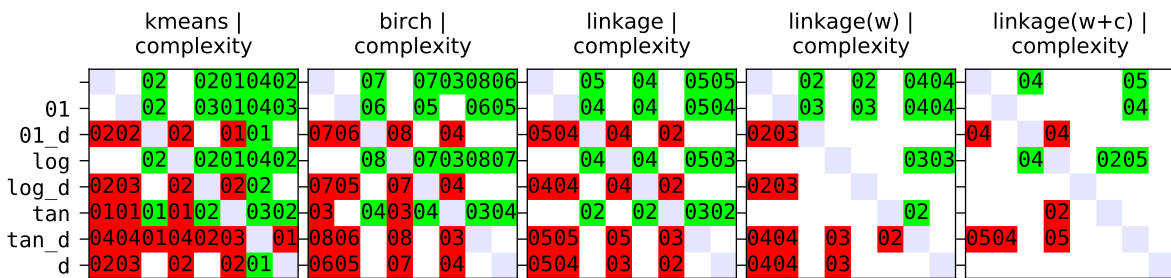
(a) Variant differences for dataset IMTS₂ and feature set *tsc*.



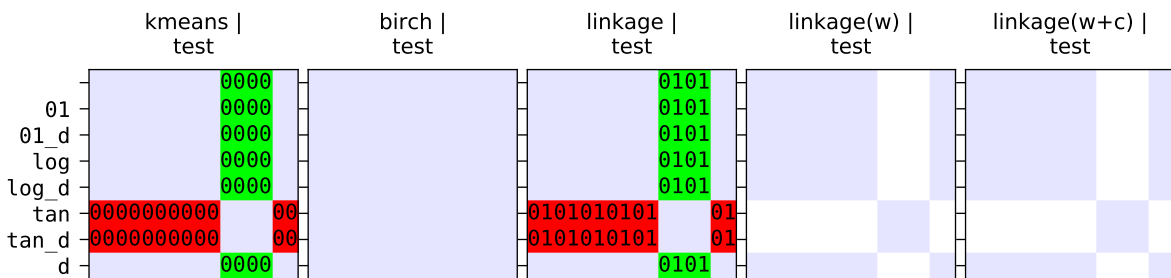
(b) Variant differences for dataset IMTS₂ and feature set *distributional*.



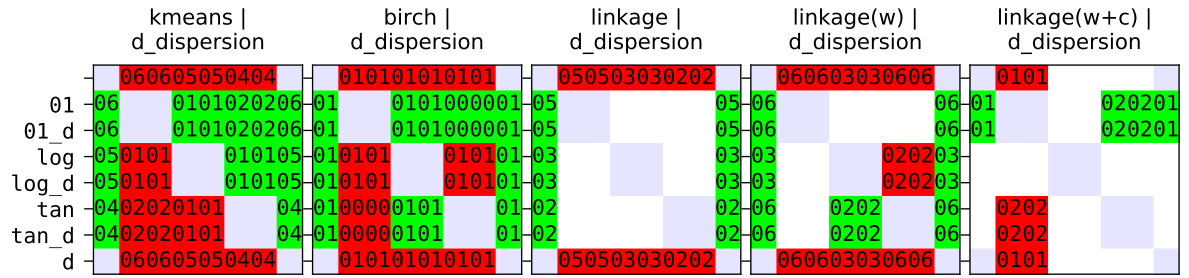
(c) Variant differences for dataset IMTS₂ and feature set *temporal*.



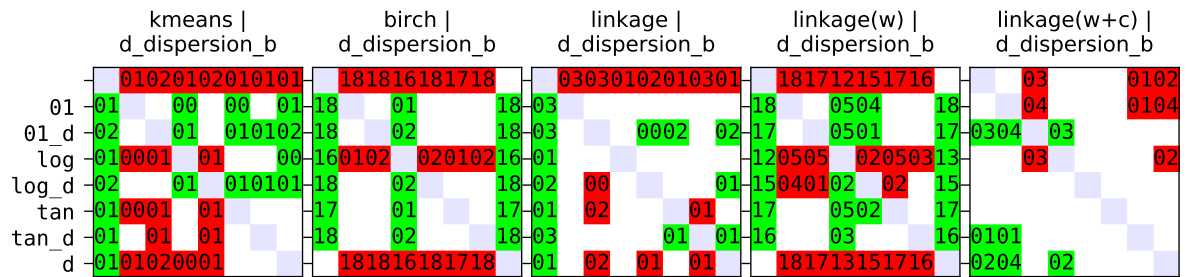
(d) Variant differences for dataset IMTS₂ and feature set *complexity*.



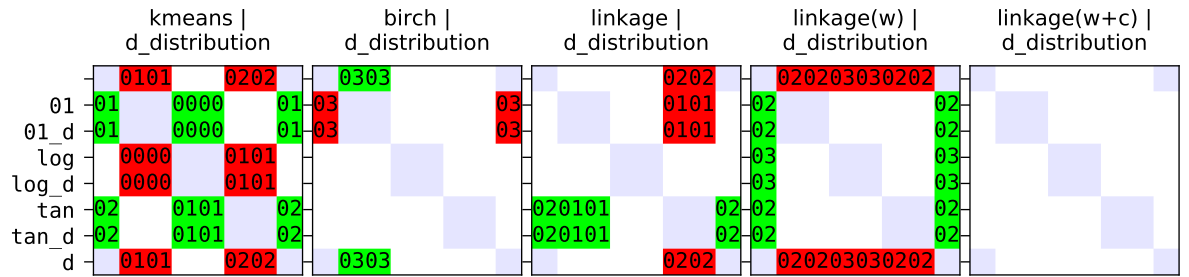
(e) Variant differences for dataset IMTS₂ and feature set *test*.



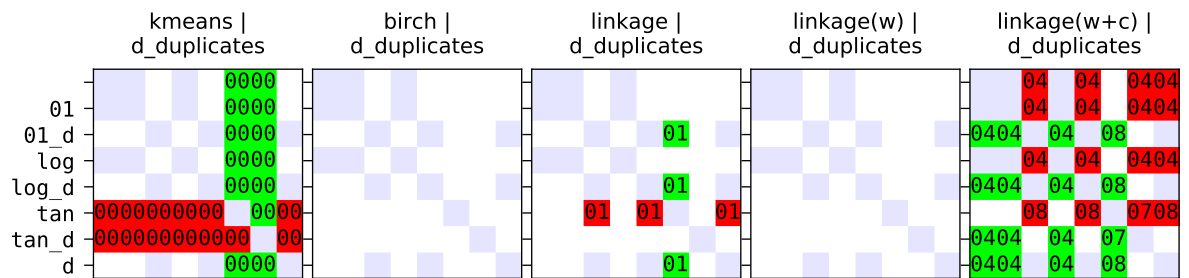
(f) Variant differences for dataset IMTS₂ and feature set *d_dispersion*.



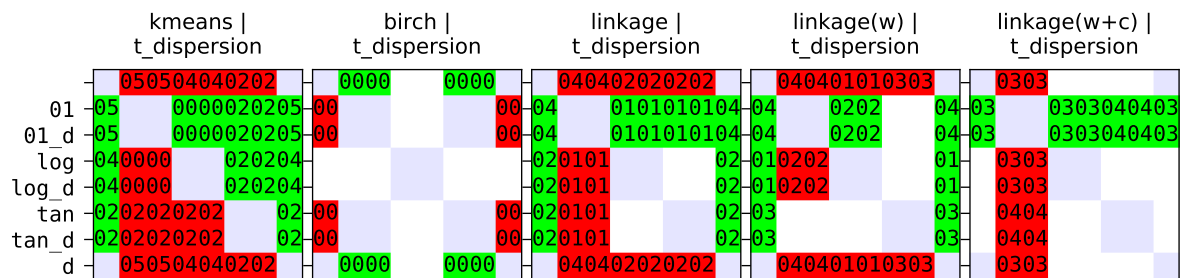
(g) Variant differences for dataset IMTS₂ and feature set *d_dispersion_b*.



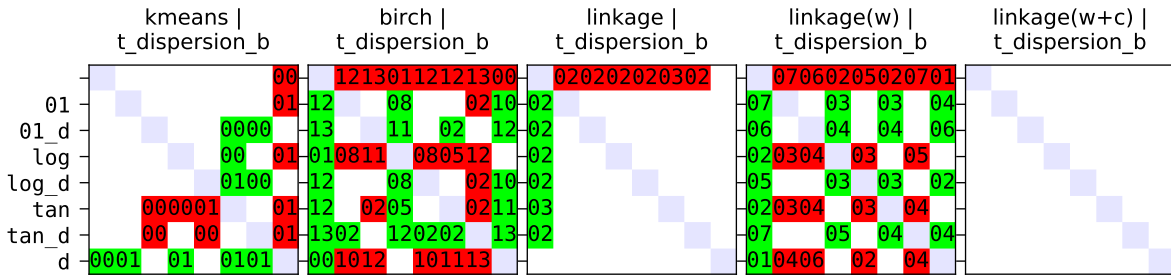
(h) Variant differences for dataset IMTS₂ and feature set *d_distribution*.



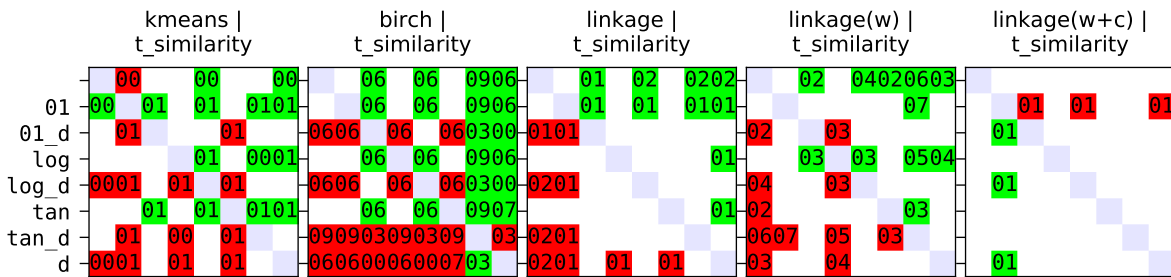
(i) Variant differences for dataset IMTS₂ and feature set *d_duplicates*.



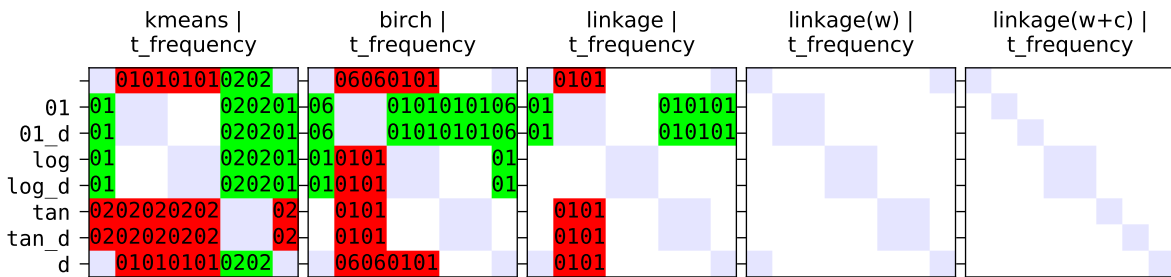
(j) Variant differences for dataset IMTS₂ and feature set *t_dispersion*.



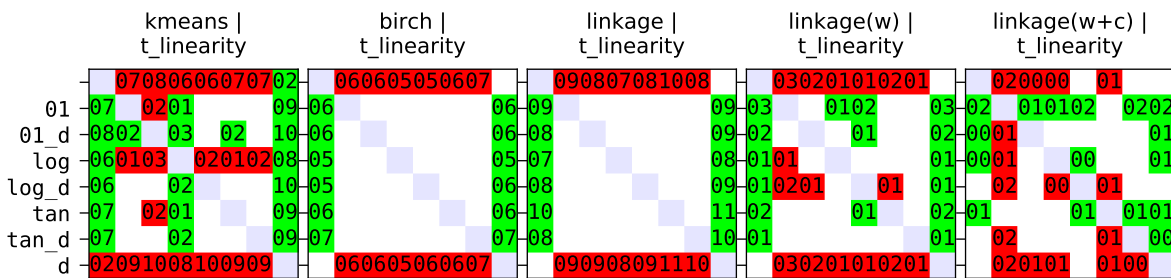
(k) Variant differences for dataset IMTS₂ and feature set *t_dispersion_b*.



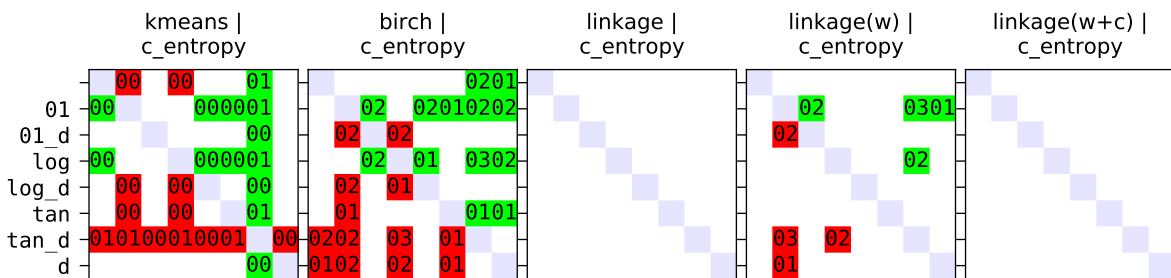
(l) Variant differences for dataset IMTS₂ and feature set *t_similarity*.



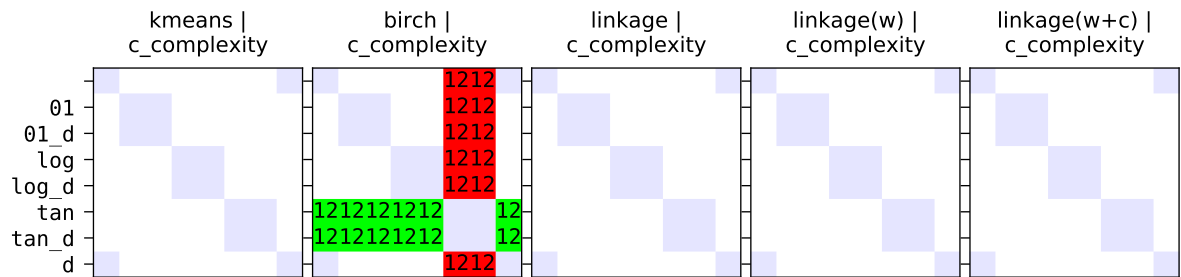
(m) Variant differences for dataset IMTS₂ and feature set *t_frequency*.



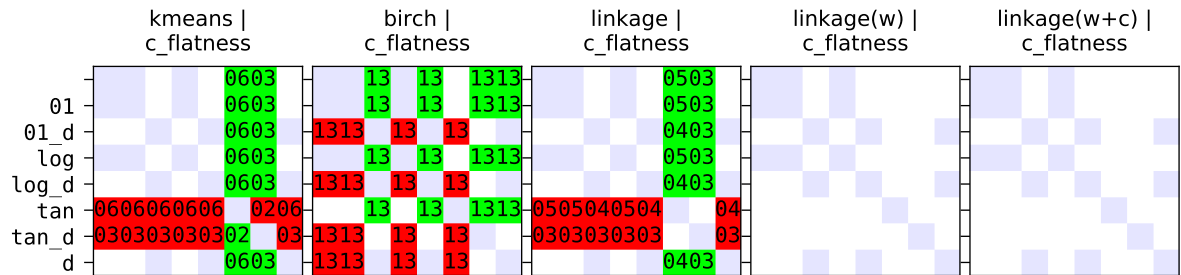
(n) Variant differences for dataset IMTS₂ and feature set *t_linearity*.



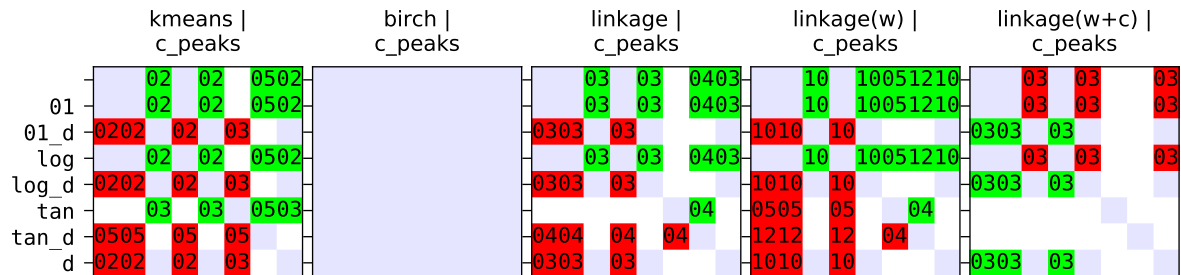
(o) Variant differences for dataset IMTS₂ and feature set *c_entropy*.



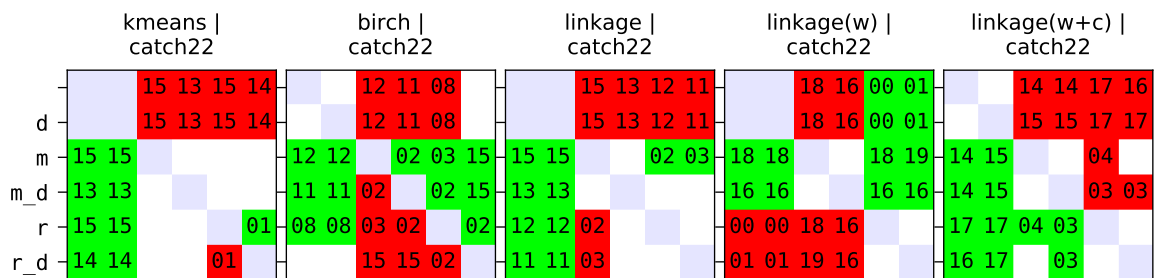
(p) Variant differences for dataset IMTS₂ and feature set *c_complexity*.



(q) Variant differences for dataset IMTS₂ and feature set *c_flatness*.



(r) Variant differences for dataset IMTS₂ and feature set *c_peaks*.



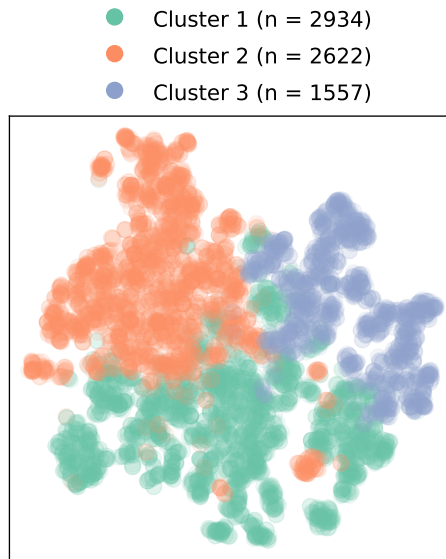
(s) Variant differences for dataset IMTS₂ and feature set *catch22*.

Figure D.6: Variant differences for the IMTS₂ dataset of the five models k-means (left), BIRCH (second), linkage (middle), linkage weighted (fourth) and linkage weighted cosine (right) in combination with all evaluated feature sets (TSC, four main groups, 13 subgroups, catch22). Abbreviations: empty = no post-processing, 01 = clip01, tan = clipTan, log = clipLog, m = minmax, r = robust, d = drop, v_d = variant with drop. Omitted column variants = row variants.

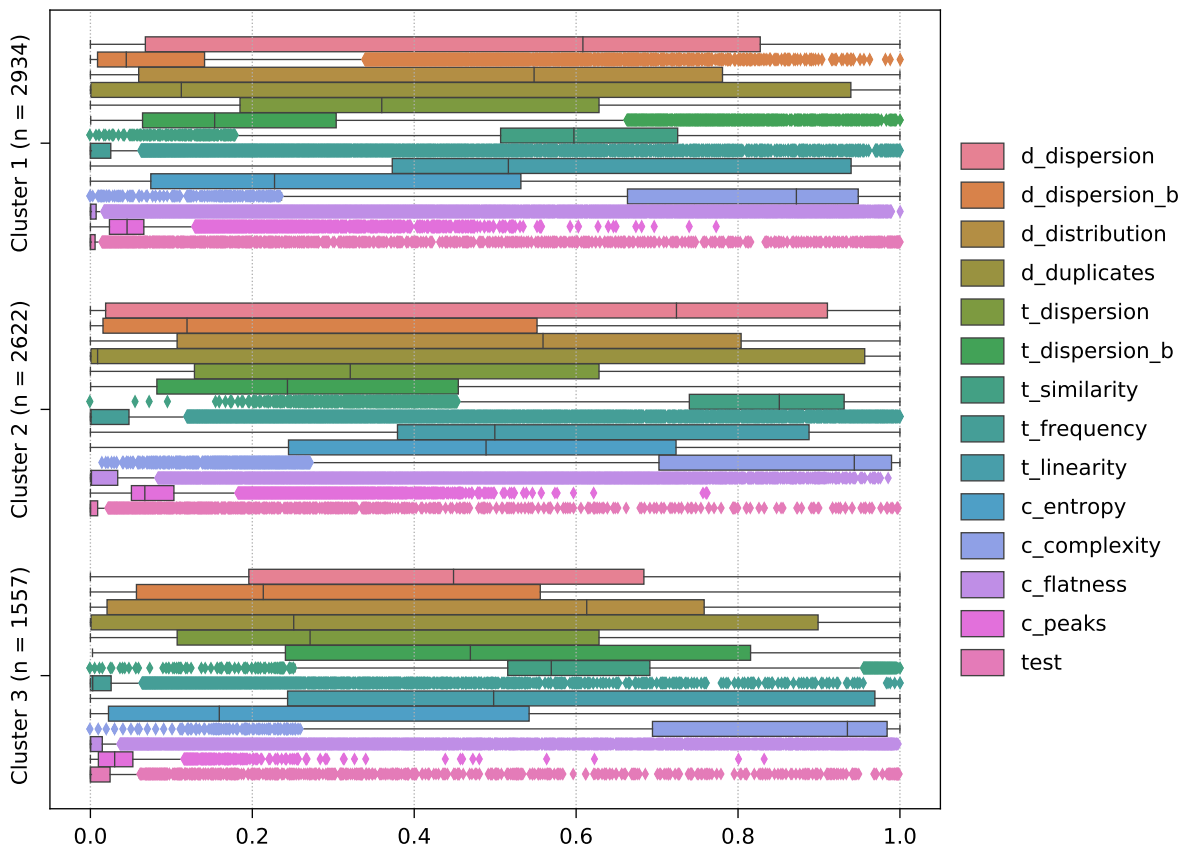
Regarding the IMTS₂ dataset, we first complete the evaluation results by presenting the initial dendrogram (with representative time series) of the Disk Available % (D-03) metric when clustered with the *distributional* feature set (cf. [Figure D.13](#)). [Figure D.14](#) displays the t-SNE visualization and cluster feature values when partitioning the same data into six new clusters instead of the previous three. In [Figure D.15](#), the t-SNE visualization, Venn diagram and cluster feature values are displayed after clustering with the entire TSC group. Lastly, we show the entire clustering output for the remaining unlabeled metrics of the IMTS₂ dataset, which are all again based on the *distributional* feature set (with our selected method). In [Figure D.16](#), we present the clustering results for the CPU Idle (H-01) metric, in [Figure D.17](#) for the CPU IO Wait (H-05) metric, in [Figure D.18](#) for the Page Faults (H-06) metric, in [Figure D.19](#) for the Memory Available % (H-07) metric and in [Figure D.20](#) for the Read Bytes (D-04) metric.

[Table D.2](#) lists all unlabeled dataset evaluation results together with various internal evaluation metrics (cf. [Section 2.4.4.1 on p. 23](#)).

Overall, we can see good clustering results throughout, especially when we cross-analyze the t-SNE visualizations, where the samples are tinted based on the predicted clusters, which allows us to observe that equally colored samples (i.e., samples from a common cluster) are in close proximity in the majority of the cases.

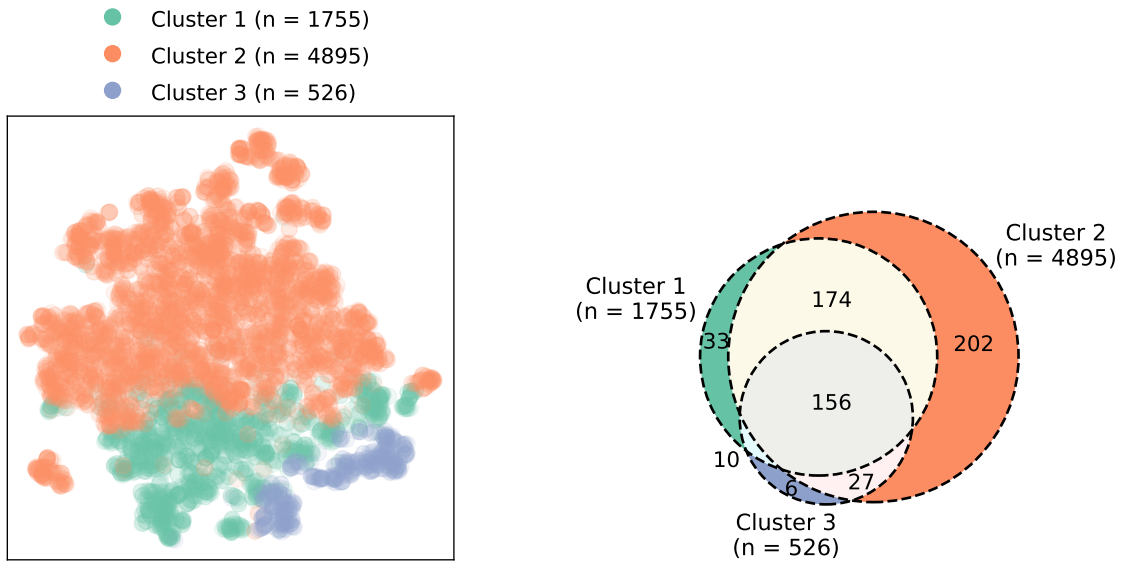


(a) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.



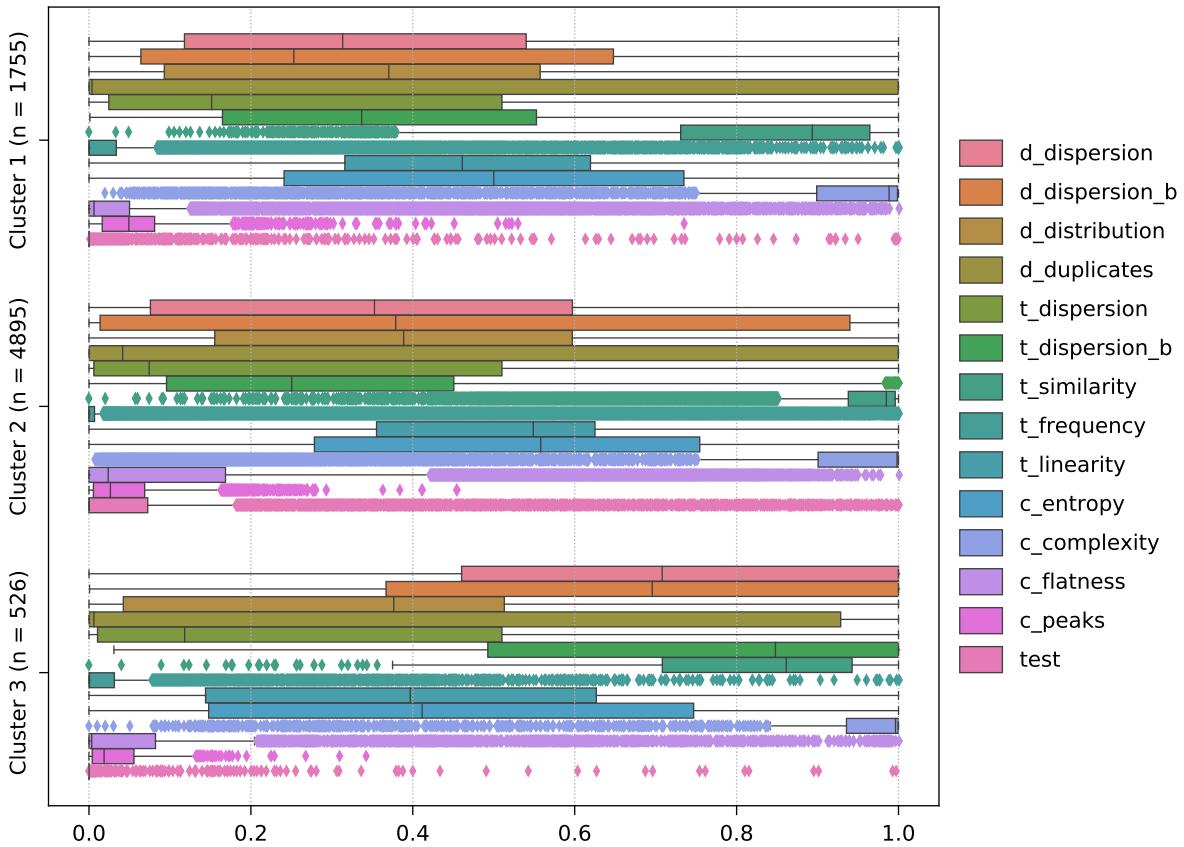
(b) Cluster feature values of all TSC (sub)groups, clipped to $[0, 1]$. Abbreviations: d = distributional, t = temporal, c = complexity, b = blockwise.

Figure D.7: Various results obtained when clustering the 7113 CPU Idle (H-01) series of the IMTS₁ dataset into three clusters. The respective cluster sizes are denoted by n .



(a) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

(b) Venn diagram showing the distribution of the 608 systems.



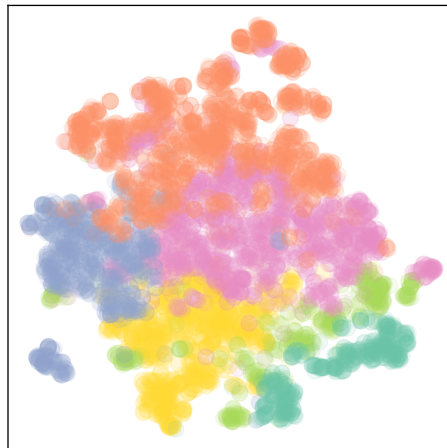
(c) Cluster feature values of all TSC (sub)groups, clipped to [0, 1]. Abbreviations: *d* = distributional, *t* = temporal, *c* = complexity, *b* = blockwise.

Figure D.8: Various results obtained when clustering the 7176 Memory Available % (H-07) series of the IMTS₁ dataset into three clusters. The respective cluster sizes are denoted by *n*.

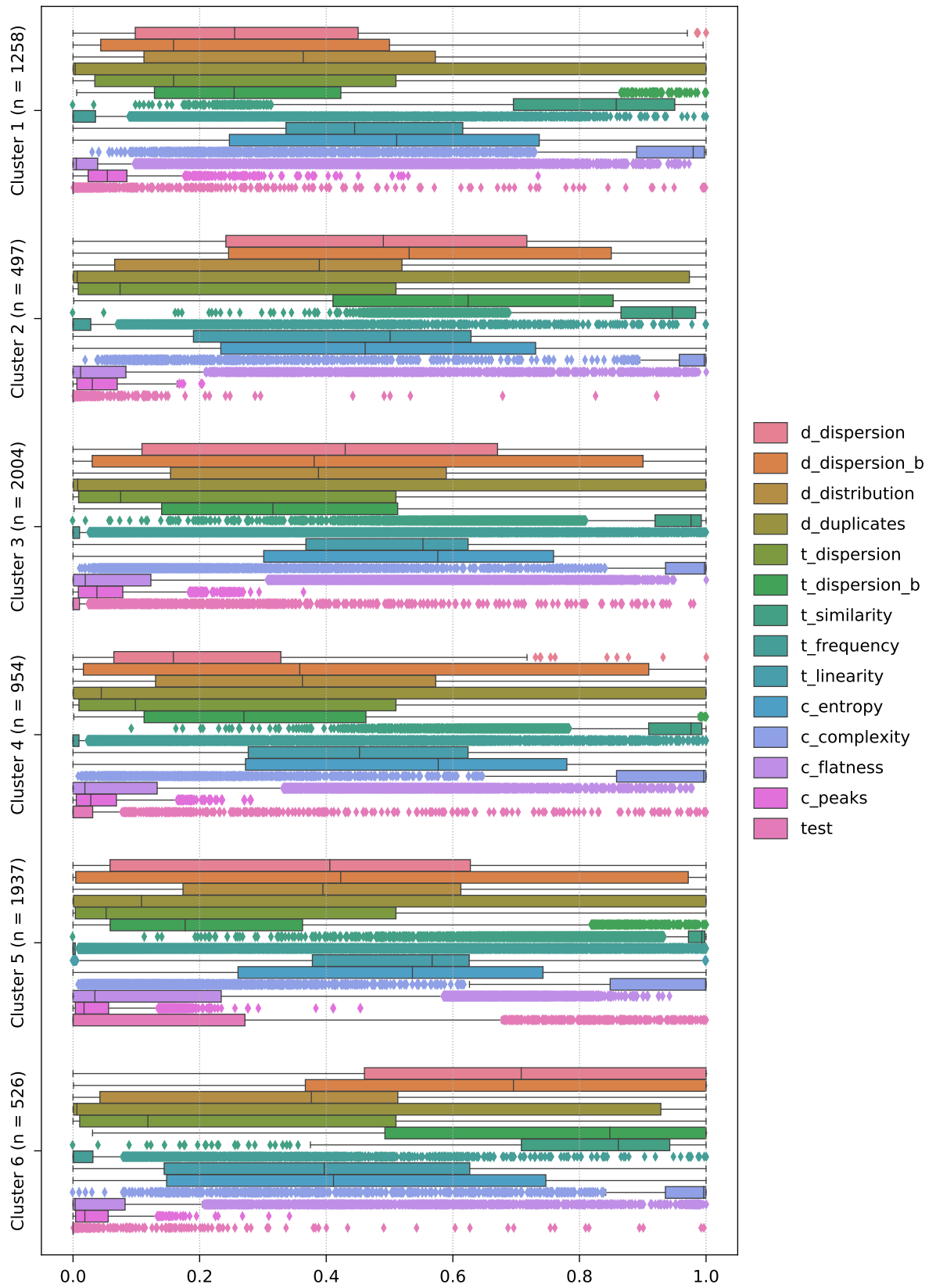
Dataset	ID: Metric	Feature Set	#C	Internal Evaluation Metrics			
				Silhouette	Davies-Bouldin	Dunn	Caliński-Harabasz
IMTS ₁	H-01: CPU Idle	<i>distributional</i>	3	0.18	1.67	0.04	2302.91
	H-06: Page Faults	<i>distributional</i>	2	0.29	1.26	0.03	1526.83
	H-07: Memory Available %	<i>distributional</i>	3	0.27	1.50	0.05	2516.14
	H-07: Memory Available %	<i>distributional</i>	6	0.12	1.98	0.04	1553.69
	D-03: Disk Available %	<i>distributional</i>	2	0.52	1.12	0.08	3611.57
	N-01: Bytes Received	<i>distributional</i>	3	0.29	1.19	0.02	3526.10
	H-01: CPU Idle	<i>distributional</i>	3	0.23	1.52	0.06	407.03
IMTS ₂	H-05: CPU IO Wait	<i>distributional</i>	2	0.35	1.16	0.09	413.41
	H-06: Page Faults	<i>distributional</i>	3	0.47	0.78	0.12	608.88
	H-07: Memory Available %	<i>distributional</i>	2	0.32	1.34	0.10	575.04
	D-03: Disk Available %	<i>distributional</i>	2	0.45	1.24	0.10	774.99
	D-03: Disk Available %	<i>distributional</i>	6	0.16	1.70	0.04	443.16
	D-03: Disk Available %	all TSC	3	0.15	1.95	0.15	398.66
	D-04: Read Bytes	<i>distributional</i>	3	0.38	0.98	0.03	1605.48

Table D.2: Internal evaluation metrics for all unlabeled dataset evaluations. #C represents the number of identified clusters.

- Cluster 1 (n = 1258)
- Cluster 2 (n = 497)
- Cluster 3 (n = 2004)
- Cluster 4 (n = 954)
- Cluster 5 (n = 1937)
- Cluster 6 (n = 526)

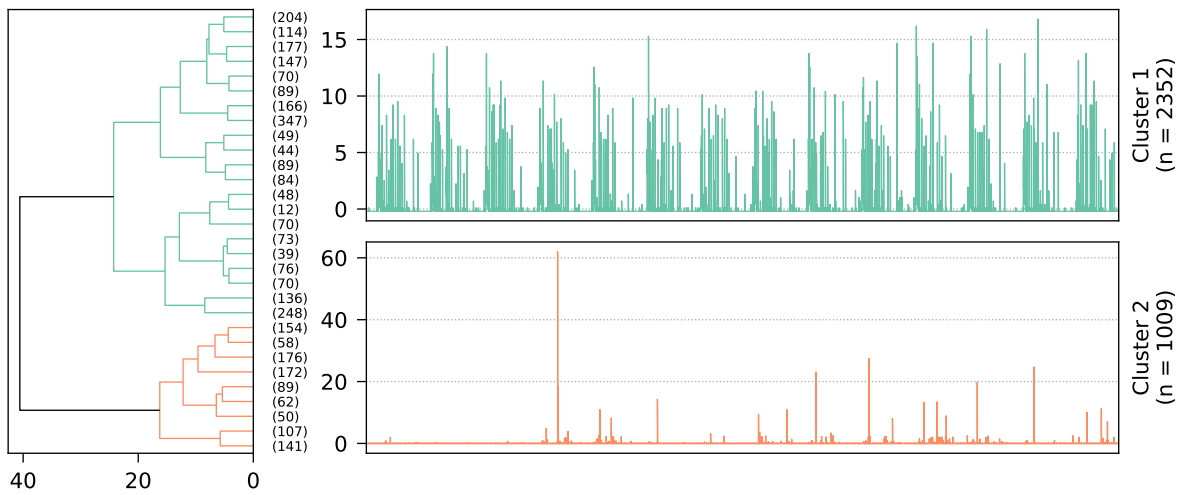


(a) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

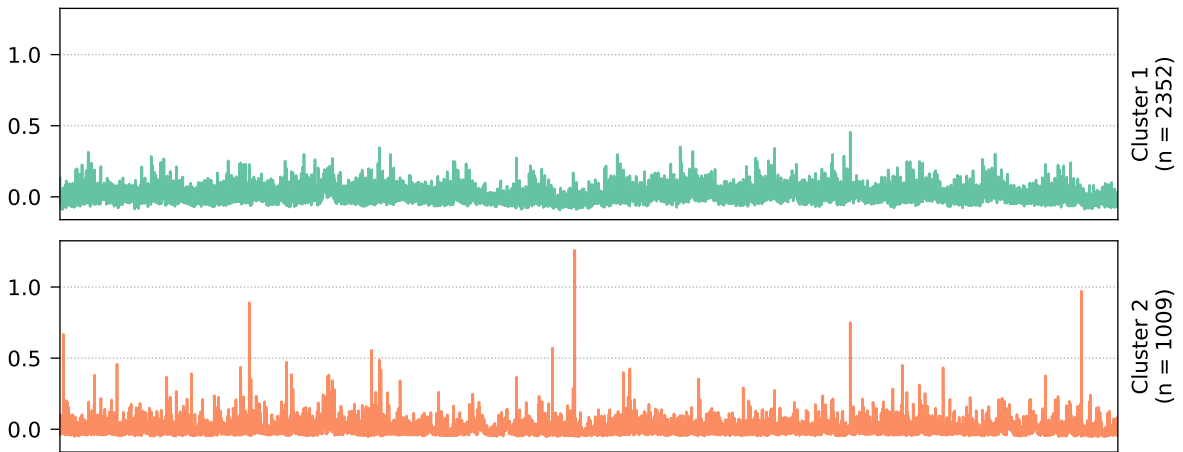


(b) Cluster feature values of all TSC (sub)groups, clipped to $[0, 1]$. Abbreviations: d = distributional, t = temporal, c = complexity, b = blockwise.

Figure D.9: Various results obtained when clustering the 7176 Memory Available % (H-07) series of the IMTS₁ dataset into six clusters. The respective cluster sizes are denoted by n .



(a) Dendrogram (left) and representative time series (right) for the two identified clusters.

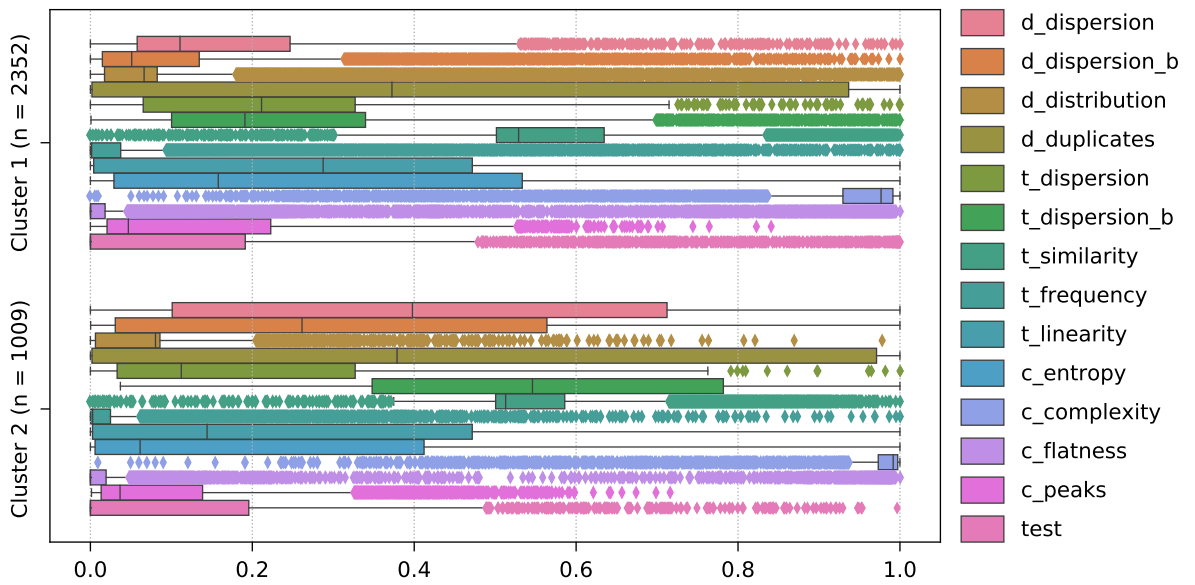


(b) Cluster time series averages.



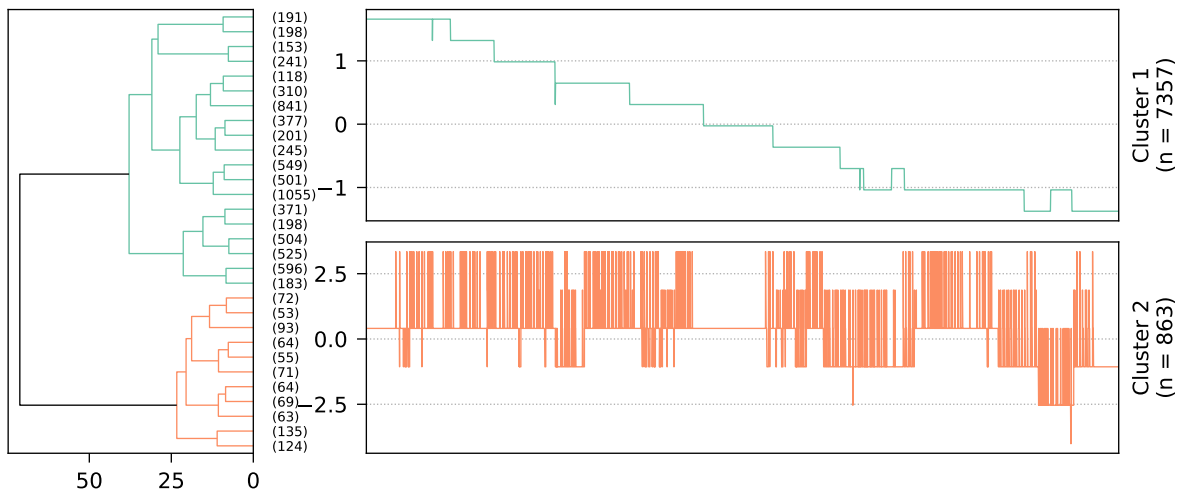
(c) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

(d) Venn diagram showing the distribution of the 437 systems.

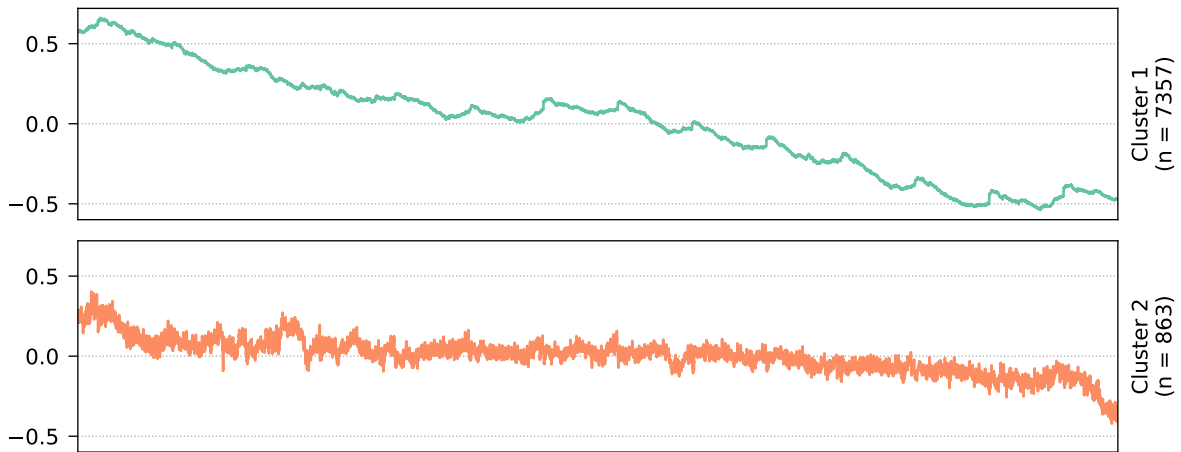


(e) Cluster feature values of all TSC (sub)groups, clipped to $[0, 1]$. Abbreviations: *d* = distributional, *t* = temporal, *c* = complexity, *b* = blockwise.

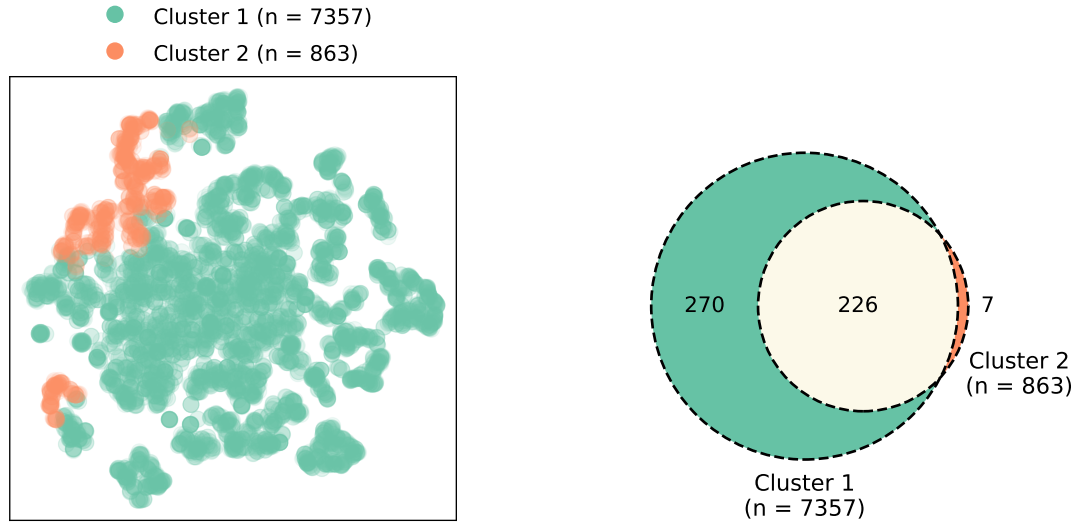
Figure D.10: Various results obtained when clustering the 3361 Page Faults (H-06) series of the IMTS₁ dataset into two clusters. The respective cluster sizes are denoted by *n*, and all time series contain 20160 data points (two weeks in one-minute resolution, ranging from 22.01.2018 00:00 UTC to 04.02.2018 23:59 UTC).



(a) Dendrogram (left) and representative time series (right) for the two identified clusters.

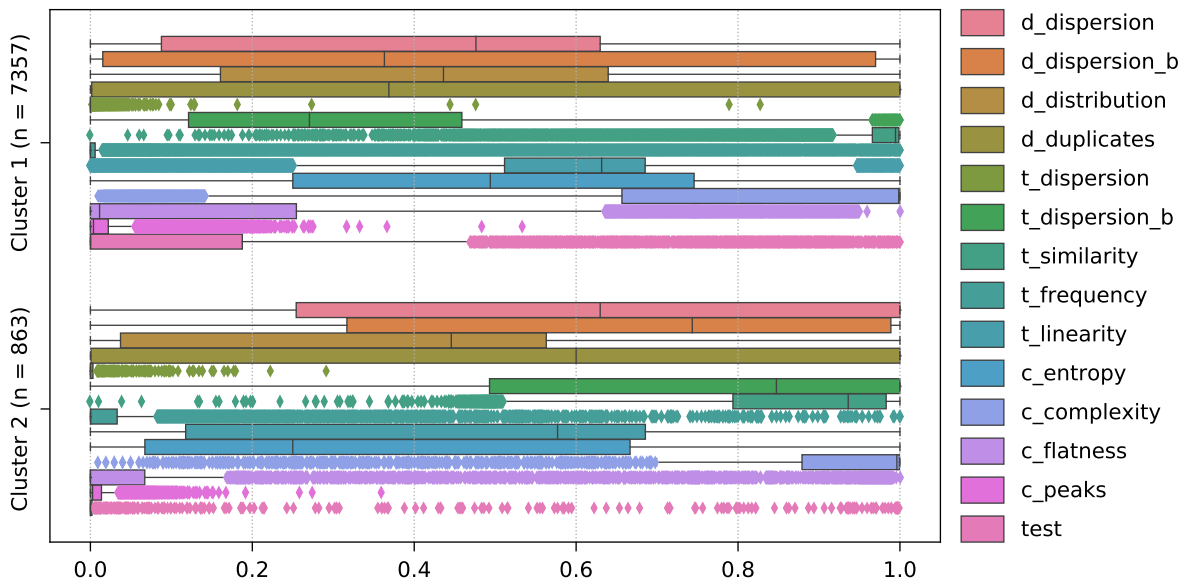


(b) Cluster time series averages.



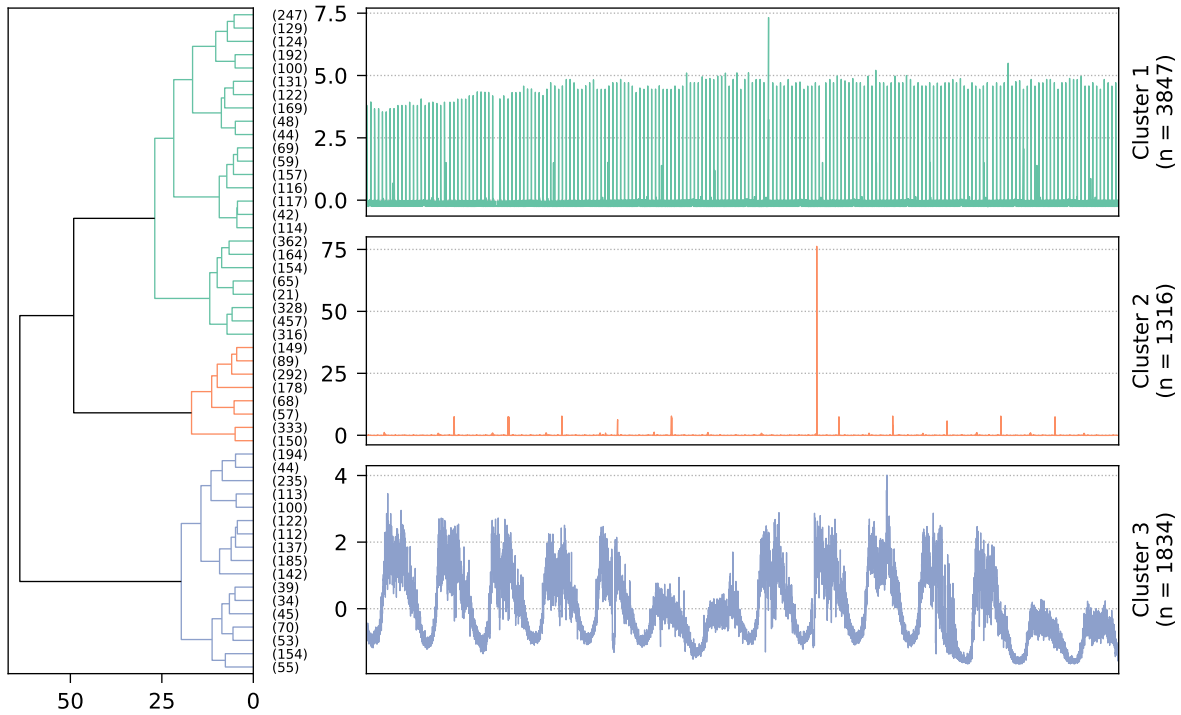
(c) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

(d) Venn diagram showing the distribution of the 503 systems.

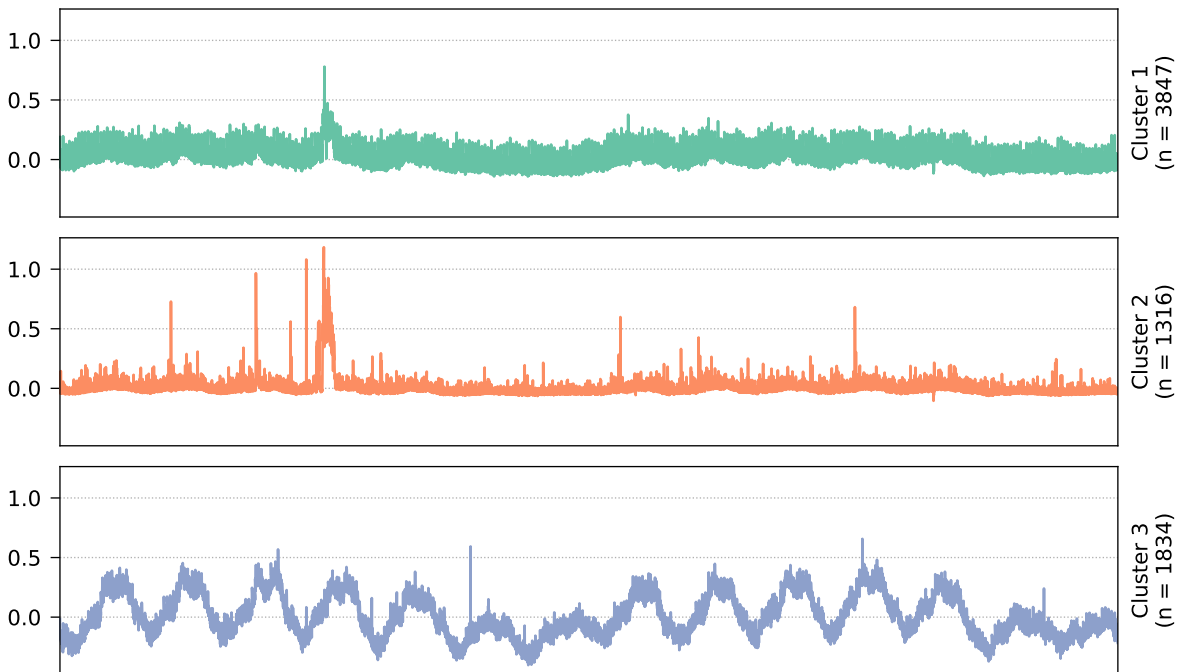


(e) Cluster feature values of all TSC (sub)groups, clipped to [0, 1]. Abbreviations: *d* = distributional, *t* = temporal, *c* = complexity, *b* = blockwise.

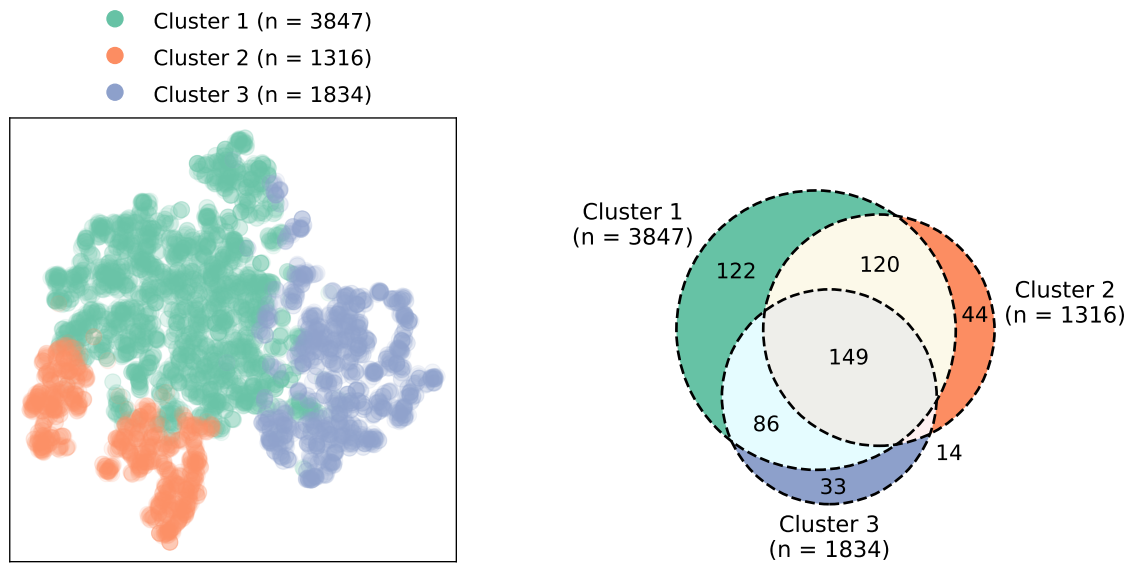
Figure D.11: Various results obtained when clustering the 8220 Disk Available % (D-03) series of the IMTS₁ dataset into two clusters. The respective cluster sizes are denoted by *n*, and all time series contain 20160 data points (two weeks in one-minute resolution, ranging from 22.01.2018 00:00 UTC to 04.02.2018 23:59 UTC).



(a) Dendrogram (left) and representative time series (right) for the three identified clusters.

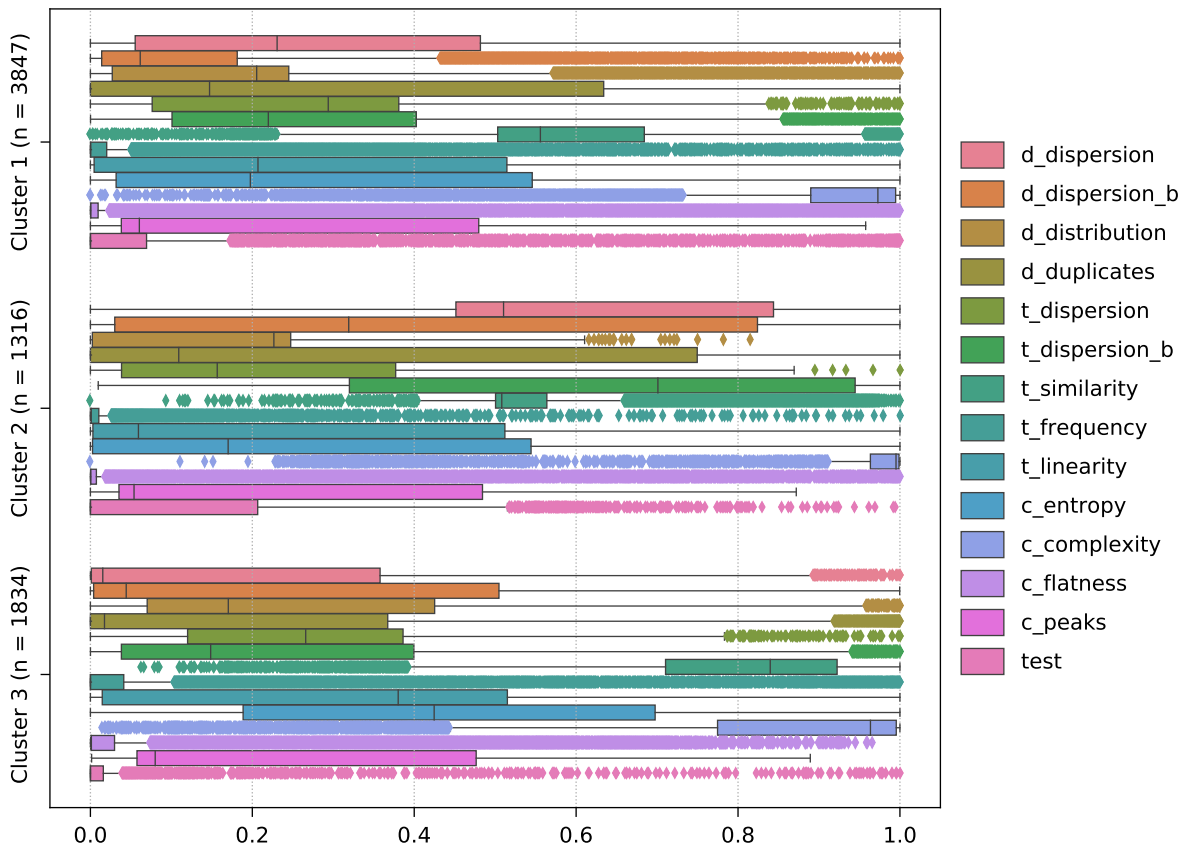


(b) Cluster time series averages.



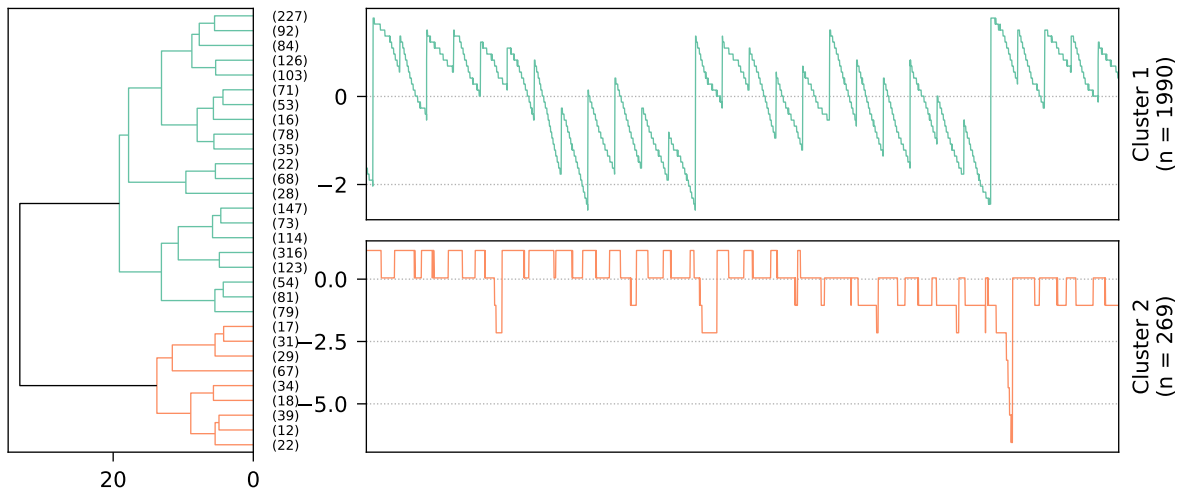
(c) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

(d) Venn diagram showing the distribution of the 568 systems.

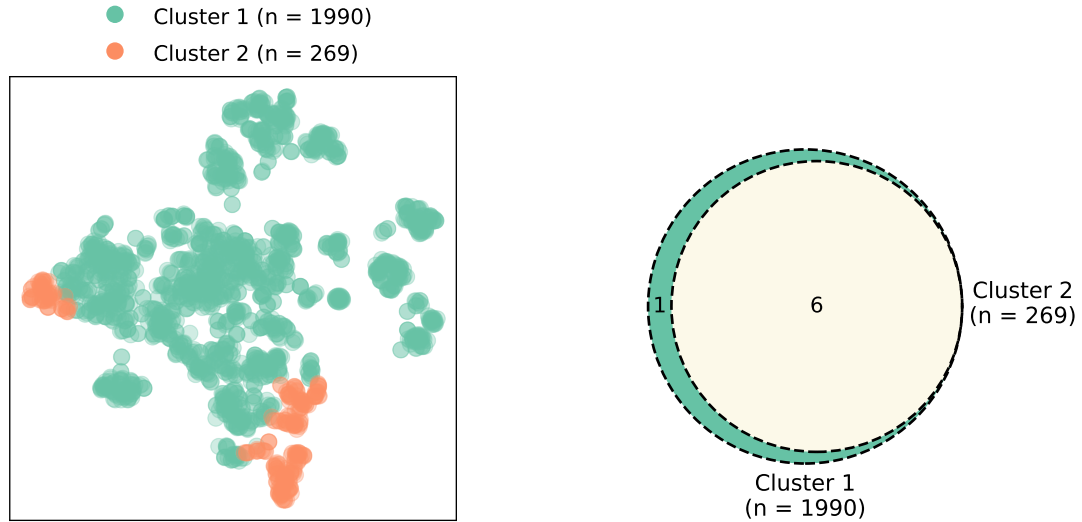


(e) Cluster feature values of all TSC (sub)groups, clipped to [0, 1]. Abbreviations: *d* = distributional, *t* = temporal, *c* = complexity, *b* = blockwise.

Figure D.12: Various results obtained when clustering the 6997 Bytes Received (N-01) series of the IMTS₁ dataset into three clusters. The respective cluster sizes are denoted by *n*, and all time series contain 20160 data points (two weeks in one-minute resolution, ranging from 22.01.2018 00:00 UTC to 04.02.2018 23:59 UTC).

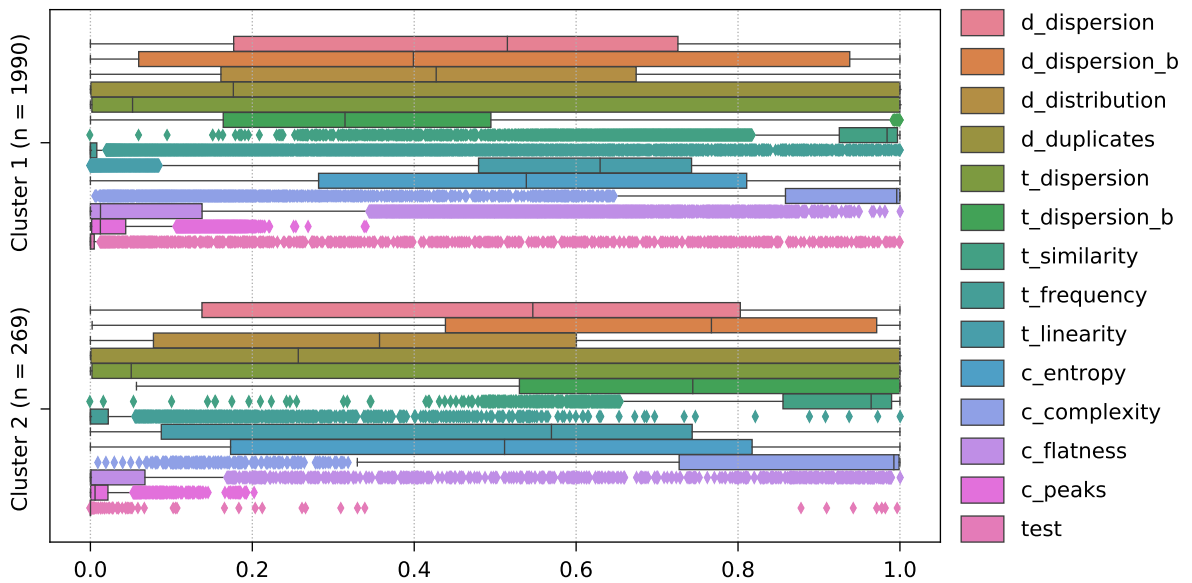


(a) Dendrogram (left) and representative time series (right) for the two identified clusters.



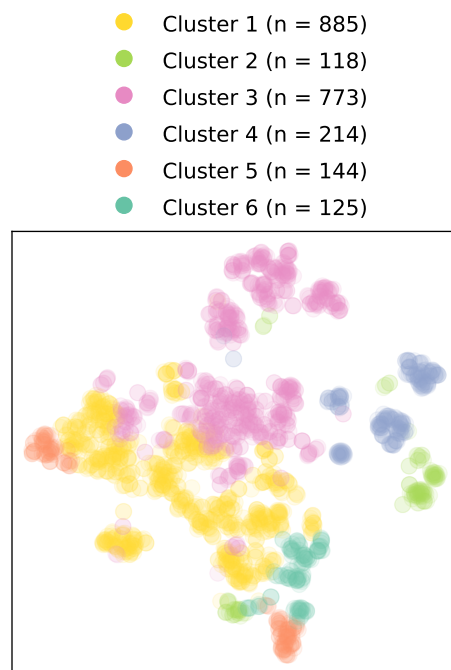
(b) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

(c) Venn diagram showing the distribution of the 7 systems.

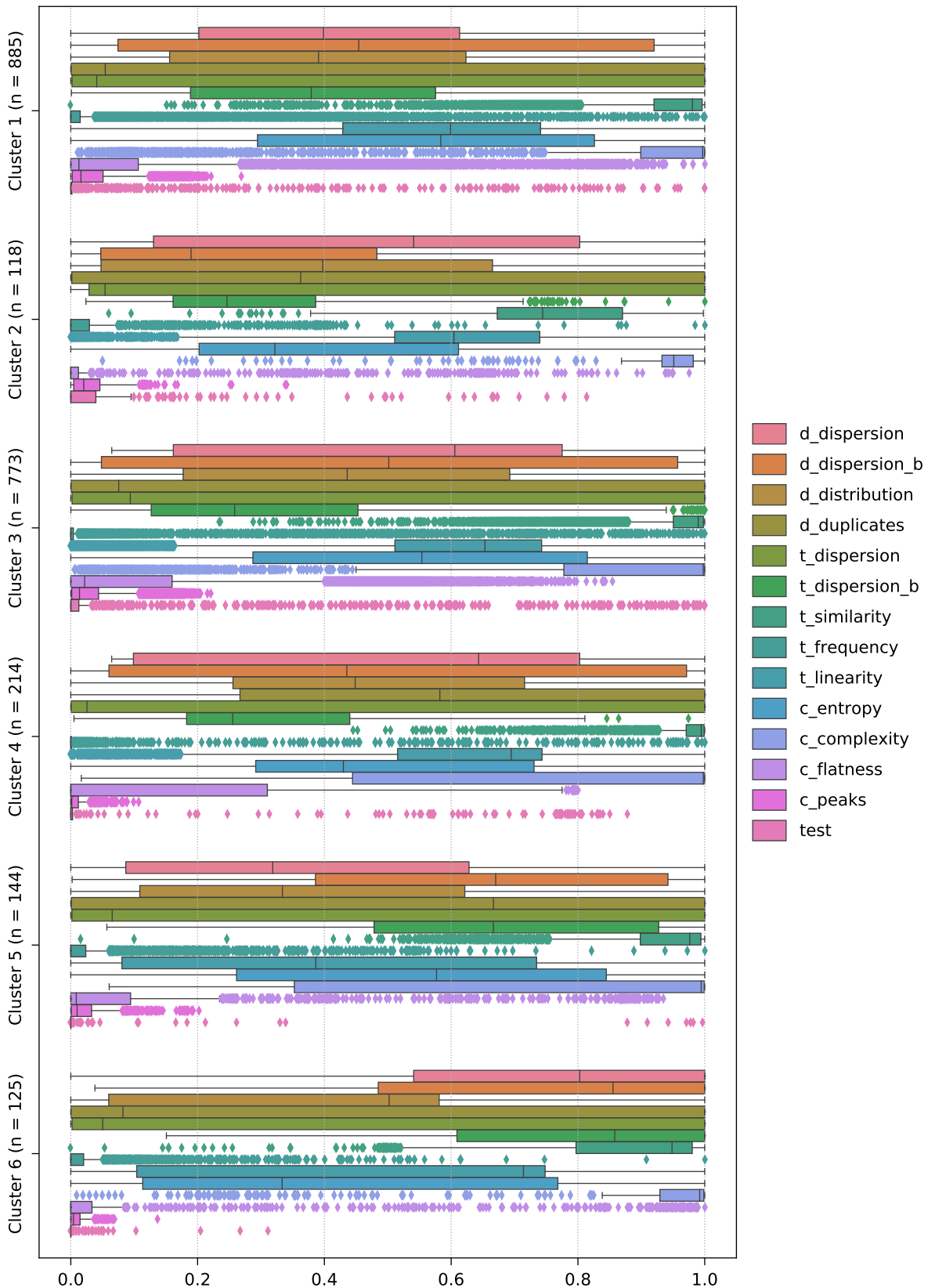


(d) Cluster feature values of all TSC (sub)groups, clipped to [0, 1]. Abbreviations: *d* = distributional, *t* = temporal, *c* = complexity, *b* = blockwise.

Figure D.13: Various results obtained when clustering the 2259 Disk Available % (D-03) series of the IMTS₂ dataset into two clusters. The respective cluster sizes are denoted by *n*, and all time series contain 40320 data points (four weeks in one-minute resolution, ranging from 15.07.2019 00:00 UTC to 11.08.2019 23:59 UTC).

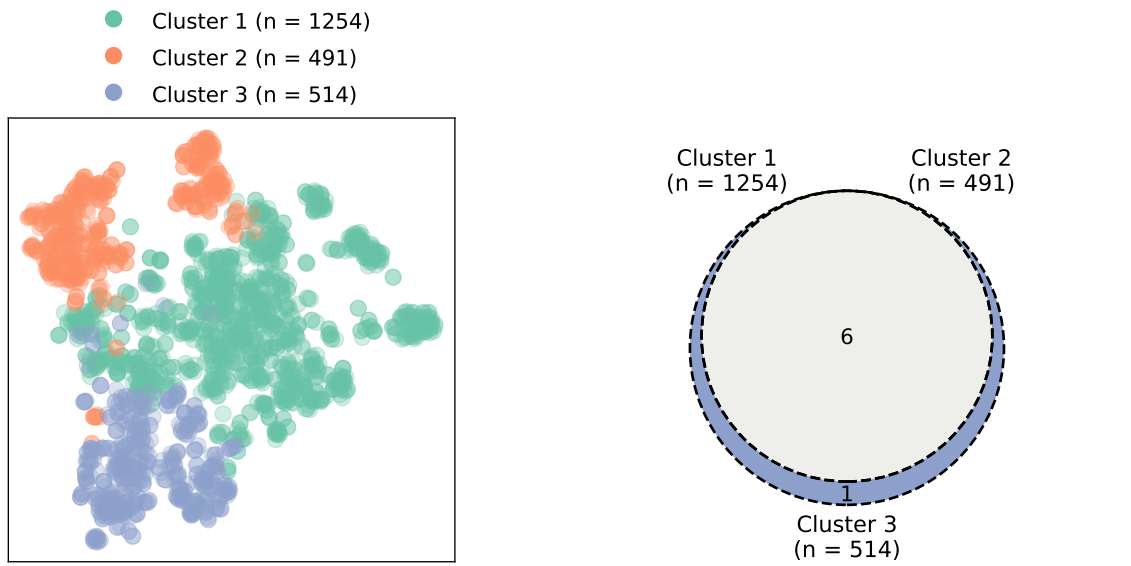


(a) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.



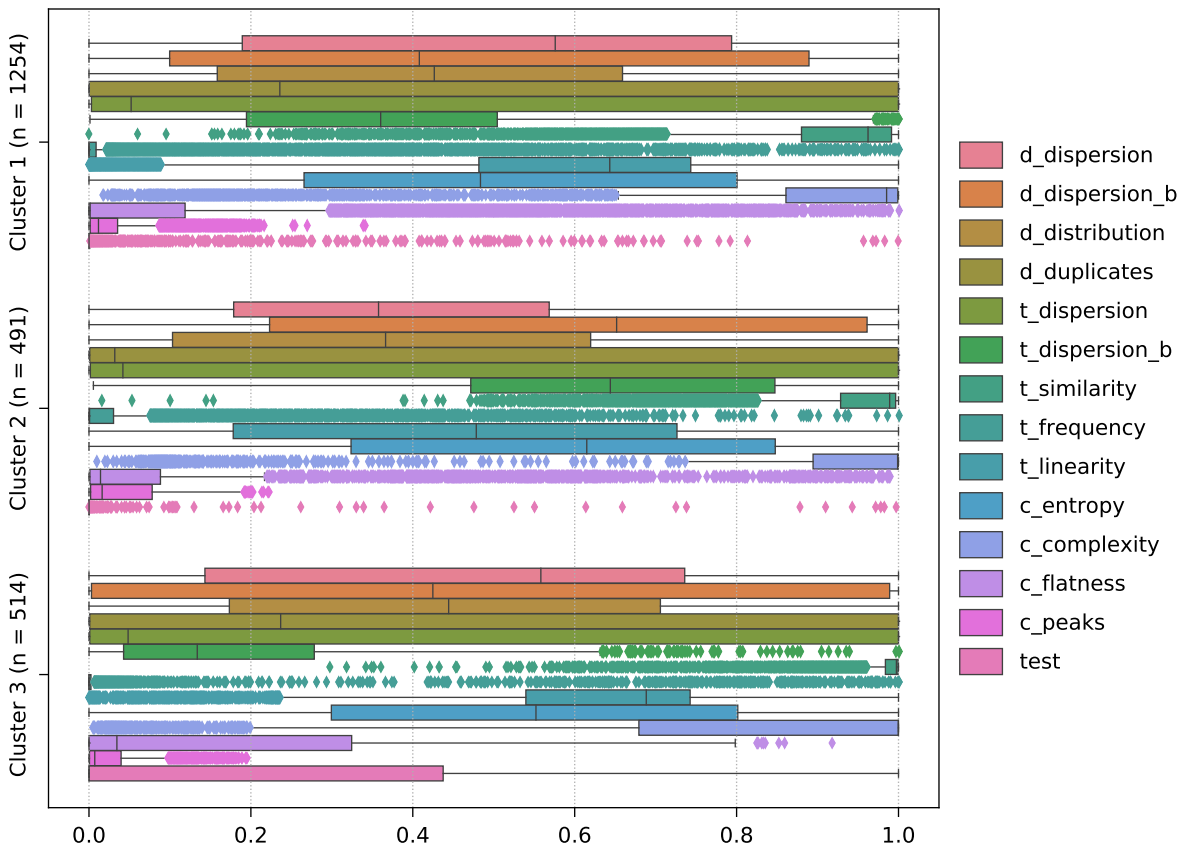
(b) Cluster feature values of all TSC (sub)groups, clipped to $[0, 1]$. Abbreviations: d = distributional, t = temporal, c = complexity, b = blockwise.

Figure D.14: Various results obtained when clustering the 2259 Disk Available % (D-03) series of the IMTS₁ dataset into six clusters. The respective cluster sizes are denoted by n .



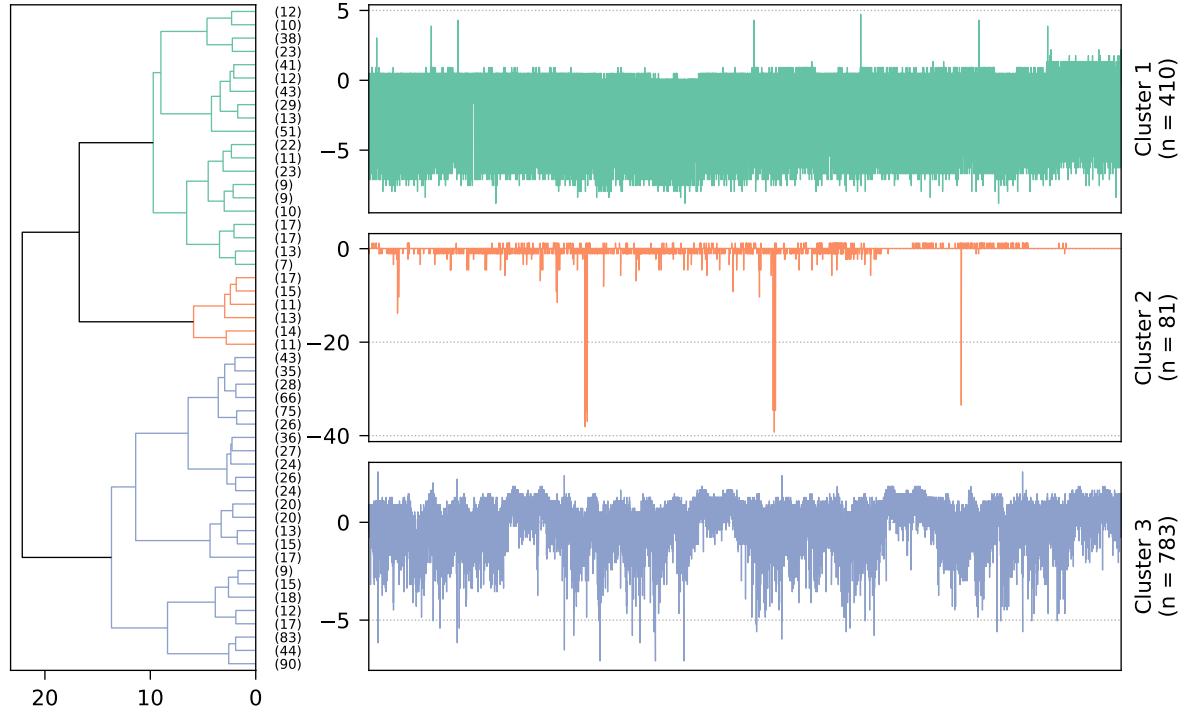
(a) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

(b) Venn diagram showing the distribution of the 7 systems.

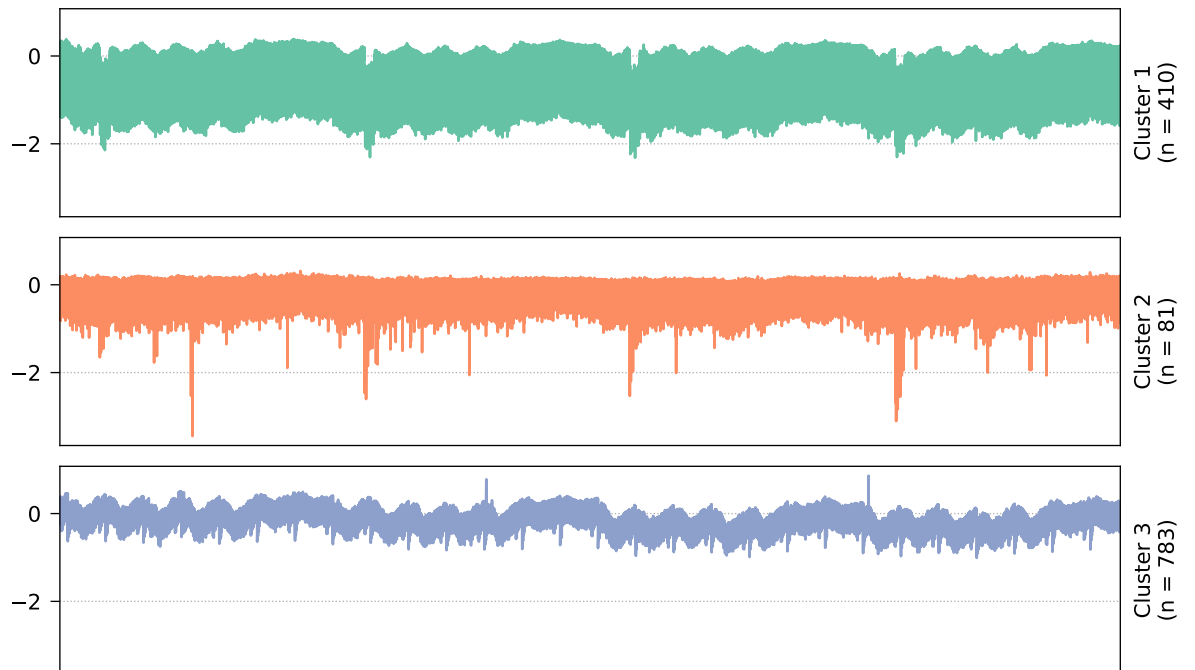


(c) Cluster feature values of all TSC (sub)groups, clipped to $[0, 1]$. Abbreviations: *d* = distributional, *t* = temporal, *c* = complexity, *b* = blockwise.

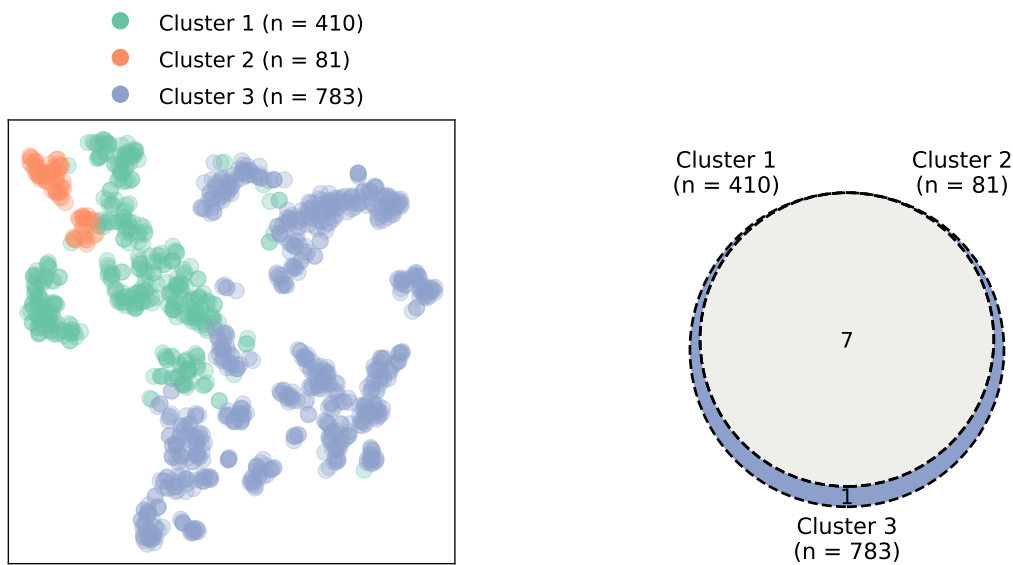
Figure D.15: Various results obtained when clustering the 2259 Disk Available % (D-03) series of the IMTS₂ dataset into three clusters using the entire set of TSC features instead of the *distributional* feature set. The respective cluster sizes are denoted by *n*.



(a) Dendrogram (left) and representative time series (right) for the three identified clusters.

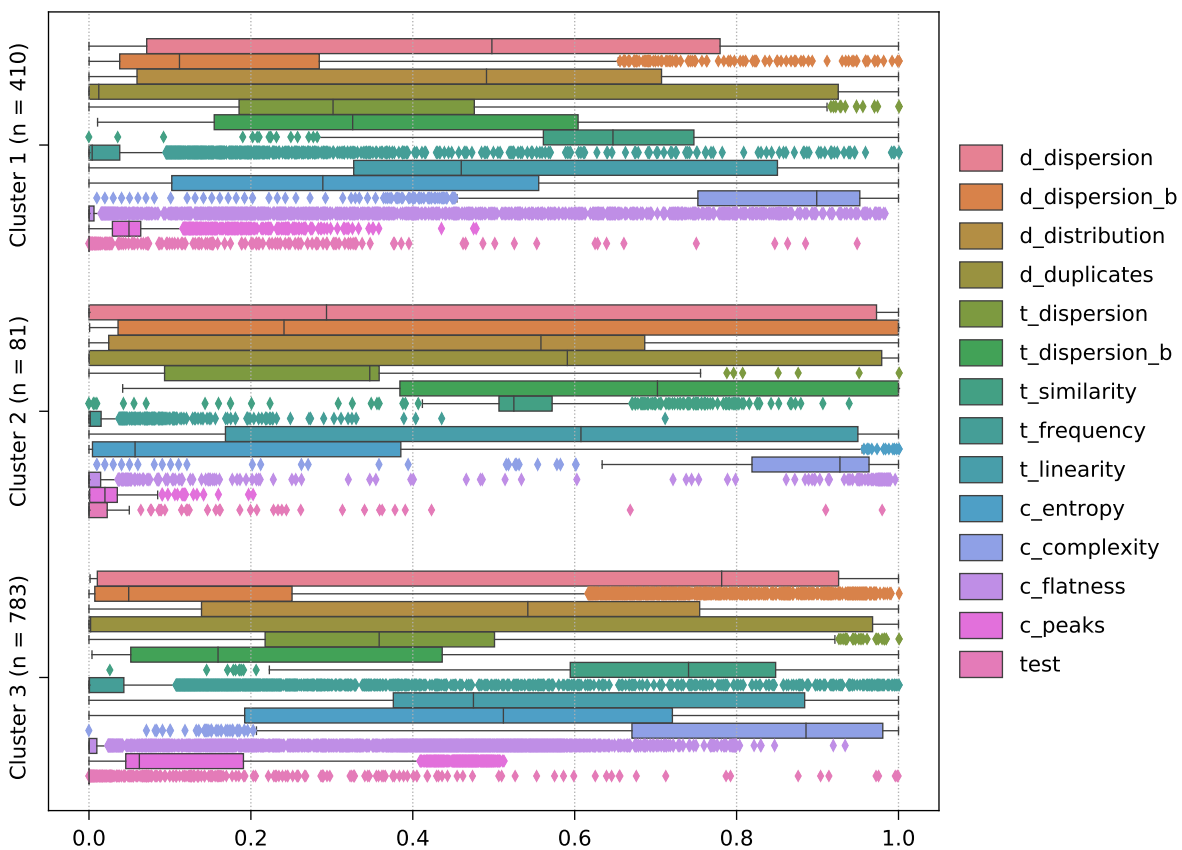


(b) Cluster time series averages.



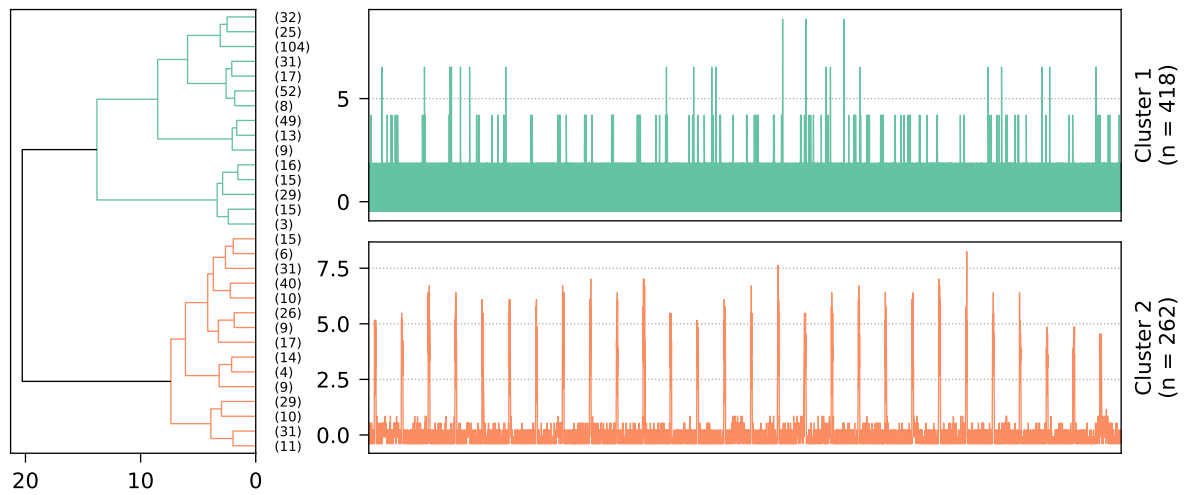
(c) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

(d) Venn diagram showing the distribution of the 8 systems.



(e) Cluster feature values of all TSC (sub)groups, clipped to $[0, 1]$. Abbreviations: d = distributional, t = temporal, c = complexity, b = blockwise.

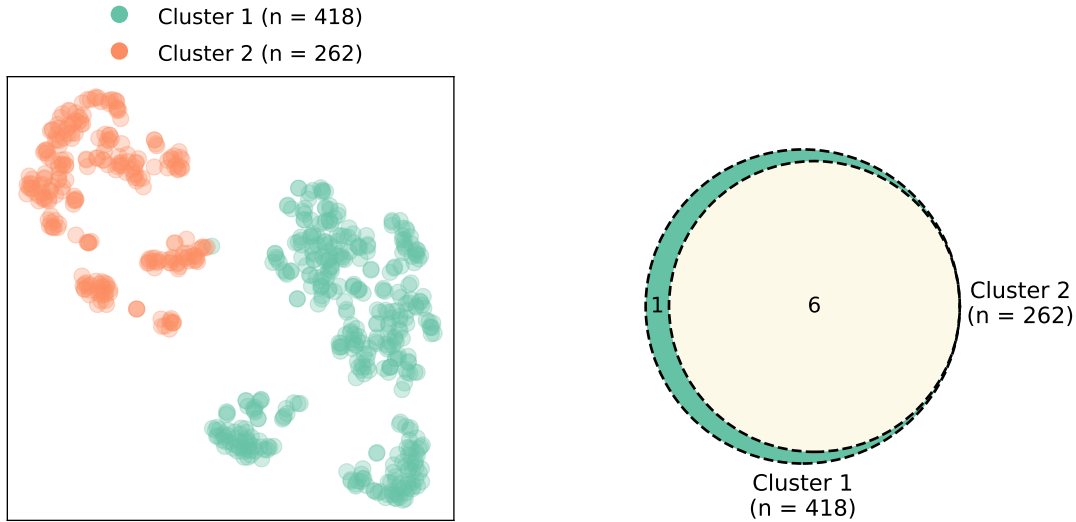
Figure D.16: Various results obtained when clustering the 1274 CPU Idle (H-01) series of the IMTS₂ dataset into three clusters. The respective cluster sizes are denoted by n , and all time series contain 40320 data points (four weeks in one-minute resolution, ranging from 15.07.2019 00:00 UTC to 11.08.2019 23:59 UTC).



(a) Dendrogram (left) and representative time series (right) for the two identified clusters.

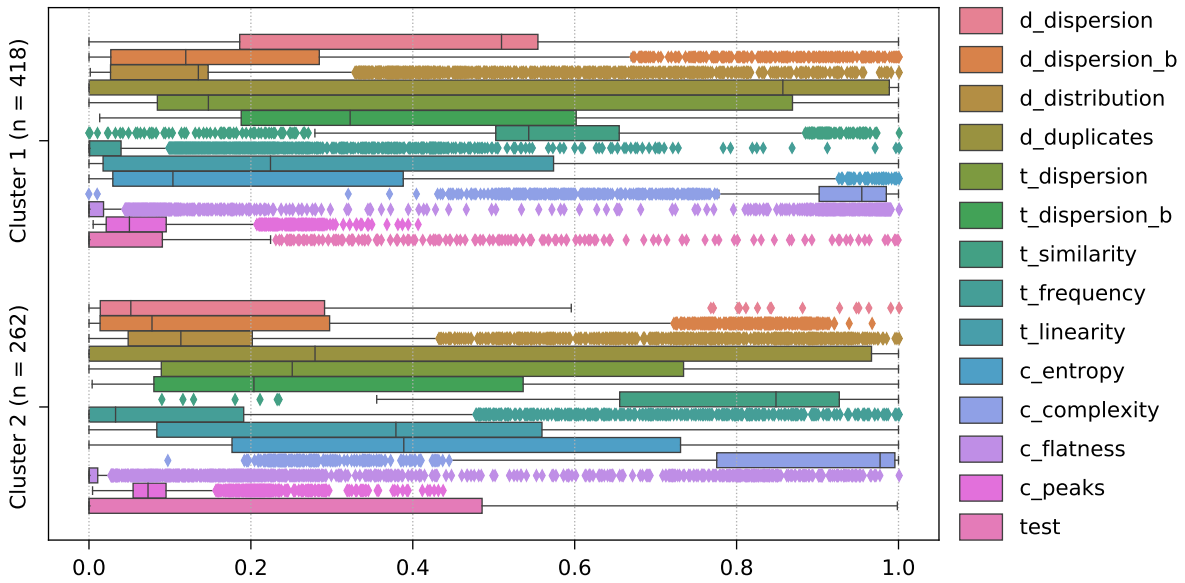


(b) Cluster time series averages.



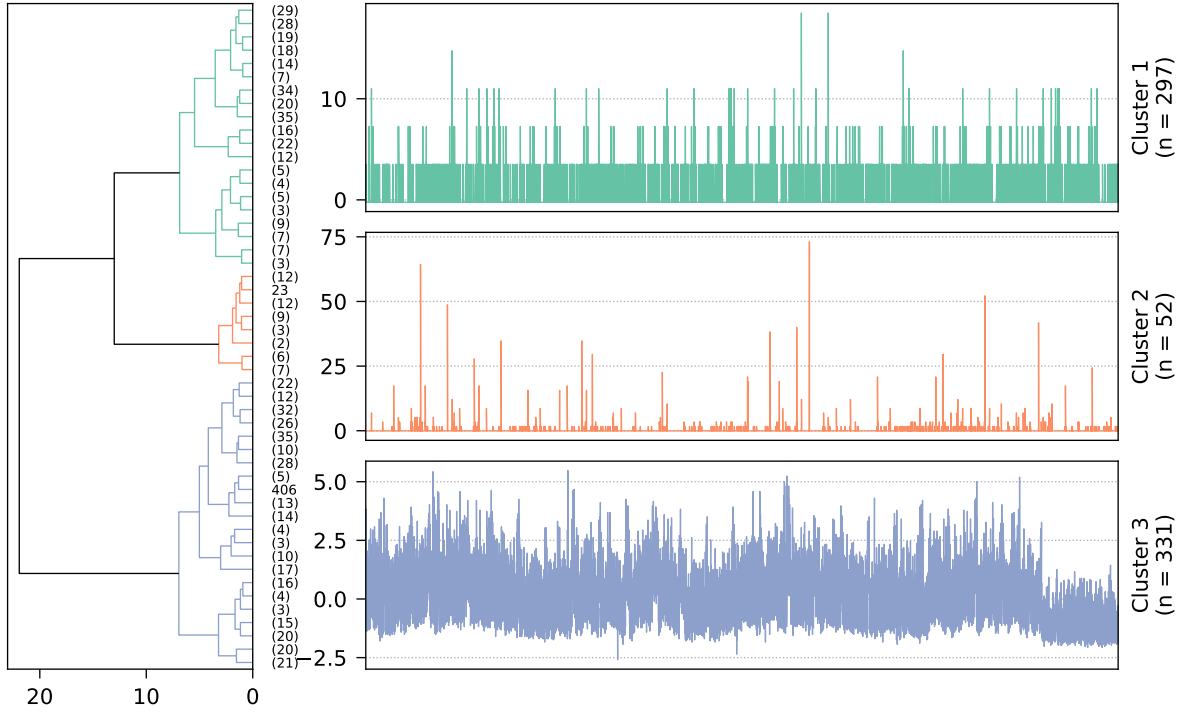
(c) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

(d) Venn diagram showing the distribution of the 7 systems.

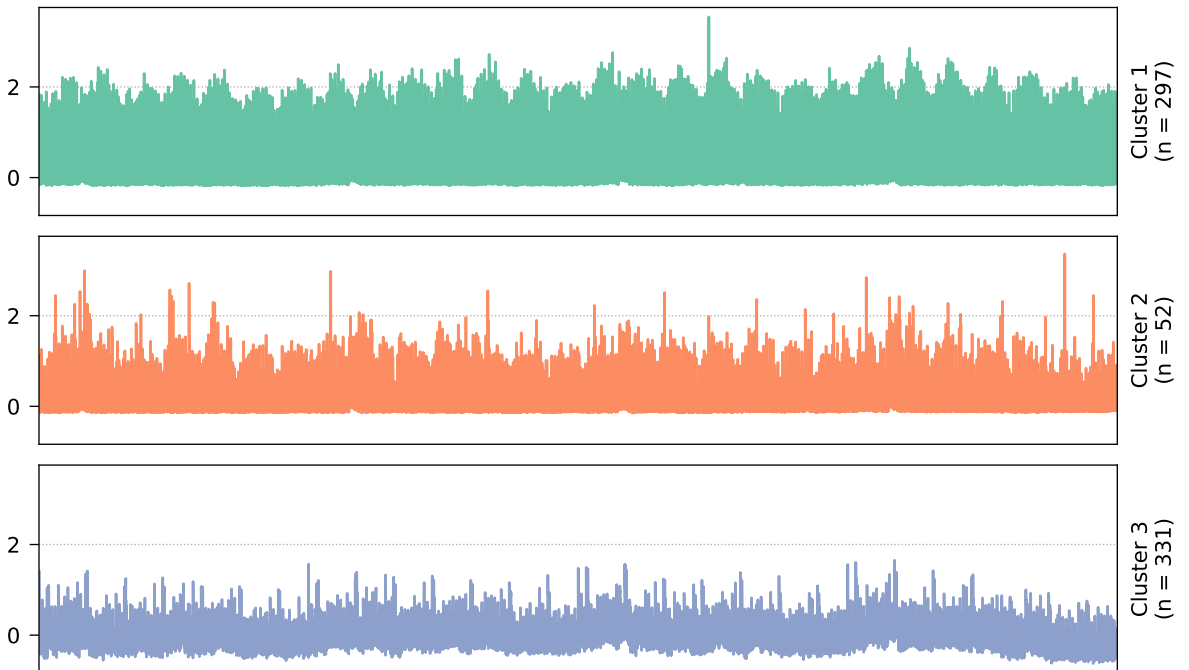


(e) Cluster feature values of all TSC (sub)groups, clipped to $[0, 1]$. Abbreviations: d = distributional, t = temporal, c = complexity, b = blockwise.

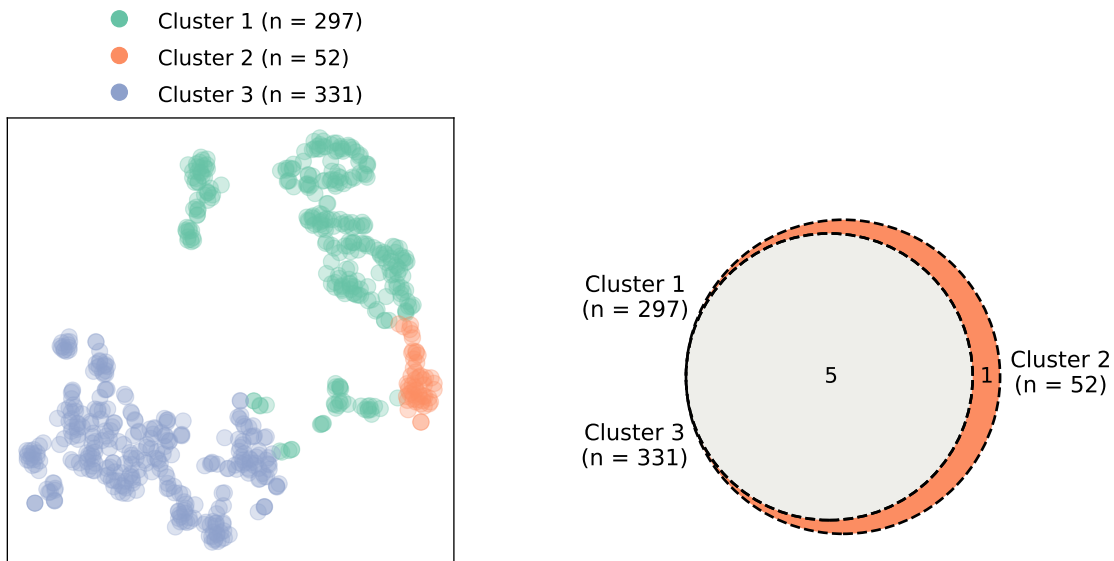
Figure D.17: Various results obtained when clustering the 680 CPU IO Wait (H-05) series of the IMTS₂ dataset into two clusters. The respective cluster sizes are denoted by n , and all time series contain 40320 data points (four weeks in one-minute resolution, ranging from 15.07.2019 00:00 UTC to 11.08.2019 23:59 UTC).



(a) Dendrogram (left) and representative time series (right) for the three identified clusters.

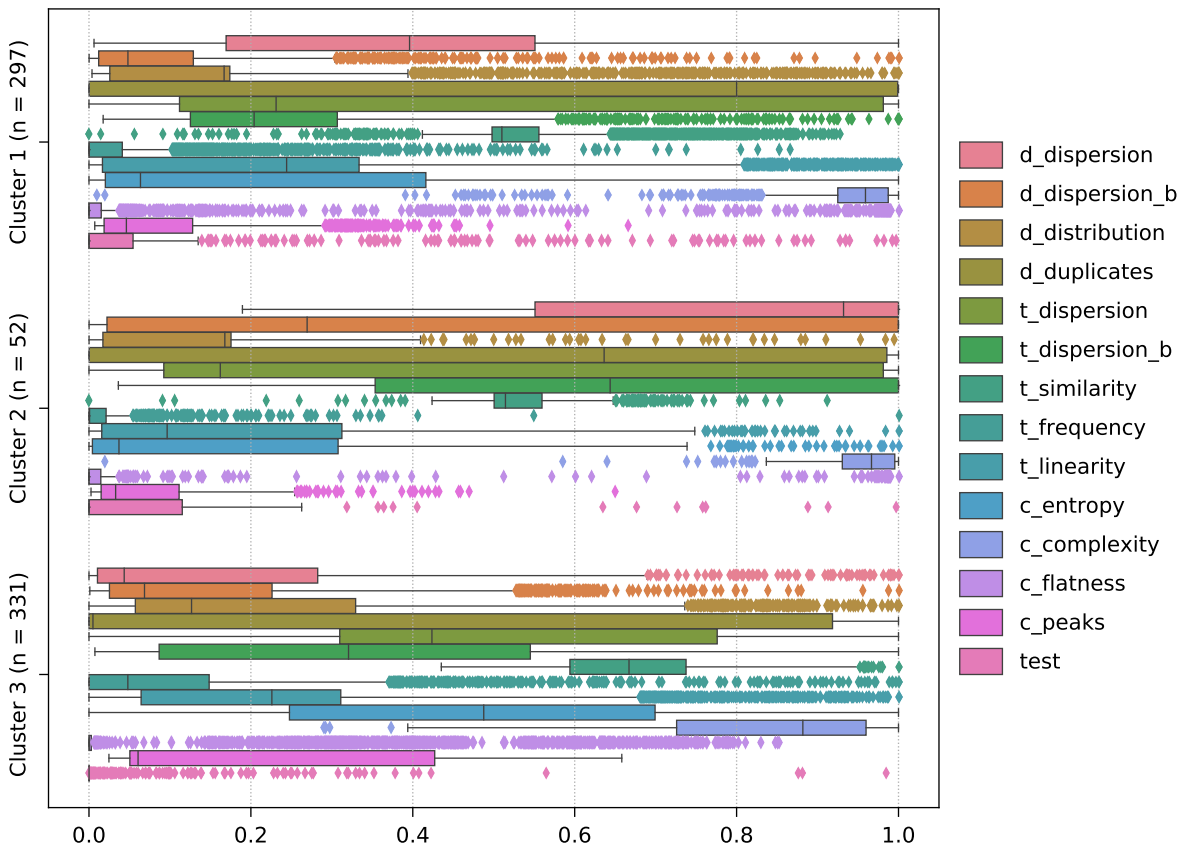


(b) Cluster time series averages.



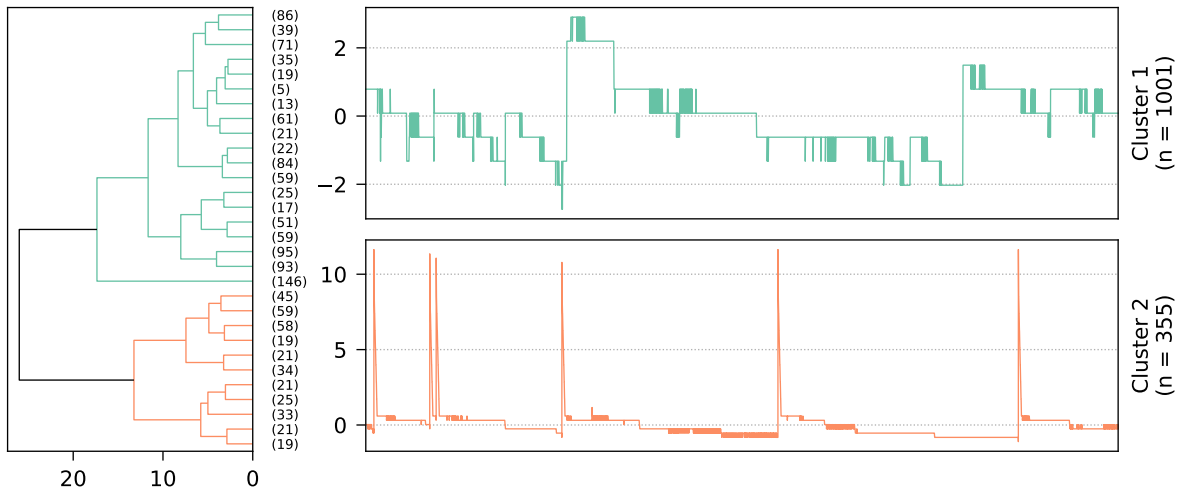
(c) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

(d) Venn diagram showing the distribution of the 6 systems.

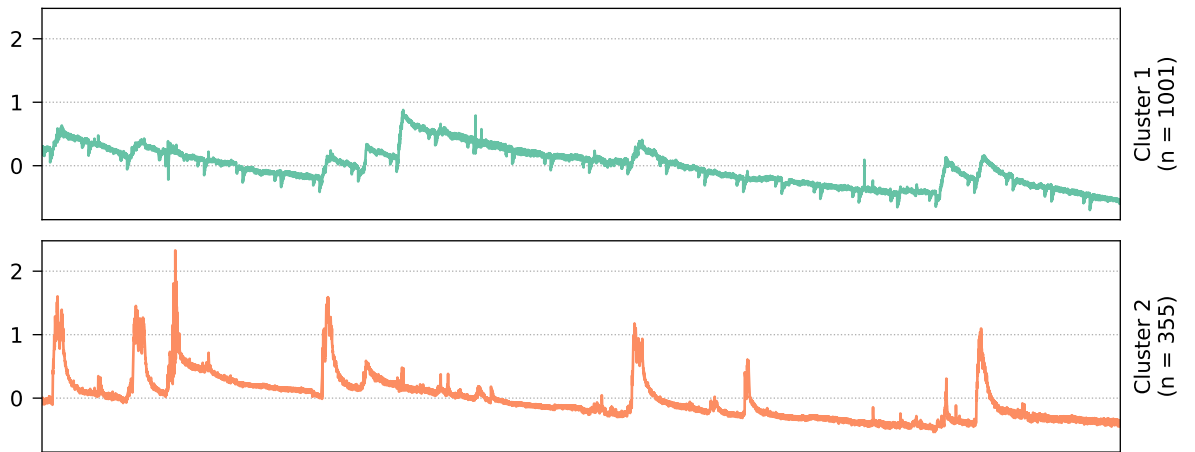


(e) Cluster feature values of all TSC (sub)groups, clipped to [0, 1]. Abbreviations: *d* = distributional, *t* = temporal, *c* = complexity, *b* = blockwise.

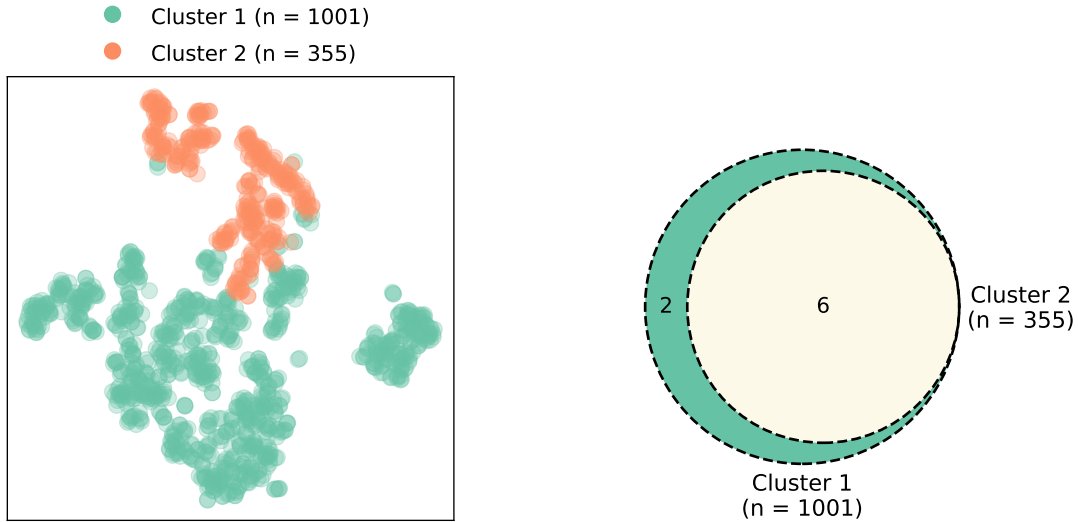
Figure D.18: Various results obtained when clustering the 680 Page Faults (H-06) series of the IMTS₂ dataset into three clusters. The respective cluster sizes are denoted by *n*, and all time series contain 40320 data points (four weeks in one-minute resolution, ranging from 15.07.2019 00:00 UTC to 11.08.2019 23:59 UTC).



(a) Dendrogram (left) and representative time series (right) for the two identified clusters.

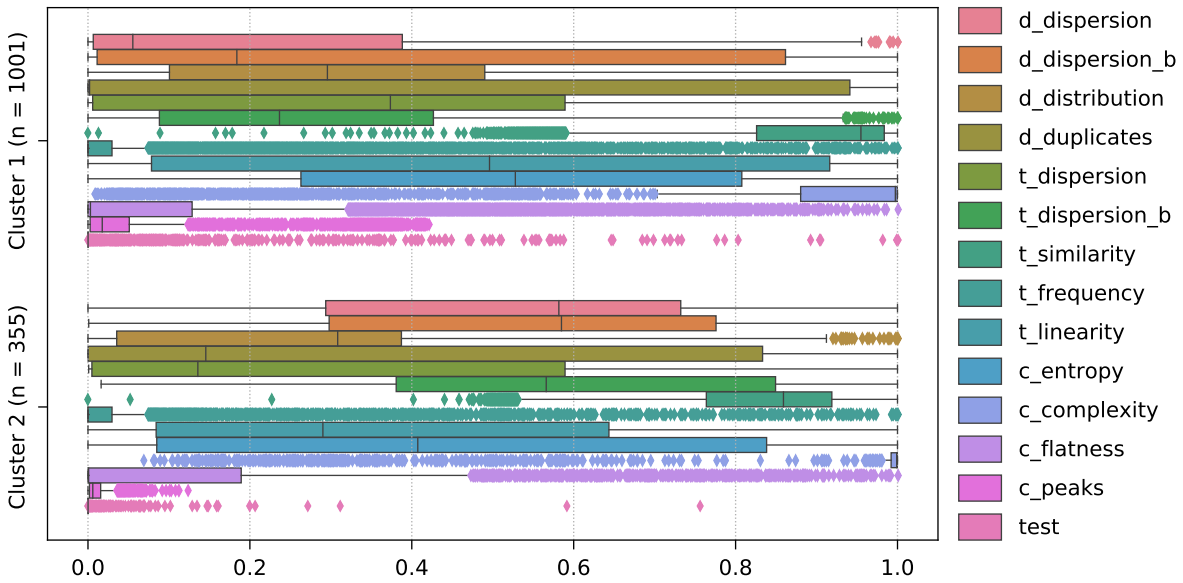


(b) Cluster time series averages.



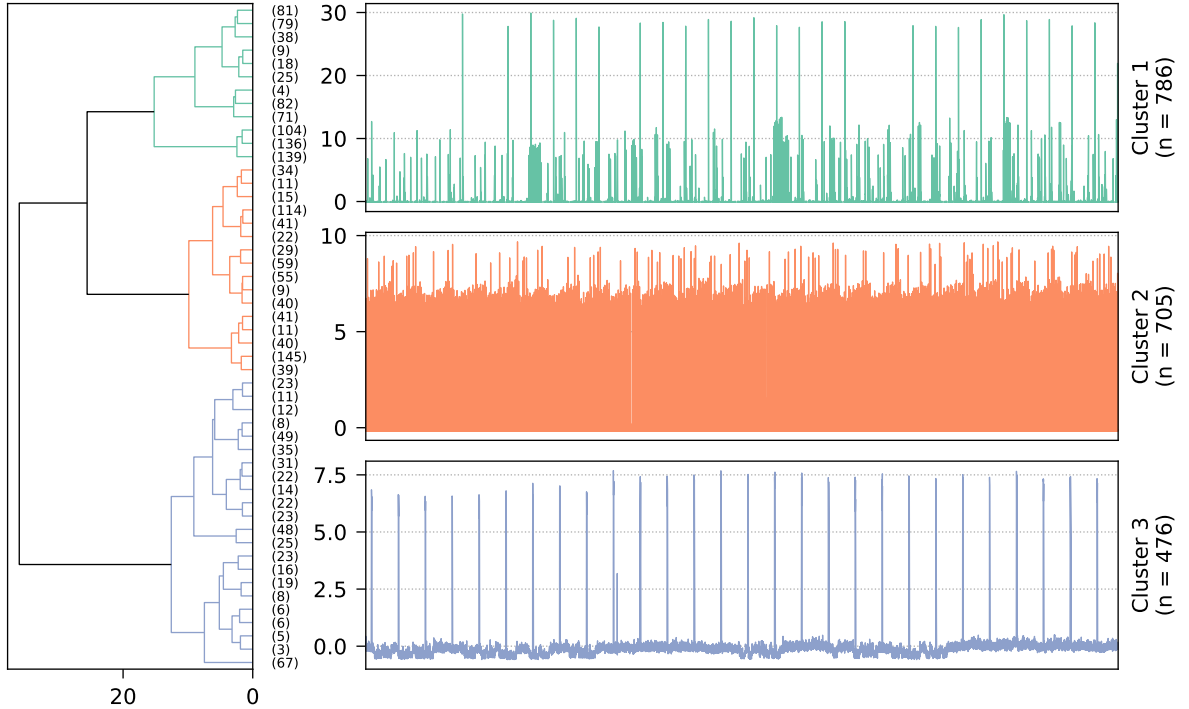
(c) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

(d) Venn diagram showing the distribution of the 8 systems.

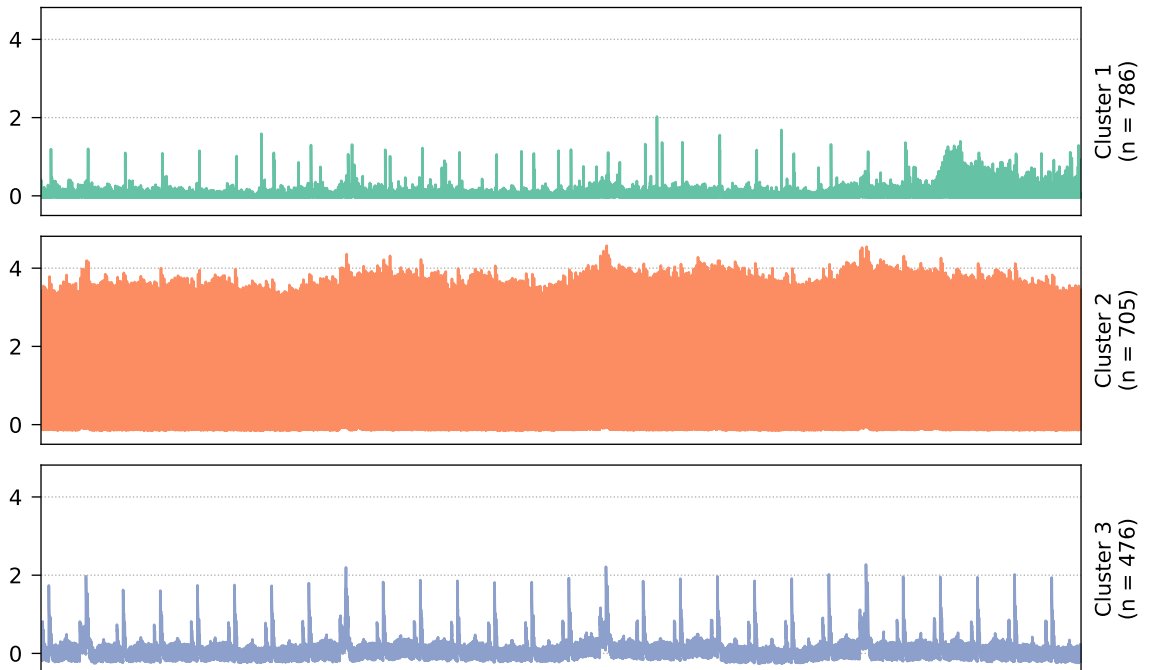


(e) Cluster feature values of all TSC (sub)groups. Abbreviations: *d* = distributional, *t* = temporal, *c* = complexity, *b* = blockwise.

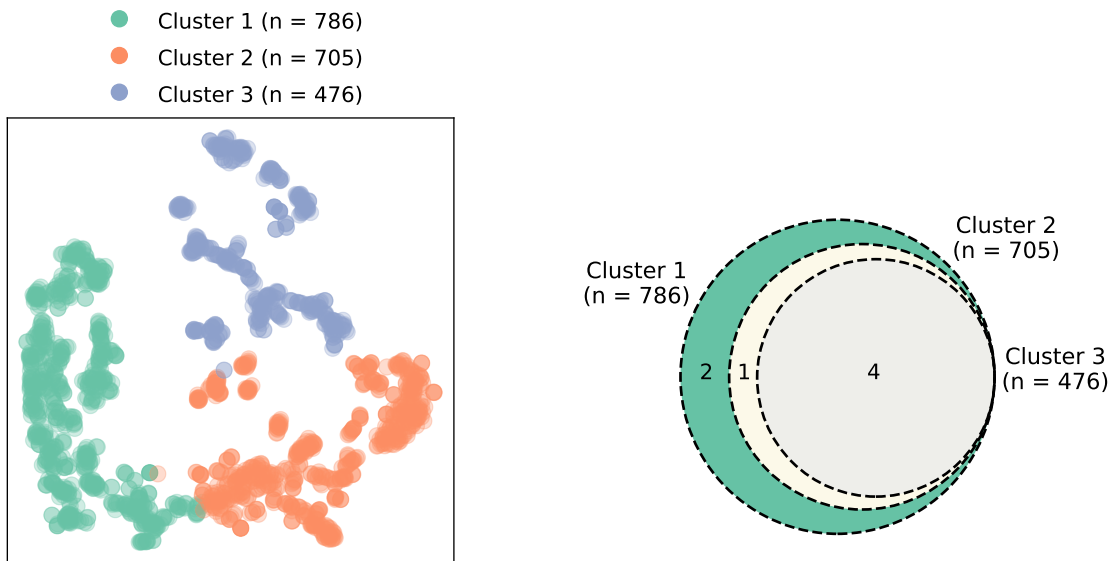
Figure D.19: Various results obtained when clustering the 1356 Memory Available % (H-07) series of the IMTS₂ dataset into two clusters. The respective cluster sizes are denoted by *n*, and all time series contain 40320 data points (four weeks in one-minute resolution, ranging from 15.07.2019 00:00 UTC to 11.08.2019 23:59 UTC).



(a) Dendrogram (left) and representative time series (right) for the three identified clusters.

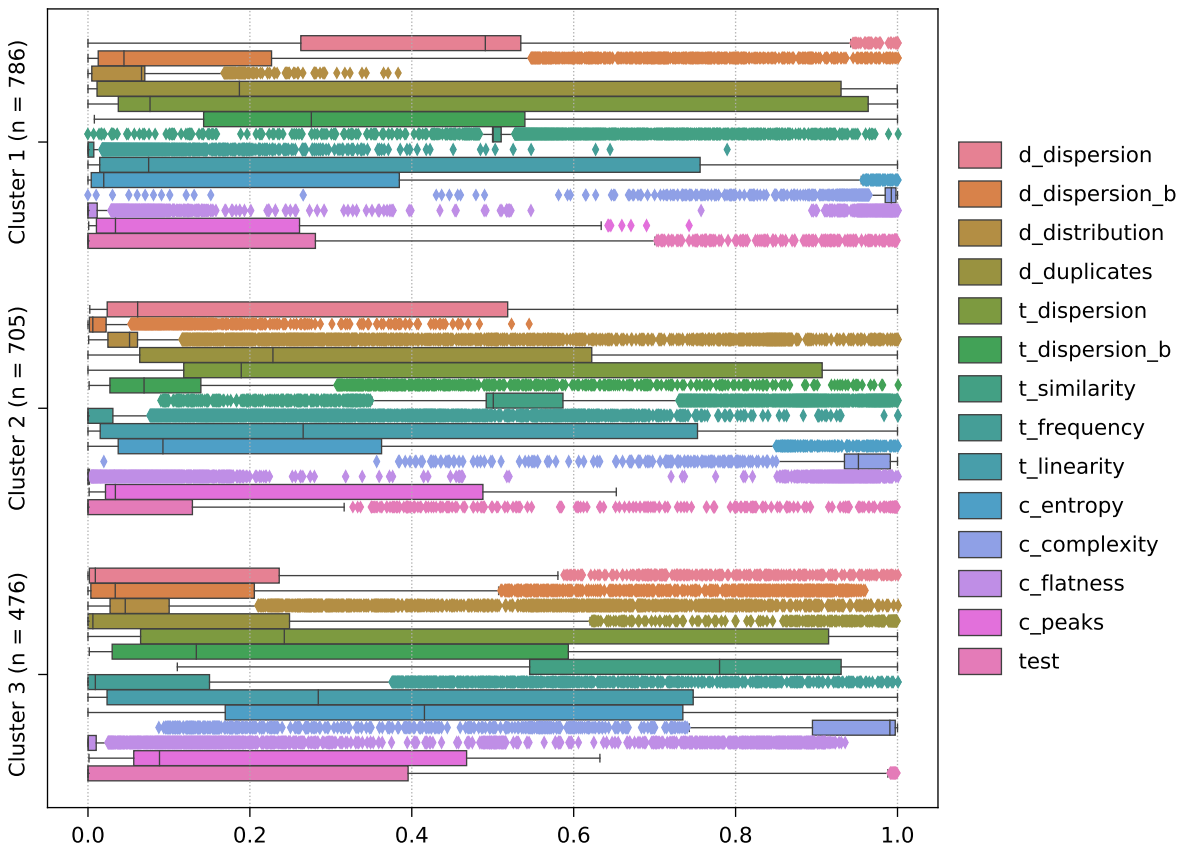


(b) Cluster time series averages.



(c) t-SNE visualization (perplexity = 30) of the *distributional* feature set. The samples are tinted according to their corresponding clusters.

(d) Venn diagram showing the distribution of the 7 systems.



(e) Cluster feature values of all TSC (sub)groups, clipped to [0, 1]. Abbreviations: *d* = distributional, *t* = temporal, *c* = complexity, *b* = blockwise.

Figure D.20: Various results obtained when clustering the 1967 Read Bytes (D-04) series of the IMTS₂ dataset into three clusters. The respective cluster sizes are denoted by *n*, and all time series contain 40320 data points (four weeks in one-minute resolution, ranging from 15.07.2019 00:00 UTC to 11.08.2019 23:59 UTC).

Bibliography

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. “TensorFlow: A System for Large-Scale Machine Learning”. In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016, pp. 265–283.
- [2] Salisu Mamman Abdulrahman, Pavel Brazdil, Wan Mohd Nazmee Wan Zainon, and Alhassan Adamu. “Simplifying the Algorithm Selection Using Reduction of Rankings of Classification Algorithms”. In: *Proceedings of the 8th International Conference on Software and Computer Applications*. ACM, 2019, pp. 140–148. DOI: [10.1145/3316615.3316674](https://doi.org/10.1145/3316615.3316674).
- [3] Saeed Aghabozorgi, Ali Seyed Shirkhorshidi, and Teh Ying Wah. “Time-series clustering — A decade review”. In: *Information Systems* 53 (2015), pp. 16–38. DOI: [10.1016/J.IS.2015.04.007](https://doi.org/10.1016/J.IS.2015.04.007).
- [4] Javier Alonso, Lluís Belanche, and Dimiter R. Avresky. “Predicting Software Anomalies using Machine Learning Techniques”. In: *Proceedings of the 10th IEEE International Symposium on Network Computing and Applications*. IEEE, 2011, pp. 163–170. DOI: [10.1109/NCA.2011.29](https://doi.org/10.1109/NCA.2011.29).
- [5] Ayman Amin, Lars Grunske, and Alan Colman. “An approach to software reliability prediction based on time series modeling”. In: *Journal of Systems and Software* 86.7 (2013), pp. 1923–1932. DOI: [10.1016/J.JSS.2013.03.045](https://doi.org/10.1016/J.JSS.2013.03.045).
- [6] David Arthur and Sergei Vassilvitskii. “k-means++: The Advantages of Careful Seeding”. In: *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*. 2007, pp. 1027–1035.
- [7] Merve Astekin, Harun Zengin, and Hasan Sözer. “DILAF: A framework for distributed analysis of large-scale system logs for anomaly detection”. In: *Software - Practice and Experience* 49.2 (2019), pp. 153–170. DOI: [10.1002/SPE.2653](https://doi.org/10.1002/SPE.2653).
- [8] Francisco Javier Baldan and José Manuel Benítez. “Complexity Measures and Features for Times Series classification”. In: *CoRR* abs/2002.12036 (2020), pp. 1–22. arXiv: [2002.12036](https://arxiv.org/abs/2002.12036). URL: <https://arxiv.org/abs/2002.12036>.
- [9] Kasun Bandara, Christoph Bergmeir, and Slawek Smyl. “Forecasting across time series databases using recurrent neural networks on groups of similar series: A clustering approach”. In: *Expert Systems with Applications* 140 (2020), 112896:1–16. DOI: [10.1016/J.ESWA.2019.112896](https://doi.org/10.1016/J.ESWA.2019.112896).

- [10] Gustavo E. A. P. A. Batista, Eamonn J. Keogh, Oben Moses Tataw, and Vinícius M. A. de Souza. “CID: an efficient complexity-invariant distance for time series”. In: *Data Mining and Knowledge Discovery* 28.3 (2014), pp. 634–669. DOI: [10.1007/S10618-013-0312-3](https://doi.org/10.1007/S10618-013-0312-3).
- [11] André Bauer, Marwin Züfle, Johannes Grohmann, Norbert Schmitt, Nikolas Herbst, and Samuel Kounev. “An Automated Forecasting Framework based on Method Recommendation for Seasonal Time Series”. In: *Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering*. ACM, 2020, pp. 48–55. DOI: [10.1145/3358960.3379123](https://doi.org/10.1145/3358960.3379123).
- [12] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. “TFX: A TensorFlow-Based Production-Scale Machine Learning Platform”. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 1387–1395. DOI: [10.1145/3097983.3098021](https://doi.org/10.1145/3097983.3098021).
- [13] Momotaz Begum and Tadashi Dohi. “A Neuro-Based Software Fault Prediction with Box-Cox Power Transformation”. In: *Journal of Software Engineering and Applications* 10.3 (2017), pp. 288–309. DOI: [10.4236/JSEA.2017.103017](https://doi.org/10.4236/JSEA.2017.103017).
- [14] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML Ain't Markup Language (YAMLTM) Version 1.2*. 3rd Edition. Oct. 2009. URL: <https://yaml.org/spec/1.2/spec.html>.
- [15] Jürgen Bernard, Christian Bors, Markus Bögl, Christian Eichner, Theresia Gschwandtner, Silvia Miksch, Heidrun Schumann, and Jörn Kohlhammer. “Combining the Automated Segmentation and Visual Analysis of Multivariate Time Series”. In: *Proceedings of the EuroVis Workshop on Visual Analytics*. 2018, pp. 49–53. DOI: [10.2312/EUROVA.20181112](https://doi.org/10.2312/EUROVA.20181112).
- [16] Jürgen Bernard, Tobias Ruppert, Oliver Goroll, Thorsten May, and Jörn Kohlhammer. “Visual-Interactive Preprocessing of Time Series Data”. In: *Proceedings of the SIGRAD Interactive Visual Analysis of Data*. Linköping University Electronic Press, 2012, pp. 39–48.
- [17] Robert Birke, Lydia Y. Chen, and Evgenia Smirni. “Data Centers in the Cloud: A Large Scale Performance Study”. In: *Proceedings of the 5th IEEE International Conference on Cloud Computing*. IEEE, 2012, pp. 336–343. DOI: [10.1109/CLOUD.2012.87](https://doi.org/10.1109/CLOUD.2012.87).
- [18] Robert Birke, Lydia Y. Chen, and Evgenia Smirni. “Multi-Resource Characterization and their (In)dependencies in Production Datacenters”. In: *Proceedings of the 14th IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2014, pp. 1–6. DOI: [10.1109/NOMS.2014.6838300](https://doi.org/10.1109/NOMS.2014.6838300).
- [19] Avrim L. Blum and Pat Langley. “Selection of Relevant Features and Examples in Machine Learning”. In: *Artificial intelligence* 97.1-2 (1997), pp. 245–271. DOI: [10.1016/S0004-3702\(97\)00063-5](https://doi.org/10.1016/S0004-3702(97)00063-5).
- [20] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B Woodard, and Hans Andersen. “Fingerprinting the Datacenter: Automated Classification of Performance Crises”. In: *Proceedings of the 5th European Conference on Computer Systems*. 2010, pp. 111–124. DOI: [10.1145/1755913.1755926](https://doi.org/10.1145/1755913.1755926).

- [21] Julio Borges, Martin A. Neumann, Christian Bauer, Yong Ding, Till Riedel, and Michael Beigl. “Predicting Target Events in Industrial Domains”. In: *Proceedings of the 13th International Conference on Machine Learning and Data Mining in Pattern Recognition*. Springer, 2017, pp. 17–31. DOI: [10.1007/978-3-319-62416-7_2](https://doi.org/10.1007/978-3-319-62416-7_2).
- [22] Sabri Boughorbel, Fethi Jarray, and Mohammed El-Anbari. “Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric”. In: *PLOS ONE* 12.6 (June 2017), pp. 1–17. DOI: [10.1371/journal.pone.0177678](https://doi.org/10.1371/journal.pone.0177678).
- [23] Pavel B. Brazdil, Carlos Soares, and Joaquim Pinto Da Costa. “Ranking Learning Algorithms: Using IBL and Meta-Learning on Accuracy and Time Results”. In: *Machine Learning* 50.3 (2003), pp. 251–277. DOI: [10.1023/A:1021713901879](https://doi.org/10.1023/A:1021713901879).
- [24] Leo Breiman. “Random Forests”. In: *Machine Learning* 45.1 (Oct. 2001), pp. 5–32. DOI: [10.1023/a:1010933404324](https://doi.org/10.1023/a:1010933404324).
- [25] E. H. Bristol. “Swinging door trending: Adaptive trend recording?”. In: *ISA National Conference Proceedings, 1990*. 1990, pp. 749–753.
- [26] Mark Brodie, Sheng Ma, Leonid Rachevsky, and Jon Champlin. “Automated Problem Determination Using Call-Stack Matching”. In: *Journal of Network and Systems Management* 13.2 (2005), pp. 219–237. DOI: [10.1007/S10922-005-4443-8](https://doi.org/10.1007/S10922-005-4443-8).
- [27] Field Cady. *The Data Science Handbook*. 1st ed. John Wiley & Sons, Inc., Feb. 2017. ISBN: 978-1-119-09294-0. DOI: [10.1002/9781119092919](https://doi.org/10.1002/9781119092919).
- [28] Tadeusz Caliński and Jerzy Harabasz. “A dendrite method for cluster analysis”. In: *Communications in Statistics* 3.1 (1974), pp. 1–27. DOI: [10.1080/03610927408827101](https://doi.org/10.1080/03610927408827101).
- [29] Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. “Workload Characterization: A Survey Revisited”. In: *ACM Computing Surveys* 48.3 (Feb. 2016), 48:1–43. DOI: [10.1145/2856127](https://doi.org/10.1145/2856127).
- [30] Claudia Canali and Riccardo Lancellotti. “Automated clustering of VMs for scalable cloud monitoring and management”. In: *Proceedings of the 20th International Conference on Software, Telecommunications and Computer Networks*. IEEE, 2012, pp. 1–5.
- [31] Márcio das Chagas Moura, Enrico Zio, Isis Didier Lins, and Enrique Droguett. “Failure and reliability prediction by support vector machines regression of time series data”. In: *Reliability Engineering & System Safety* 96.11 (2011), pp. 1527–1534. DOI: [10.1016/J.RESS.2011.06.006](https://doi.org/10.1016/J.RESS.2011.06.006).
- [32] Robert N. Charette. “Why Software Fails”. In: *IEEE Spectrum* 42.9 (2005), pp. 42–49. DOI: [10.1109/MSPEC.2005.1502528](https://doi.org/10.1109/MSPEC.2005.1502528).
- [33] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. “SMOTE: Synthetic Minority Over-sampling Technique”. In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357. DOI: [10.1613/JAIR.953](https://doi.org/10.1613/JAIR.953).
- [34] Yanpei Chen, Archana Sulochana Ganapathi, Rean Griffith, and Randy H. Katz. *Analysis and Lessons from a Publicly Available Google Cluster Trace*. Tech. rep. UCB/EECS-2010-95. Electrical Engineering and Computer Sciences, University of California at Berkeley, June 2010.
- [35] Maximilian Christ, Nils Braun, Julius Neuffer, and Andreas W. Kempa-Liehr. “Time Series Feature Extraction on basis of Scalable Hypothesis tests (tsfresh – A Python package)”. In: *Neurocomputing* 307 (2018), pp. 72–77. DOI: [10.1016/J.NEUCOM.2018.03.067](https://doi.org/10.1016/J.NEUCOM.2018.03.067).

- [36] Maximilian Christ, Andreas W. Kempa-Liehr, and Michael Feindt. “Distributed and parallel time series feature extraction for industrial big data applications”. In: *CoRR* abs/1610.07717 (2016), pp. 1–36. arXiv: [1610.07717](https://arxiv.org/abs/1610.07717). URL: <http://arxiv.org/abs/1610.07717>.
- [37] Wikimedia Commons. *An example of the correlation of x and y for various distributions of (x,y) pairs*. Accessed: 2020-09-29. May 2011. URL: https://commons.wikimedia.org/wiki/File:Correlation_examples2.svg.
- [38] David Camilo Corrales, Agapito Ledezma, and Juan Carlos Corrales. “From Theory to Practice: A Data Quality Framework for Classification Tasks”. In: *Symmetry* 10.7 (2018), 248:1–29. DOI: [10.3390/SYM10070248](https://doi.org/10.3390/SYM10070248).
- [39] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 153–167. DOI: [10.1145/3132747.3132772](https://doi.org/10.1145/3132747.3132772).
- [40] Douglas Crockford. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. RFC Editor, Dec. 2017, pp. 1–16. URL: <http://www.rfc-editor.org/rfc/rfc8259.txt>.
- [41] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. “RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps”. In: *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering*. ACM, 2016, pp. 820–831. DOI: [10.1145/2884781.2884844](https://doi.org/10.1145/2884781.2884844).
- [42] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. “ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity”. In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 2012, pp. 1084–1093. DOI: [10.1109/ICSE.2012.6227111](https://doi.org/10.1109/ICSE.2012.6227111).
- [43] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. “Desh: Deep Learning for System Health Prediction of Lead Times to Failure in HPC”. In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 2018, pp. 40–51. DOI: [10.1145/3208040.3208051](https://doi.org/10.1145/3208040.3208051).
- [44] Hoang Anh Dau, Anthony Bagnall, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, and Eamonn Keogh. “The UCR Time Series Archive”. In: *IEEE/CAA Journal of Automatica Sinica* 6.6 (2019), pp. 1293–1305. DOI: [10.1109/JAS.2019.1911747](https://doi.org/10.1109/JAS.2019.1911747).
- [45] Hoang Anh Dau, Eamonn Keogh, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, Yanping, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, Gustavo Batista, and Hexagon-ML. *The UCR Time Series Classification Archive*. Accessed: 2020-12-28. Oct. 2018. URL: https://www.cs.ucr.edu/~eamonn/time_series_data_2018/.
- [46] David L. Davies and Donald W. Bouldin. “A Cluster Separation Measure”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1.2 (1979), pp. 224–227. DOI: [10.1109/TPAMI.1979.4766909](https://doi.org/10.1109/TPAMI.1979.4766909).
- [47] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. “UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems”. In: *Proceedings of the 9th International Conference on Autonomic Computing*. 2012, pp. 191–200. DOI: [10.1145/2371536.2371572](https://doi.org/10.1145/2371536.2371572).

- [48] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. “Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox”. In: *Proceedings of the 27th IEEE International Conference on Software Maintenance*. IEEE, 2011, pp. 333–342. DOI: [10.1109/ICSM.2011.6080800](https://doi.org/10.1109/ICSM.2011.6080800).
- [49] Sheng Di, Derrick Kondo, and Franck Cappello. “Characterizing Cloud Applications on a Google Data Center”. In: *Proceedings of the 42nd International Conference on Parallel Processing*. IEEE, 2013, pp. 468–473. DOI: [10.1109/ICPP.2013.56](https://doi.org/10.1109/ICPP.2013.56).
- [50] Sheng Di, Derrick Kondo, and Walfredo Cirne. “Characterization and Comparison of Cloud versus Grid Workloads”. In: *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, 2012, pp. 230–238. DOI: [10.1109/CLUSTER.2012.35](https://doi.org/10.1109/CLUSTER.2012.35).
- [51] David A. Dickey and Wayne A. Fuller. “Distribution of the Estimators for Autoregressive Time Series with a Unit Root”. In: *Journal of the American Statistical Association* 74.366a (1979), pp. 427–431. DOI: [10.1080/01621459.1979.10482531](https://doi.org/10.1080/01621459.1979.10482531).
- [52] Joseph C. Dunn. “A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters”. In: *Journal of Cybernetics* 3.3 (1973), pp. 32–57.
- [53] Jennifer G. Dy and Carla E. Brodley. “Feature Selection for Unsupervised Learning”. In: *Journal of Machine Learning Research* 5 (2004), pp. 845–889.
- [54] Roy Thomas Fielding. “Architectural styles and the design of network-based software architectures”. PhD thesis. University of California, Irvine, 2000.
- [55] Olga Fink, Enrico Zio, and Ulrich Weidmann. “Predicting component reliability and level of degradation with complex-valued neural networks”. In: *Reliability Engineering & System Safety* 121 (2014), pp. 198–206. DOI: [10.1016/J.RESS.2013.08.004](https://doi.org/10.1016/J.RESS.2013.08.004).
- [56] Edward B. Fowlkes and Colin L. Mallows. “A Method for Comparing Two Hierarchical Clusterings”. In: *Journal of the American Statistical Association* 78.383 (1983), pp. 553–569. DOI: [10.1080/01621459.1983.10478008](https://doi.org/10.1080/01621459.1983.10478008).
- [57] Jan Frenzel, Yedhu Sastri, Christoph Lehmann, Taras Lazariv, René Jäkel, and Wolfgang E. Nagel. “A Generalized Service Infrastructure for Data Analytics”. In: *Proceedings of the 4th IEEE International Conference on Big Data Computing Service and Applications*. IEEE, 2018, pp. 25–32. DOI: [10.1109/BIGDATASERVICE.2018.00013](https://doi.org/10.1109/BIGDATASERVICE.2018.00013).
- [58] Ilenia Fronza, Alberto Sillitti, Giancarlo Succi, Mikko Terho, and Jelena Vlasenko. “Failure prediction based on log files using Random Indexing and Support Vector Machines”. In: *Journal of Systems and Software* 86.1 (2013), pp. 2–11. DOI: [10.1016/J.JSS.2012.06.025](https://doi.org/10.1016/J.JSS.2012.06.025).
- [59] Song Fu and Cheng-Zhong Xu. “Exploring Event Correlation for Failure Prediction in Coalitions of Clusters”. In: *Proceedings of the 20th ACM/IEEE Conference on Supercomputing*. ACM, 2007, 41:1–12. DOI: [10.1145/1362622.1362678](https://doi.org/10.1145/1362622.1362678).
- [60] Tak-chung Fu. “A review on time series data mining”. In: *Engineering Applications of Artificial Intelligence* 24.1 (2011), pp. 164–181. DOI: [10.1016/J.ENGAPPAI.2010.09.007](https://doi.org/10.1016/J.ENGAPPAI.2010.09.007).
- [61] Ben D. Fulcher. “Feature-based time-series analysis”. In: *CoRR* abs/1709.08055 (2017), pp. 1–28. arXiv: [1709.08055](https://arxiv.org/abs/1709.08055). URL: <http://arxiv.org/abs/1709.08055>.
- [62] Ben D. Fulcher and Nick S. Jones. “hctsa: A Computational Framework for Automated Time-Series Phenotyping Using Massive Feature Extraction”. In: *Cell Systems* 5.5 (2017), 527–531.e3. DOI: [10.1016/J.CELS.2017.10.001](https://doi.org/10.1016/J.CELS.2017.10.001).

- [63] Ben D. Fulcher and Nick S. Jones. “Highly Comparative Feature-Based Time-Series Classification”. In: *IEEE Transactions on Knowledge and Data Engineering* 26.12 (2014), pp. 3026–3037. DOI: [10.1109/TKDE.2014.2316504](https://doi.org/10.1109/TKDE.2014.2316504).
- [64] Ben D. Fulcher, Max A. Little, and Nick S. Jones. “Highly comparative time-series analysis: the empirical structure of time series and their methods”. In: *Journal of the Royal Society Interface* 10.83 (2013), 20130048:1–12. DOI: [10.1098/RSIF.2013.0048](https://doi.org/10.1098/RSIF.2013.0048).
- [65] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. “Fixing Recurring Crash Bugs via Analyzing Q&A Sites”. In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2015, pp. 307–318. DOI: [10.1109/ASE.2015.81](https://doi.org/10.1109/ASE.2015.81).
- [66] Salvador García, Sergio Ramírez-Gallego, Julián Luengo, José Manuel Benítez, and Francisco Herrera. “Big data preprocessing: methods and prospects”. In: *Big Data Analytics* 1.1 (2016), 9:1–22. DOI: [10.1186/S41044-016-0014-0](https://doi.org/10.1186/S41044-016-0014-0).
- [67] Maryam Abdul Ghafoor and Junaid Haroon Siddiqui. “Cross Platform Bug Correlation using Stack Traces”. In: *Proceedings of the 14th International Conference on Frontiers of Information Technology*. IEEE, 2016, pp. 199–204. DOI: [10.1109/FIT.2016.044](https://doi.org/10.1109/FIT.2016.044).
- [68] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. “Debugging in the (Very) Large: Ten Years of Implementation and Experience”. In: *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*. 2009, pp. 103–116. DOI: [10.1145/1629575.1629586](https://doi.org/10.1145/1629575.1629586).
- [69] Anjali Goyal and Neetu Sardana. “Machine Learning or Information Retrieval Techniques for Bug Triaging: Which is Better?” In: *e-Informatica Software Engineering Journal* 11.1 (2017). DOI: [10.5277/E-INF170106](https://doi.org/10.5277/E-INF170106).
- [70] Alibaba Group. *Alibaba Cluster Trace Program*. Accessed: 2021-01-25. 2017. URL: <https://github.com/alibaba/clusterdata>.
- [71] Theresia Gschwandtner, Wolfgang Aigner, Silvia Miksch, Johannes Gärtner, Simone Kriglstein, Margit Pohl, and Nik Suchy. “TimeCleanser: A Visual Analytics Approach for Data Cleansing of Time-Oriented Data”. In: *Proceedings of the 14th International Conference on Knowledge Technologies and Data-Driven Business*. 2014, pp. 1–8. DOI: [10.1145/2637748.2638423](https://doi.org/10.1145/2637748.2638423).
- [72] Manish Gupta, Abhishek B. Sharma, Haifeng Chen, and Guofei Jiang. “Context-Aware Time Series Anomaly Detection for Complex Systems”. In: *Proceedings of the 2nd Workshop on Data Mining for Service and Maintenance*. 2013, pp. 14–22.
- [73] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. “Array programming with NumPy”. In: *Nature* 585 (2020), pp. 357–362. DOI: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2).
- [74] Haibo He, Yang Bai, Eduardo A Garcia, and Shutao Li. “ADASYN: Adaptive Synthetic Sampling Approach For Imbalanced Learning”. In: *Proceedings of the IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. IEEE, 2008, pp. 1322–1328. DOI: [10.1109/IJCNN.2008.4633969](https://doi.org/10.1109/IJCNN.2008.4633969).

- [75] Xin He, Kaiyong Zhao, and Xiaowen Chu. “AutoML: A survey of the state-of-the-art”. In: *Knowledge-Based Systems* 212 (2021), 106622:1–27. DOI: [10.1016/J.KNOSYS.2020.106622](https://doi.org/10.1016/J.KNOSYS.2020.106622).
- [76] Sepp Hochreiter. *Theoretical Concepts of Machine Learning*. Lecture notes. Johannes Kepler University Linz, Austria, 2014. URL: http://www.bioinf.at/teaching/ss2018/ss_vl_tcml/ML_theoretical.pdf.
- [77] Seyedrebar Hosseini, Burak Turhan, and Dimuthu Gunarathna. “A Systematic Literature Review and Meta-analysis on Cross Project Defect Prediction”. In: *IEEE Transactions on Software Engineering* 45.2 (2017), pp. 111–147. DOI: [10.1109/TSE.2017.2770124](https://doi.org/10.1109/TSE.2017.2770124).
- [78] Lawrence Hubert and Phipps Arabie. “Comparing Partitions”. In: *Journal of Classification* 2.1 (1985), pp. 193–218. DOI: [10.1007/bf01908075](https://doi.org/10.1007/bf01908075).
- [79] H. E. Hurst. “Long-Term Storage Capacity of Reservoirs”. In: *Transactions of the American Society of Civil Engineers* 116.1 (1951), pp. 770–799.
- [80] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. “Algorithm runtime prediction: Methods & evaluation”. In: *Artificial Intelligence* 206 (2014), pp. 79–111. DOI: [10.1016/J.ARTINT.2013.10.003](https://doi.org/10.1016/J.ARTINT.2013.10.003).
- [81] Rob J. Hyndman and George Athanasopoulos. “ARIMA models”. In: *Forecasting: Principles and Practice*. OTexts, 2018, pp. 221–274. URL: <https://otexts.com/fpp2/>.
- [82] Rob J. Hyndman, Earo Wang, and Nikolay Laptev. “Large-Scale Unusual Time Series Detection”. In: *Proceedings of the 15th IEEE International Conference on Data Mining Workshop*. IEEE, 2015, pp. 1616–1619. DOI: [10.1109/ICDMW.2015.104](https://doi.org/10.1109/ICDMW.2015.104).
- [83] Google Inc. *Borg cluster traces from Google*. Accessed: 2021-01-25. 2011. URL: <https://github.com/google/cluster-data>.
- [84] InfluxData. *InfluxDB: Purpose-Built Open Source Time Series Database*. Accessed: 2020-10-01. 2020. URL: <https://www.influxdata.com/>.
- [85] Arunima Jaiswal and Ruchika Malhotra. “Software reliability prediction using machine learning techniques”. In: *International Journal of System Assurance Engineering and Management* 9.1 (2018), pp. 230–244. DOI: [10.1007/S13198-016-0543-Y](https://doi.org/10.1007/S13198-016-0543-Y).
- [86] Femke Jansen, Mike Holenderski, Tanir Ozcelebi, Paulien Dam, and Bas Tjisma. “Predicting machine failures from industrial time series data”. In: *Proceedings of the 5th International Conference on Control, Decision and Information Technologies*. IEEE, 2018, pp. 1091–1096. DOI: [10.1109/CODIT.2018.8394915](https://doi.org/10.1109/CODIT.2018.8394915).
- [87] Pablo A. Jaskowiak, Ricardo J. G. B. Campello, and Ivan G. Costa. “On the selection of appropriate distances for gene expression data clustering”. In: *BMC Bioinformatics* 15.2 (2014), S2:1–17. DOI: [10.1186/1471-2105-15-S2-S2](https://doi.org/10.1186/1471-2105-15-S2-S2).
- [88] Congfeng Jiang, Guangjie Han, Jiangbin Lin, Gangyong Jia, Weisong Shi, and Jian Wan. “Characteristics of Co-Allocated Online Services and Batch Jobs in Internet Data Centers: A Case Study From Alibaba Cloud”. In: *IEEE Access* 7 (2019), pp. 22495–22508. DOI: [10.1109/ACCESS.2019.2897898](https://doi.org/10.1109/ACCESS.2019.2897898).
- [89] Joe H. Ward Jr. “Hierarchical Grouping to Optimize an Objective Function”. In: *Journal of the American Statistical Association* 58.301 (1963), pp. 236–244. DOI: [10.1080/01621459.1963.10500845](https://doi.org/10.1080/01621459.1963.10500845).
- [90] Mario Kahlhofer. “Exploring Supervised Event Prediction in Multi-System Monitoring”. MA thesis. Johannes Kepler University Linz, Austria, Aug. 2019. URL: <https://epub.jku.at/obvulihs/content/titleinfo/4403446>.

- [91] Fazle Karim, Somshubra Majumdar, Houshang Darabi, and Shun Chen. “LSTM Fully Convolutional Networks for Time Series Classification”. In: *IEEE access* 6 (2017), pp. 1662–1669. DOI: [10.1109/ACCESS.2017.2779939](https://doi.org/10.1109/ACCESS.2017.2779939).
- [92] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. “Workload Characterization and Prediction in the Cloud: A Multiple Time Series Approach”. In: *Proceedings of the 13th IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2012, pp. 1287–1294. DOI: [10.1109/NOMS.2012.6212065](https://doi.org/10.1109/NOMS.2012.6212065).
- [93] Dongsun Kim, Xinming Wang, Sunghun Kim, Andreas Zeller, Shing-Chi Cheung, and Sooyong Park. “Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts”. In: *IEEE Transactions on Software Engineering* 37.3 (2011), pp. 430–447. DOI: [10.1109/TSE.2011.20](https://doi.org/10.1109/TSE.2011.20).
- [94] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. “Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage”. In: *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2011, pp. 486–493. DOI: [10.1109/DSN.2011.5958261](https://doi.org/10.1109/DSN.2011.5958261).
- [95] S. B. Kotsiantis, Dimitris Kanellopoulos, and P. E. Pintelas. “Data Preprocessing for Supervised Learning”. In: *International Journal of Computer Science* 1.1 (2006), pp. 111–117.
- [96] S. Kullback and R. A. Leibler. “On Information and Sufficiency”. In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86. DOI: [10.1214/AOMS/1177729694](https://doi.org/10.1214/AOMS/1177729694).
- [97] Denis Kwiatkowski, Peter C. B. Phillips, Peter Schmidt, and Yongcheol Shin. “Testing the null hypothesis of stationarity against the alternative of a unit root: How sure are we that economic time series have a unit root?” In: *Journal of Econometrics* 54.1–3 (1992), pp. 159–178. DOI: [10.1016/0304-4076\(92\)90104-Y](https://doi.org/10.1016/0304-4076(92)90104-Y).
- [98] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: A LLVM-Based Python JIT Compiler”. In: *Proceedings of the 2nd Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 2015, 7:1–6. DOI: [10.1145/2833157.2833162](https://doi.org/10.1145/2833157.2833162).
- [99] Zhiling Lan, Ziming Zheng, and Yawei Li. “Toward Automated Anomaly Identification in Large-Scale Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 21.2 (2009), pp. 174–187. DOI: [10.1109/TPDS.2009.52](https://doi.org/10.1109/TPDS.2009.52).
- [100] Martin Långkvist, Lars Karlsson, and Amy Loutfi. “A review of unsupervised feature learning and deep learning for time-series modeling”. In: *Pattern Recognition Letters* 42 (2014), pp. 11–24. DOI: [10.1016/J.PATREC.2014.01.008](https://doi.org/10.1016/J.PATREC.2014.01.008).
- [101] Nikolay Laptev, Saeed Amizadeh, and Ian Flint. “Generic and Scalable Framework for Automated Time-series Anomaly Detection”. In: *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 1939–1947. DOI: [10.1145/2783258.2788611](https://doi.org/10.1145/2783258.2788611).
- [102] Guillaume Lemaître, Fernando Nogueira, Dayvid V. Oliveira, and Christos K. Aridas. *User Guide of imbalanced-learn: Under-sampling*. Accessed: 2021-01-27. 2020. URL: https://imbalanced-learn.org/stable/under_sampling.html.
- [103] Christiane Lemke and Bogdan Gabrys. “Meta-learning for time series forecasting and forecast combination”. In: *Neurocomputing* 73.10 (2010), pp. 2006–2016. DOI: [10.1016/J.NEUCOM.2009.09.020](https://doi.org/10.1016/J.NEUCOM.2009.09.020).
- [104] Vladimir I. Levenshtein. “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet physics. Doklady* 10.8 (1966), pp. 707–710.
- [105] Dynatrace LLC. *The Leader in Cloud Monitoring*. Accessed: 2021-01-29. 2021. URL: <https://www.dynatrace.com/>.

- [106] Jungang Lou, Yunliang Jiang, Qing Shen, Zhangguo Shen, Zhen Wang, and Ruiqin Wang. “Software reliability prediction via relevance vector regression”. In: *Neurocomputing* 186 (2016), pp. 66–73. DOI: [10.1016/J.NEUCOM.2015.12.077](https://doi.org/10.1016/J.NEUCOM.2015.12.077).
- [107] Jungang Lou, Yunliang Jiang, Qing Shen, and Ruiqin Wang. “Failure prediction by relevance vector regression with improved quantum-inspired gravitational search”. In: *Journal of Network and Computer Applications* 103 (2018), pp. 171–177. DOI: [10.1016/J.JNCA.2017.11.013](https://doi.org/10.1016/J.JNCA.2017.11.013).
- [108] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. “Understanding variable importances in forests of randomized trees”. In: *Advances in Neural Information Processing Systems*. Vol. 26. 2013, pp. 431–439.
- [109] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. “Imbalance in the cloud: An analysis on Alibaba cluster trace”. In: *Proceedings of the 5th IEEE International Conference on Big Data*. IEEE, 2017, pp. 2884–2892. DOI: [10.1109/BIGDATA.2017.8258257](https://doi.org/10.1109/BIGDATA.2017.8258257).
- [110] Carl Henning Lubba, Sarab S. Sethi, Philip Knaute, Simon R. Schultz, Ben D. Fulcher, and Nick S. Jones. “catch22: CAnonical Time-series CHaracteristics”. In: *Data Mining and Knowledge Discovery* 33.6 (2019), pp. 1821–1852. DOI: [10.1007/S10618-019-00647-X](https://doi.org/10.1007/S10618-019-00647-X).
- [111] Laurens van der Maaten and Geoffrey Hinton. “Visualizing Data using t-SNE”. In: *Journal of Machine Learning Research* 9 (2008), pp. 2579–2605.
- [112] James MacQueen. “Some Methods for Classification and Analysis of Multivariate Observations”. In: *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*. Vol. 1. 1967, pp. 281–297.
- [113] Igor Manojlović, Goran Švenda, Aleksandar Erdeljan, and Milan Gavrić. “Time series grouping algorithm for load pattern recognition”. In: *Computers in Industry* 111 (2019), pp. 140–147. DOI: [10.1016/J.COMPIND.2019.07.009](https://doi.org/10.1016/J.COMPIND.2019.07.009).
- [114] Marin Matijaš, Johan A. K. Suykens, and Slavko Krajcar. “Load forecasting using a multivariate meta-learning system”. In: *Expert Systems with Applications* 40.11 (2013), pp. 4427–4437. DOI: [10.1016/J.ESWA.2013.01.047](https://doi.org/10.1016/J.ESWA.2013.01.047).
- [115] Brian W. Matthews. “Comparison of the predicted and observed secondary structure of T4 phage lysozyme”. In: *Biochimica et Biophysica Acta (BBA) - Protein Structure* 405.2 (1975), pp. 442–451. ISSN: 0005-2795. DOI: [https://doi.org/10.1016/0005-2795\(75\)90109-9](https://doi.org/10.1016/0005-2795(75)90109-9).
- [116] Robert McGill, John W. Tukey, and Wayne A. Larsen. “Variations of Box Plots”. In: *The American Statistician* 32.1 (1978), pp. 12–16.
- [117] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. 2010, pp. 56–61. DOI: [10.25080/MAJORA-92BF1922-00A](https://doi.org/10.25080/MAJORA-92BF1922-00A).
- [118] Bart van Merriënboer, Dzmitry Bahdanau, Vincent Dumoulin, Dmitriy Serdyuk, David Warde-Farley, Jan Chorowski, and Yoshua Bengio. “Blocks and Fuel: Frameworks for deep learning”. In: *CoRR* abs/1506.00619 (2015), pp. 1–5. arXiv: [1506.00619](https://arxiv.org/abs/1506.00619). URL: <http://arxiv.org/abs/1506.00619>.
- [119] Microsoft. *Application Domains for Report Server Applications*. Accessed: 2020-10-17. Mar. 2017. URL: <https://docs.microsoft.com/en-us/sql/reporting-services/report-server/application-domains-for-report-server-applications>.

- [120] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. “YALE: Rapid Prototyping for Complex Data Mining Tasks”. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2006, pp. 935–940. DOI: [10.1145/1150402.1150531](https://doi.org/10.1145/1150402.1150531).
- [121] Natwar Modani, Rajeev Gupta, Guy Lohman, Tanveer Syeda-Mahmood, and Laurent Mignet. “Automatically Identifying Known Software Problems”. In: *Proceedings of the 23rd IEEE International Conference on Data Engineering Workshop*. IEEE, 2007, pp. 433–441. DOI: [10.1109/ICDEW.2007.4401026](https://doi.org/10.1109/ICDEW.2007.4401026).
- [122] Martin Monperrus. “Automatic Software Repair: a Bibliography”. In: *ACM Computing Surveys (CSUR)* 51.1 (2018), pp. 1–24. DOI: [10.1145/3105906](https://doi.org/10.1145/3105906).
- [123] Usue Mori, Alexander Mendiburu, and Jose A. Lozano. “Similarity Measure Selection for Clustering Time Series Databases”. In: *IEEE Transactions on Knowledge and Data Engineering* 28.1 (2016), pp. 181–195. DOI: [10.1109/TKDE.2015.2462369](https://doi.org/10.1109/TKDE.2015.2462369).
- [124] Syed Shariyar Murtaza, Nazim H. Madhavji, Mechelle Gittens, and Abdelwahab Hamou-Lhadj. “Identifying Recurring Faulty Functions in Field Traces of a Large Industrial Software System”. In: *IEEE Transactions on Reliability* 64.1 (2014), pp. 269–283. DOI: [10.1109/TR.2014.2366274](https://doi.org/10.1109/TR.2014.2366274).
- [125] Alex Nanopoulos, Rob Alcock, and Yannis Manolopoulos. “Feature-based Classification of Time-series Data”. In: *International Journal of Computer Research* 10.3 (2001), pp. 49–61.
- [126] Frank Nielsen. “Hierarchical clustering”. In: *Introduction to HPC with MPI for Data Science*. Springer, 2016, pp. 221–239. DOI: [10.1007/978-3-319-21903-5](https://doi.org/10.1007/978-3-319-21903-5).
- [127] Edward E. Ogheneovo. “Software Dysfunction: Why Do Software Fail?” In: *Journal of Computer and Communications* 2.6 (2014), pp. 25–35. DOI: [10.4236/JCC.2014.26004](https://doi.org/10.4236/JCC.2014.26004).
- [128] Burcu Ozcelik and Cemal Yilmaz. “Seer: A Lightweight Online Failure Prediction Approach”. In: *IEEE Transactions on Software Engineering* 42.1 (2016), pp. 26–46. DOI: [10.1109/TSE.2015.2442577](https://doi.org/10.1109/TSE.2015.2442577).
- [129] John Paparrizos and Luis Gravano. “K-Shape: Efficient and Accurate Clustering of Time Series”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1855–1870. DOI: [10.1145/2723372.2737793](https://doi.org/10.1145/2723372.2737793).
- [130] Jinhee Park, Nakwon Lee, and Jongmoon Baik. “On the Long-term Predictive Capability of Data-driven Software Reliability Model: An Empirical Evaluation”. In: *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 45–54. DOI: [10.1109/ISSRE.2014.28](https://doi.org/10.1109/ISSRE.2014.28).
- [131] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.
- [132] Nancy Pérez-Castro, Aldo Márquez-Grajales, Héctor Gabriel Acosta-Mesa, and Efrén Mezura-Montes. “Full Model Selection issue in Temporal Data through Evolutionary Algorithms: A Brief Review”. In: *Proceedings of the IEEE Congress on Evolutionary Computation*. IEEE, 2017, pp. 2451–2457. DOI: [10.1109/CEC.2017.7969602](https://doi.org/10.1109/CEC.2017.7969602).
- [133] Bruno Almeida Pimentel and André C. P. L. F. de Carvalho. “Statistical versus Distance-Based Meta-Features for Clustering Algorithm recommendation Using Meta-Learning”. In: *Proceedings of the International Joint Conference on Neural Networks*. IEEE, 2018, pp. 1–8. DOI: [10.1109/IJCNN.2018.8489182](https://doi.org/10.1109/IJCNN.2018.8489182).

- [134] Teerat Pitakrat, Jonas Grunert, Oliver Kabierschke, Fabian Keller, and André Van Hoorn. “A Framework for System Event Classification and Prediction by Means of Machine Learning”. In: *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*. 2014, pp. 173–180. DOI: [10.4108/ICST.VALUETOOLS.2014.258197](https://doi.org/10.4108/ICST.VALUETOOLS.2014.258197).
- [135] Teerat Pitakrat, Dušan Okanović, André van Hoorn, and Lars Grunske. “Hora: Architecture-aware online failure prediction”. In: *Journal of Systems and Software* 137 (2018), pp. 669–685. DOI: [10.1016/J.JSS.2017.02.041](https://doi.org/10.1016/J.JSS.2017.02.041).
- [136] Faimison Porto, Leandro Minku, Emilia Mendes, and Adenilso Simao. “A Systematic Study of Cross-Project Defect Prediction With Meta-Learning”. In: *CoRR* abs/1802.06025 (2018), pp. 1–28. arXiv: [1802.06025](https://arxiv.org/abs/1802.06025). URL: <http://arxiv.org/abs/1802.06025>.
- [137] John W. Pratt. “Remarks on Zeros and Ties in the Wilcoxon Signed Rank Procedures”. In: *Journal of the American Statistical Association* 54.287 (1959), pp. 655–667. DOI: [10.1080/01621459.1959.10501526](https://doi.org/10.1080/01621459.1959.10501526).
- [138] Fangyun Qin, Zheng Zheng, Chenggang Bai, Yu Qiao, Zhenyu Zhang, and Cheng Chen. “Cross-Project Aging Related Bug Prediction”. In: *Proceedings of the 15th IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 43–48. DOI: [10.1109/QRS.2015.17](https://doi.org/10.1109/QRS.2015.17).
- [139] Alfredo Ramirez and Charles Cox. “Improving on the Range Rule of Thumb”. In: *Rose-Hulman Undergraduate Mathematics Journal* 13.2 (2012), pp. 1–13.
- [140] Jeff Reback, Wes McKinney, jbrockmendel, Joris Van den Bossche, Tom Augspurger, Phillip Cloud, gyoung, Sinhrks, Adam Klein, Matthew Roeschke, Simon Hawkins, Jeff Tratner, Chang She, William Ayd, Terji Petersen, Marc Garcia, Jeremy Schendel, Andy Hayden, MomIsBestFriend, Vytautas Jancauskas, Pietro Battiston, Skipper Seabold, chris-b1, h-vetinari, Stephan Hoyer, Wouter Overmeire, alimcmaster1, Kaiqi Dong, Christopher Whelan, and Mortada Mehyar. *pandas-dev/pandas: Pandas 1.0.3*. Version 1.0.3. Mar. 2020. DOI: [10.5281/ZENODO.3715232](https://doi.org/10.5281/ZENODO.3715232).
- [141] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. *Google cluster-usage traces: format + schema*. Tech. rep. Google Inc., Oct. 2011.
- [142] Hansheng Ren, Bixiong Xu, Yujing Wang, Chao Yi, Congrui Huang, Xiaoyu Kou, Tony Xing, Mao Yang, Jie Tong, and Qi Zhang. “Time-Series Anomaly Detection Service at Microsoft”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2019, pp. 3009–3017. DOI: [10.1145/3292500.3330680](https://doi.org/10.1145/3292500.3330680).
- [143] John R. Rice. “The Algorithm Selection Problem”. In: *Advances in Computers*. Vol. 15. Elsevier, 1976, pp. 65–118. DOI: [10.1016/S0065-2458\(08\)60520-3](https://doi.org/10.1016/S0065-2458(08)60520-3).
- [144] Joshua S. Richman and J. Randall Moorman. “Physiological time-series analysis using approximate entropy and sample entropy”. In: *American Journal of Physiology-Heart and Circulatory Physiology* 278.6 (2000), H2039–H2049. DOI: [10.1152/AJPHEART.2000.278.6.H2039](https://doi.org/10.1152/AJPHEART.2000.278.6.H2039).
- [145] Lior Rokach and Oded Maimon. “Clustering Methods”. In: *Data Mining and Knowledge Discovery Handbook*. Ed. by Oded Maimon and Lior Rokach. Springer US, 2005, pp. 321–352. ISBN: 978-0-387-25465-4. DOI: [10.1007/0-387-25465-X_15](https://doi.org/10.1007/0-387-25465-X_15).

- [146] Andrew Rosenberg and Julia Hirschberg. “V-Measure: A conditional entropy-based external cluster evaluation measure”. In: *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. 2007, pp. 410–420.
- [147] Peter J. Rousseeuw. “Silhouettes: a graphical aid to the interpretation and validation of cluster analysis”. In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65. DOI: [10.1016/0377-0427\(87\)90125-7](https://doi.org/10.1016/0377-0427(87)90125-7).
- [148] Magnus Sahlgren. “An Introduction to Random Indexing”. In: *Proceedings of the Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Terminology and Knowledge Engineering*. 2005, pp. 1–9.
- [149] Felix Salfner, Maren Lenk, and Miroslaw Malek. “A Survey of Online Failure Prediction Methods”. In: *ACM Computing Surveys (CSUR)* 42.3 (2010), pp. 1–42. DOI: [10.1145/1670679.1670680](https://doi.org/10.1145/1670679.1670680).
- [150] Felix Salfner and Steffen Tschirpke. “Error Log Processing for Accurate Failure Prediction”. In: *Proceedings of the 1st USENIX Workshop on the Analysis of System Logs*. 2008.
- [151] Christopher Schölzel. *Nonlinear measures for dynamical systems*. Version 0.5.2. June 2019. DOI: [10.5281/ZENODO.3814723](https://doi.org/10.5281/ZENODO.3814723).
- [152] Andreas Schörghener, Paul Grünbacher, and Hanspeter Mössenböck. “Selecting Time Series Clustering Methods based on Run-Time Costs”. In: *Proceedings of the 11th Symposium on Software Performance*. Accepted for publication. GI Softwaretechnik-Trends, 2020. URL: https://www.performance-symposium.org/fileadmin/user_upload/palladio-conference/2020/Papers/SSP2020_paper_1.pdf.
- [153] Andreas Schörghener, Mario Kahlhofer, Peter Chalupar, Paul Grünbacher, and Hanspeter Mössenböck. “A Framework for Preprocessing Multivariate, Topology-Aware Time Series and Event Data in a Multi-System Environment”. In: *Proceedings of the 19th IEEE International Symposium on High Assurance Systems Engineering*. IEEE, 2019, pp. 115–122. DOI: [10.1109/HASE.2019.00026](https://doi.org/10.1109/HASE.2019.00026).
- [154] Andreas Schörghener, Mario Kahlhofer, Peter Chalupar, Hanspeter Mössenböck, and Paul Grünbacher. “On the Difficulties of Supervised Event Prediction based on Unbalanced Real-World Data in Multi-System Monitoring”. In: *Proceedings of the 10th Symposium on Software Performance*. GI Softwaretechnik-Trends, 2019, pp. 38–40. URL: http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Schoergener.pdf.
- [155] Andreas Schörghener, Mario Kahlhofer, Peter Chalupar, Hanspeter Mössenböck, and Paul Grünbacher. “Using Multi-System Monitoring Time Series to Predict Performance Events”. In: *Proceedings of the 9th Symposium on Software Performance*. GI Softwaretechnik-Trends, 2018, pp. 55–57. URL: http://pi.informatik.uni-siegen.de/stt/39_3/01_Fachgruppenberichte/SSP18/SchoergenerKahlhoferChalupar+18.pdf.
- [156] Andreas Schörghener, Mario Kahlhofer, Paul Grünbacher, and Hanspeter Mössenböck. “Can We Predict Performance Events with Time Series Data from Monitoring Multiple Systems?” In: *Companion of the 10th ACM/SPEC International Conference on Performance Engineering*. ACM, 2019, pp. 9–12. DOI: [10.1145/3302541.3313101](https://doi.org/10.1145/3302541.3313101).

- [157] Andreas Schörghener, Mario Kahlhofer, Hanspeter Mössenböck, and Paul Grünbacher. “Using Crash Frequency Analysis to Identify Error-Prone Software Technologies in Multi-System Monitoring”. In: *Proceedings of the 18th IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2018, pp. 183–190. DOI: [10.1109/QRS.2018.00032](https://doi.org/10.1109/QRS.2018.00032).
- [158] Andreas Schörghener, Thomas Natschläger, Paul Grünbacher, Mario Kahlhofer, Peter Chalupar, and Hanspeter Mössenböck. “An Approach for Ranking Feature-based Clustering Methods and its Application in Multi-System Infrastructure Monitoring”. In: *AI-Enabled Software Development and Operations (AI4DevOps) of the 47th EuroMicro Conference on Software Engineering and Advanced Applications*. Accepted for publication. IEEE, 2021.
- [159] Thomas Schreiber and Andreas Schmitz. “Discrimination power of measures for nonlinearity in a time series”. In: *Physical Review E* 55 (5 May 1997), pp. 5443–5447. DOI: [10.1103/PHYSREVE.55.5443](https://doi.org/10.1103/PHYSREVE.55.5443).
- [160] Thomas Schreiber and Andreas Schmitz. “Surrogate time series”. In: *Physica D: Nonlinear Phenomena* 142.3 (2000), pp. 346–382. DOI: [10.1016/S0167-2789\(00\)00043-9](https://doi.org/10.1016/S0167-2789(00)00043-9).
- [161] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. “Do Stack Traces Help Developers Fix Bugs?” In: *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*. IEEE, 2010, pp. 118–121. DOI: [10.1109/MSR.2010.5463280](https://doi.org/10.1109/MSR.2010.5463280).
- [162] Arthur Schuster. “On the investigation of hidden periodicities with application to a supposed 26 day period of meteorological phenomena”. In: *Terrestrial Magnetism* 3.1 (1898), pp. 13–41. DOI: [10.1029/TM003I001P00013](https://doi.org/10.1029/TM003I001P00013).
- [163] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180. RFC Editor, Oct. 2005, pp. 1–8. URL: <http://www.rfc-editor.org/rfc/rfc4180.txt>.
- [164] Bikash Sharma, Praveen Jayachandran, Akshat Verma, and Chita R. Das. “CloudPD: Problem Determination and Diagnosis in Shared Dynamic Clouds”. In: *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2013, pp. 1–12. DOI: [10.1109/DSN.2013.6575298](https://doi.org/10.1109/DSN.2013.6575298).
- [165] Virendra Singh Shekhawat, Avinash Gautam, and Ashish Thakrar. “Datacenter Workload Classification and Characterization: An Empirical Approach”. In: *Proceedings of the 13th IEEE International Conference on Industrial and Information Systems*. IEEE, 2018, pp. 1–7. DOI: [10.1109/ICIINFS.2018.8721402](https://doi.org/10.1109/ICIINFS.2018.8721402).
- [166] Siqi Shen, Vincent Van Beek, and Alexandru Iosup. “Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters”. In: *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2015, pp. 465–474. DOI: [10.1109/CCGRID.2015.60](https://doi.org/10.1109/CCGRID.2015.60).
- [167] Kevin Sheppard. *ARCH Toolbox for Python*. Feb. 2015. DOI: [10.5281/ZENODO.15681](https://doi.org/10.5281/ZENODO.15681).
- [168] Weiwei Shi, Yongxin Zhu, Tian Huang, Gehao Sheng, Yong Lian, Guoxing Wang, and Yufeng Chen. “An Integrated Data Preprocessing Framework Based on Apache Spark for Fault Diagnosis of Power Grid Equipment”. In: *Journal of Signal Processing Systems* 86.2 (2017), pp. 221–236. DOI: [10.1007/S11265-016-1119-4](https://doi.org/10.1007/S11265-016-1119-4).
- [169] Connor Shorten and Taghi M. Khoshgoftaar. “A survey on Image Data Augmentation for Deep Learning”. In: *Journal of Big Data* 6.1 (2019), pp. 1–48. DOI: [10.1186/S40537-019-0197-0](https://doi.org/10.1186/S40537-019-0197-0).
- [170] Michael Sipser. “Time Complexity”. In: *Introduction to the Theory of Computation*. Cengage Learning, 2012, pp. 275–330. ISBN: 978-1-133-18779-0.

- [171] Steven S. Skiena. *The Data Science Design Manual*. 1st ed. Springer, Aug. 2017. ISBN: 978-3-319-55443-3. DOI: [10.1007/978-3-319-55444-0](https://doi.org/10.1007/978-3-319-55444-0).
- [172] Robert R. Sokal and Charles D. Michener. “A Statistical Method for Evaluating Systematic Relationships”. In: *University of Kansas Science Bulletin* 38.6 (1958), pp. 1409–1438.
- [173] Saúl Solorio-Fernández, J. Ariel Carrasco-Ochoa, and José Fco. Martínez-Trinidad. “A review of unsupervised feature selection methods”. In: *Artificial Intelligence Review* 53.2 (2020), pp. 907–948. DOI: [10.1007/S10462-019-09682-Y](https://doi.org/10.1007/S10462-019-09682-Y).
- [174] Christopher Streiffer, Ramya Raghavendra, Theophilus Benson, and Mudhakar Srivatsa. “Learning to Simplify Distributed Systems Management”. In: *Proceedings of the 6th IEEE International Conference on Big Data*. IEEE, 2018, pp. 1837–1845. DOI: [10.1109/BIGDATA.2018.8622058](https://doi.org/10.1109/BIGDATA.2018.8622058).
- [175] Yongmin Tan, Xiaohui Gu, and Haixun Wang. “Adaptive System Anomaly Prediction for Large-Scale Hosting Infrastructures”. In: *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. 2010, pp. 173–182. DOI: [10.1145/1835698.1835741](https://doi.org/10.1145/1835698.1835741).
- [176] The Joblib development team. *Joblib: running Python functions as pipeline jobs*. Version 0.14.1. 2019. URL: <https://joblib.readthedocs.io/>.
- [177] The Scikit-learn development team. *sklearn.tree.DecisionTreeClassifier*. Version 0.24.1. Accessed: 2021-03-03. 2020. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>.
- [178] Timo Teräsvirta, Chien-Fu Lin, and Clive W. J. Granger. “Power of the neural network linearity test”. In: *Journal of Time Series Analysis* 14.2 (1993), pp. 209–220. DOI: [10.1111/J.1467-9892.1993.TB00139.X](https://doi.org/10.1111/J.1467-9892.1993.TB00139.X).
- [179] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. “Sieve: Actionable Insights from Monitored Metrics in Distributed Systems”. In: *Proceedings of the 18th ACM/I-FIP/USENIX Middleware Conference*. ACM, 2017, pp. 14–27. DOI: [10.1145/3135974.3135977](https://doi.org/10.1145/3135974.3135977).
- [180] Ivan Tomek. “Two Modifications of CNN”. In: *IEEE Transactions on Systems, Man and Cybernetics* 6.11 (1976), pp. 769–772. DOI: [10.1109/TSMC.1976.4309452](https://doi.org/10.1109/TSMC.1976.4309452).
- [181] Jamal Uddin, Rozaida Ghazali, Mustafa Mat Deris, Rashid Naseem, and Habib Shah. “A survey on bug prioritization”. In: *Artificial Intelligence Review* 47.2 (2017), pp. 145–180. DOI: [10.1007/S10462-016-9478-6](https://doi.org/10.1007/S10462-016-9478-6).
- [182] Terry T. Um, Franz M. J. Pfister, Daniel Pichler, Satoshi Endo, Muriel Lang, Sandra Hirche, Urban Fietzek, and Dana Kulić. “Data Augmentation of Wearable Sensor Data for Parkinson’s Disease Monitoring Using Convolutional Neural Networks”. In: *Proceedings of the 19th ACM International Conference on Multimodal Interaction*. 2017, pp. 216–220. DOI: [10.1145/3136755.3136817](https://doi.org/10.1145/3136755.3136817).
- [183] Joaquin Vanschoren. “Meta-Learning”. In: *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Springer, 2019, pp. 35–61. ISBN: 978-3-030-05318-5. DOI: [10.1007/978-3-030-05318-5_2](https://doi.org/10.1007/978-3-030-05318-5_2).
- [184] John Venn. “I. On the diagrammatic and mechanical representation of propositions and reasonings”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 10.59 (1880), pp. 1–18. DOI: [10.1080/14786448008626877](https://doi.org/10.1080/14786448008626877).

- [185] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: [10.1038/s41592-019-0686-2](https://doi.org/10.1038/s41592-019-0686-2).
- [186] Shaohua Wang, Foutse Khomh, and Ying Zou. “Improving bug management using correlations in crash reports”. In: *Empirical Software Engineering* 21.2 (2016), pp. 337–367. DOI: [10.1007/S10664-014-9333-9](https://doi.org/10.1007/S10664-014-9333-9).
- [187] Xiaozhe Wang, Kate Smith, and Rob Hyndman. “Characteristic-Based Clustering for Time Series Data”. In: *Data Mining and Knowledge Discovery* 13.3 (2006), pp. 335–364. DOI: [10.1007/S10618-005-0039-X](https://doi.org/10.1007/S10618-005-0039-X).
- [188] Martin Wattenberg, Fernanda Viégas, and Ian Johnson. “How to Use t-SNE Effectively”. In: *Distill* (2016). DOI: [10.23915/distill.00002](https://doi.org/10.23915/distill.00002). URL: <http://distill.pub/2016/misread-tsne>.
- [189] Frank Wilcoxon. “Individual Comparisons by Ranking Methods”. In: *Biometrics Bulletin* 1.6 (1945), pp. 80–83. DOI: [10.2307/3001968](https://doi.org/10.2307/3001968).
- [190] Andrew W. Williams, Soila M. Pertet, and Priya Narasimhan. “Tiresias: Black-Box Failure Prediction in Distributed Systems”. In: *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8. DOI: [10.1109/IPDPS.2007.370345](https://doi.org/10.1109/IPDPS.2007.370345).
- [191] John R. Williams. “Clustering Household Electricity Use Profiles”. In: *Proceedings of the 1st Workshop on Machine Learning for Sensory Data Analysis*. ACM, 2013, pp. 19–26. DOI: [10.1145/2542652.2542656](https://doi.org/10.1145/2542652.2542656).
- [192] Dennis L. Wilson. “Asymptotic Properties of Nearest Neighbor Rules Using Edited Data”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 2.3 (1972), pp. 408–421. DOI: [10.1109/TSMC.1972.4309137](https://doi.org/10.1109/TSMC.1972.4309137).
- [193] Seunghye J. Wilson. “Data representation for time series data mining: time domain approaches”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 9.1 (2017), e1392:1–6. DOI: [10.1002/WICS.1392](https://doi.org/10.1002/WICS.1392).
- [194] Svante Wold, Kim Esbensen, and Paul Geladi. “Principal Component Analysis”. In: *Chemometrics and Intelligent Laboratory Systems* 2.1 (1987), pp. 37–52. DOI: [10.1016/0169-7439\(87\)80084-9](https://doi.org/10.1016/0169-7439(87)80084-9).
- [195] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. “Detecting Large-Scale System Problems by Mining Console Logs”. In: *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*. ACM, 2009, pp. 117–132. DOI: [10.1145/1629575.1629587](https://doi.org/10.1145/1629575.1629587).
- [196] Yan Xu, Yanming Sun, Jiafu Wan, Xiaolong Liu, and Zhiting Song. “Industrial Big Data for Fault Diagnosis: Taxonomy, Review, and Applications”. In: *IEEE Access* 5 (2017), pp. 17368–17380. DOI: [10.1109/ACCESS.2017.2731945](https://doi.org/10.1109/ACCESS.2017.2731945).
- [197] Ji Xue, Evgenia Smirni, Thomas Scherer, Robert Birke, and Lydia Y. Chen. “PROST: Predicting Resource Usages with Spatial and Temporal Dependencies”. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2016, pp. 125–126. DOI: [10.1145/2851553.2858678](https://doi.org/10.1145/2851553.2858678).

- [198] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. “An Empirical Study on Configuration Errors in Commercial and Open Source Systems”. In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 2011, pp. 159–172. DOI: [10.1145/2043556.2043572](https://doi.org/10.1145/2043556.2043572).
- [199] Li Yu, Ziming Zheng, Zhiling Lan, and Susan Coghlan. “Practical Online Failure Prediction for Blue Gene/P: Period-based vs Event-driven”. In: *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*. IEEE, 2011, pp. 259–264. DOI: [10.1109/DSNW.2011.5958823](https://doi.org/10.1109/DSNW.2011.5958823).
- [200] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. “SherLog: Error Diagnosis by Connecting Clues from Run-time Logs”. In: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2010, pp. 143–154. DOI: [10.1145/1735970.1736038](https://doi.org/10.1145/1735970.1736038).
- [201] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. “Improving Software Diagnosability via Log Enhancement”. In: *ACM Transactions on Computer Systems (TOCS)* 30.1 (2012), pp. 1–28. DOI: [10.1145/2110356.2110360](https://doi.org/10.1145/2110356.2110360).
- [202] Jie Zhang, Xiaoyin Wang, Dan Hao, Bing Xie, Lu Zhang, and Hong Mei. “A survey on bug-report analysis”. In: *Science China Information Sciences* 58.2 (2015), pp. 1–24. DOI: [10.1007/S11432-014-5241-2](https://doi.org/10.1007/S11432-014-5241-2).
- [203] Qi Zhang, Joseph L. Hellerstein, and Raouf Boutaba. “Characterizing Task Usage Shapes in Google Compute Clusters”. In: *Proceedings of the 5th International Workshop on Large Scale Distributed Systems and Middleware*. ACM, 2011.
- [204] Shenglin Zhang, Ying Liu, Dan Pei, Yu Chen, Xianping Qu, Shimin Tao, and Zhi Zang. “Rapid and Robust Impact Assessment of Software Changes in Large Internet-Based Services”. In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 2015, 2:1–13. DOI: [10.1145/2716281.2836087](https://doi.org/10.1145/2716281.2836087).
- [205] Tao Zhang, He Jiang, Xiapu Luo, and Alvin TS Chan. “A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions”. In: *The Computer Journal* 59.5 (2016), pp. 741–773. DOI: [10.1093/COMJNL/BXV114](https://doi.org/10.1093/COMJNL/BXV114).
- [206] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. “BIRCH: An Efficient Data Clustering Method for Very Large Databases”. In: *ACM SIGMOD Record* 25.2 (June 1996), pp. 103–114. DOI: [10.1145/235968.233324](https://doi.org/10.1145/235968.233324).
- [207] Xiao Zhang, Fanjing Meng, Pengfei Chen, and Jingmin Xu. “TaskInsight: A Fine-Grained Performance Anomaly Detection and Problem Locating System”. In: *Proceedings of the 9th IEEE International Conference on Cloud Computing*. IEEE, 2016, pp. 917–920. DOI: [10.1109/CLOUD.2016.0136](https://doi.org/10.1109/CLOUD.2016.0136).
- [208] Zhiyuan Zhang, Xuehai Tang, Jizhong Han, and Peng Wang. “Sibyl: Host Load Prediction with an Efficient Deep Learning Model in Cloud Computing”. In: *Proceedings of the 18th International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2018, pp. 226–237. DOI: [10.1007/978-3-030-05054-2_17](https://doi.org/10.1007/978-3-030-05054-2_17).
- [209] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. “Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process”. In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 2009, pp. 91–100. DOI: [10.1145/1595696.1595713](https://doi.org/10.1145/1595696.1595713).
- [210] Indrè Žliobaitė, Jorn Bakker, and Mykola Pechenizkiy. “Beating the baseline prediction in food sales: How intelligent an intelligent predictor is?” In: *Expert Systems with Applications* 39.1 (2012), pp. 806–815. DOI: [10.1016/J.ESWA.2011.07.078](https://doi.org/10.1016/J.ESWA.2011.07.078).

Curriculum Vitae

Personal Information



Name: Dipl.-Ing. **Andreas Schörghumer**, BSc
Address: Mannheimstraße 6/11/71, 4040 Linz, Austria
Telephone: +43 664 88583231
E-Mail: andischoe@gmail.com
Nationality: Austria
Date of Birth: 15.05.1993, 4600 Wels, Austria

Professional Experience

Lecturer

03/2019 – Today

Johannes Kepler University Linz – Teaching in the area of software development (computer science)

Researcher

03/2017 – Today

Johannes Kepler University Linz – Data analysis of large, heterogeneous systems of our APM industry partner Dynatrace as part of the Christian Doppler Laboratory for Monitoring and Evolution of Very-Large-Scale Software Systems (MEVSS) until 08/2020, from then on at the Institute for Software Systems Engineering. Application of statistical methods for error analytics, supervised machine learning for the prediction of performance-related events and unsupervised clustering of time series

Student Researcher

09/2015 – 02/2017

Johannes Kepler University Linz – Java lock contention analysis via a sampling-based Java agents with JVMTI as part of the Christian Doppler Labors for Monitoring and Evolution of Very-Large-Scale Software Systems (MEVSS)

Paramedic und Control Operator

07-08/2013 – 2015

Red Cross Eferding (control operator) und Red Cross Wilhering (paramedic)

Paramedic (Civilian Service)

11/2011 – 07/2012

Red Cross Wilhering

Teaching Experience

Teaching

03/2019 – Today

Softwareentwicklung 2 (Software Development 2) in summer semester 2019 and 2020 (hybrid + digital teaching)
Softwareentwicklung 1 (Software Development 1) in winter semester 2019 and 2020 (hybrid + digital teaching)

Thesis Supervision

Bachelor's thesis of Peter Chalupar: "Cross-System Event Prediction with Random Forests using Time Series"
Master's thesis of Mario Kahlhofer: "Exploring Supervised Event Prediction in Multi-System Monitoring"
Master's thesis of Ante Dilber: "Visualizing System Architectures with Time Series and Event Metadata"

Volunteering

Control Operator

09/2013 – 03/2017

Red Cross Eferding

Paramedic

10/2011 – 03/2017

Red Cross Eferding und Red Cross Wilhering

Academic Education

PhD in Technical Sciences 03/2017 – Today

Doktor der technischen Wissenschaften (Dr. techn.) – Johannes Kepler University Linz
PhD thesis: “Data Analysis and Error Analytics in Large-Scale Heterogeneous Software Systems“
Up to now, all subjects graded “sehr gut“ (very good)

Master in Computer Science 10/2015 – 02/2017

Diplom-Ingenieur (Dipl.-Ing.), Major subject: Software Engineering – Johannes Kepler University Linz
Master’s thesis: “Efficient Sampling-based Lock Contention Profiling in Java“
Passed with distinction, all subject groups graded “sehr gut“ (very good), completed below minimum period of study

Bachelor Informatik (Informatics) 10/2012 – 10/2015

Bachelor of Science (BSc) – Johannes Kepler Universität Linz
Bachelor’s thesis: “Improved Eyes-Free Vehicle Control Using Touch Based Gestural Input on the Steering Wheel“
Passed with distinction, all subject groups graded “sehr gut“ (very good)

Senior Grade with Matura (Higher School Certificate) 09/2007 – 07/2011

Emphasis: fine arts – BORG Grieskirchen
Passed with distinction, all subjects graded “sehr gut“ (very good)

Lower Grade 09/2003 – 07/2007

Gymnasium Dachsberg, Prambachkirchen
All subjects graded “sehr gut“ (very good)

Additional Education

Control Operator (Red Cross) 04/2012 – 09/2013

Gruppenkommandant (Red Cross Leadership Training) 10/2012 – 12/2012

Emergency Vehicle Driver (Red Cross) 04/2012 – 08/2012

Knowledge

Languages: German (native), English (fluent), basic knowledge in French und Spanish (A1/A2), Latin (six years)

Programming Languages: Excellent Knowledge in Java and Python, good knowledge in Kotlin, basic knowledge in C, C++, C#, SQL

Tools and Miscellaneous: Microsoft Office (Word, Excel, PowerPoint), LaTeX, Moodle, Zoom, Slack

Awards

Merit-based Scholarship of the Technical and Natural Sciences Faculty

For sessions 2012/2013, 2013/2014, 2014/2015 and 2015/2016

Nomination for the IEEE Student Paper Contest 2016

Submission based on the bachelor’s thesis (touch-based vehicle control)

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references. This printed thesis is identical with the electronic version submitted.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.



Linz, July 12, 2021

Dipl.-Ing. Andreas Schörgenhumer, BSc