Intelligent
Embedded Systems

14

Bernhard Schlegel

Off-Board Car Diagnostics Based on Heterogeneous, Highly Imbalanced and High-Dimensional Data Using Machine Learning Techniques

kassel
university
press

Intelligent
Embedded Systems

Band 14

Herausgegeben von
Prof. Dr. Bernhard Sick, Universität Kassel

Bernhard Schlegel

Off-Board Car Diagnostics Based on Heterogeneous, Highly Imbalanced and High-Dimensional Data Using Machine Learning Techniques

kassel university press

This work has been accepted by the Faculty of Electrical Engineering / Computer Science of the University of Kassel as a thesis for acquiring the academic degree of Doktor der Naturwissenschaften (Dr. rer. nat.).

Supervisor: Prof. Dr. Bernhard Sick, University of Kassel
Co-Supervisor: Prof. Dr. Ludwig Brabetz, University of Kassel

Defense day: 29th May 2019

Bibliographic information published by Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at http://dnb.dnb.de.

Zugl.: Kassel, Univ., Diss. 2019
ISBN 978-3-7376-0738-4 (print)
ISBN 978-3-7376-0739-1 (e-book)
DOI: http://dx.medra.org/10.19211/KUP9783737607391
URN: https://nbn-resolving.org/urn:nbn:de:0002-407391

www.upress.uni-kassel.de

Printed in Germany

Preface

The following Ph.D. thesis was made possible by a cooperation between Prof. Bernhard Sick, head of Lab for Intelligent Embedded Systems at the University of Kassel, and the Bayerische Motoren Werke Aktiengesellschaft (BMW AG). It was carried out from 2015 to 2018. The idea was originally brought up by Axel Knaut from BMW AG, who saw the untapped potential of the available data.

Munich, August 3, 2019

Bernhard Schlegel

Danksagung

Ganz besonderer Dank gilt Prof. Dr. rer. nat. Bernhard Sick, der mich während aller Phasen der Promotion gefordert und gefördert hat und außerdem stets ein offenes Ohr, exzellente Anmerkungen und viel Geduld hatte. Weiter möchte ich Prof. Dr. rer. nat. Ludwig Brabetz für die Zweitgebutachtung meiner Promotion danken.

Außerdem möchte ich Dr.-Ing. Hermann Hajek, Axel Knaut, Dr.-Ing. Dieter Strobel und Dr.-Ing. Florian Preuß und der BMW AG für die Zusammenarbeit, die Freiräume und die Ermöglichung von zahlreichen Konferenzteilnahmen bedanken. Herzlicher Dank geht auch an die Doktoranden und Doktoren des Fachgebietes IES der Universität Kassel für die zahlreichen Anregungen und die tatkräftige Unterstützung sowie an meine Ko-Autoren von BMW, Peter Wolf und Artur Mrowca.

Vielen Dank auch an Marc Kaminski für seine hervorragende Arbeit und viele, interessante Unterhaltungen quer über den Tisch.

Außerdem möchte ich mich herzlich bei meinen Freunden bedanken, insbesondere bei der “Crew” (Daniel, Kevin, Lukas, Moritz, und Dr. rer. nat. Oliver) und Andrea für “inspirierende” Momente. Herzlichen Dank geht außerdem an Micha, die mich stets mit Wissen versorgt hat.

Zuletzt geht mein Dank an meine Eltern Norbert und Marina, die mich von Anbeginn meines Studiums bis in die finalen Phasen der Promotion unterstützt haben und Annika, die mir auch in arbeitsintensiven Zeiten zur Seite stand.

Abstract

Data-driven maintenance poses many challenges. Four very important of them, namely coping with a high dimensional and heterogeneos feature space, the highly imbalanced data sets, the Remaining Useful Lifetime (RUL) prediction of monitored parts based on short yet variable length timeseries, and already large but steadily further increasing data set size are identified. Each of the challenges is dealt with in one chapter. Novel techniques are designed, implemented, validated, and compared to existing approaches based on a variety of (publicly available) data sets for general applicability. In the following multiple concepts are proposed and evaluated in great detail: A feature selection pipeline with multiple consecutive stages of increasing run-time complexity but also increasing accuracy to tackle the high dimensional feature space. Existing techniques to tackle imbalance are evaluated and compared to a novel technique that stands out due to its extremely low computational complexity. Two novel techniques based on cascaded Random Forests (RFs) and on density-based estimation that outperform current state of the art techniques for RUL prediction. And finally: The evaluation of an in-memory cluster computing framework regarding its suitability for not only large-scale data set extraction from a relational database, preprocessing and transformation of the dataset but also machine learning.

Zusammenfassung

Die datengetriebene Wartung und Instandhaltung birgt eine Vielzahl von Herausforderungen. Vier sehr wichtige von ihnen wurden identifiziert: Die hohe Dimensionalität und Heterogenität des vorliegenden Merkmalsraumes, die hohe Imbalance der Datensätze, die Vorhersage der Restlebensdauer von überwachten Komponenten auf Basis von kurzen bzw. unterschiedlich langen Zeitreihen und die bereits sehr große und kontinuierlich weiter wachsendende Menge von Daten. Jeder dieser Herausforderungen ist ein dediziertes Kapitel gewidmet. Hierzu wurden neuartige Techniken entwickelt, implementiert, validiert und mit existierenden Ansätzen auf Basis einer Vielzahl von teilweise öffentlich verfügbaren Datensätzen hinsichtlich ihrer allgemeinen Anwendbarkeit verglichen. Folgende Konzepte werden vorgestellt und im Detail bewertet: Eine Pipeline zur Merkmalsauswahl mit mehreren, aufeinander folgenden Schichten mit jeweils steigender Berechnungskomplexität und Genauigkeit, um wichtige Merkmale aus hochdimensionalen Merkmalsräumen zu extrahieren. Existierende Techniken zur Beherrschung starker Imbalance werden evaluiert und mit einer neuartigen Technik, die eine extrem geringe Berechnungskomplexität aufweist, verglichen. Zwei neuartige Techniken auf Basis von kaskadierten Random Forests bzw. auf Basis von Dichteschätzung werden vorgestellt. Diese über-treffen bereits existierende Lösungen zur Vorhersage der Restlebensdauer von Komponenten. Zum Abschluss werden die vielversprechendsten Methoden für ein In-Memory Cluster Computing Framework implementiert und dieses hinsichtlich seiner Eignung zur Datenextraktion und -transformation sowie zur Modellbildung, untersucht.

Contents

Preface v

Acknowledgment vi

Abstract vii

Zusammenfassung viii

1 Introduction **1**

1.1 Motivation 1

1.2 Data Sources 3

1.3 Problem Formulation 4

1.4 Objectives 6

1.5 Structure of this Thesis 7

1.6 List of Relevant Publications 7

2 Preliminary Considerations **9**

2.1 Notation 9

2.2 Machine Learning Models 10

2.2.1 Random Forests 10

2.2.2 Logistic Regression 13

2.2.3 K-nearest Neighbor 14

2.3 Measuring Classification Performance in Imbalanced Scenarios 16

3 Feature Selection **21**

3.1 State of the Art 22

3.1.1 Diagnostics 22

3.1.2 Filter Measures 23

3.1.3 Wrapper and Embedded Measures 33

3.2 Research Demand . . . 35
3.3 Data Sets . . . 36
3.3.1 Diagnostic Automotive Data . . . 36
3.3.2 Publicly Available Data Sets . . . 37
3.4 Proposed Solution . . . 38
3.4.1 Preparation Layer . . . 39
3.4.2 Filter Layer . . . 40
3.4.3 Wrapper Layer . . . 41
3.4.4 Model Layer . . . 42
3.4.5 Hyperparameters . . . 42
3.5 Evaluation . . . 43
3.5.1 Detailed Look on Feature Group Importance . . . 44
3.5.2 Filter Layer Evaluation . . . 46
3.5.3 Wrapper Layer Evaluation . . . 49
3.5.4 Remaining Hyperparameters . . . 50
3.5.5 Evaluation on Publicly Available Data . . . 55
3.6 Summary and Conclusion . . . 57

4 Dealing with Imbalance **62**

4.1 State of the Art . . . 63
4.1.1 Preprocessing Techniques for Imbalanced Data Sets . . . 65
4.1.2 Objective Feature Noise, Borderline, and Overlap Measure . . . 73
4.2 Research Demand . . . 75
4.3 Data Sets . . . 76
4.4 Evaluation . . . 79
4.4.1 Preliminary Investigations . . . 80
4.4.2 Robustness . . . 82
4.4.3 Influence of Preprocessing Techniques on the Classification Performance . . . 84
4.4.4 Computational Complexity . . . 90
4.5 Lessons Learned and Conclusions . . . 91

5 Remaining Useful Lifetime **94**

5.1 Notation and Definitions 95
5.2 State of the Art 95
5.3 Research Demand 98
5.4 Data Sets 99
5.5 Proposed Solution 101
5.5.1 Approach of Wang 102
5.5.2 Polynomial-Based Feature Selection 105
5.5.3 Distribution-Based Similarity Estimation 106
5.5.4 Bucketized RUL Regression 110
5.6 Evaluation 113
5.6.1 Hyperparameters of Distribution-based Similarity Estimation 113
5.6.2 Hyperparameters of Bucketized RUL Regression 115
5.6.3 Summary 122
5.7 Conclusion and Outlook 124

6 Apache Spark and Application **126**

6.1 State of the Spark 127
6.1.1 Immutable Data Sets 129
6.1.2 Actions and Lazy Transformations 129
6.1.3 Estimators and Transformers 130
6.1.4 Ephemeral Intermediate Results 130
6.2 Research Demand 130
6.3 Apache Spark Pipeline 131
6.3.1 Preprocessing 132
6.3.2 Modeling 135
6.3.3 Cluster Setup 137
6.3.4 Graphical User Interface and Backend 137
6.4 Evaluation 138
6.4.1 Limitations of Apache Spark 139
6.4.2 Data Sets 143
6.4.3 Experiments 145
6.5 Brief Economic Analysis 151

6.6 Conclusion 154

7 Summary **155**

7.1 Summary, Conclusions, and Discussion 155

7.2 Recommendations for Further Work 158

A Spark Cluster Setup and Application Launch **160**

Chapter 1

Introduction

The growing complexity and diversity of vehicle systems render it increasingly hard to identify and resolve the root cause of an unplanned maintenance session at hand in a dealer workshop. This especially holds for workshop staff with limited qualifications and experience. By today, the workshop staff is supported by expert-based systems, where knowledge was manually generated, i. e., by formalizing human knowledge using rules.

However, this approach does not seem to be effective, since misdiagnosed and misrepaired cars are an steadily increasing cost factor in the automotive field[1]. Especially given the growing variant diversity, differing environmental conditions, and the increasing product complexity, the current expert-knowledge-based offboard diagnostics practiced in workshops today seem to be doomed.

On the other hand, rich car- and workshop-data is already available today and remains widely unused. Due to the telematics enabled data collection, the amount of data is expected to grow even further in the future. A highly autonomous, machine-learning-based approach seems promising since the large amount of data is presumably sufficient to automatically model even complex relationships and error patterns.

1.1 Motivation

According to a survey performed by BearingPoint [7], being unable to identify the issue (and thus performing the right actions) is the biggest contributor to warranty incidents for suppliers and the third biggest contributor for Original Equipment Manufactur-

[1] From 2011 to 2016, the warranty cost of BMW increased by 19% every year [9, 10, 11, 12, 13, 14] on average.

ers (OEMs) in the automotive industry. The great importance of so-called No Trouble Found (NTF) cases are a strong indicator that the current expert knowledge-based diagnostics are unable to cope with the constantly growing demands on the workshop and vehicle diagnostics. The reasons for this are manifold: The complexity of cars is constantly growing due to increased connectivity between and inside cars as well as trends such as hybridization. Especially the latter yields far more complex drivetrains incorporating not only a combustion engine, such as in conventional cars, but also an electric engine and a high voltage energy storage. The high level of interdependencies between components make diagnosing these systems very hard using systems based on Knowledge Databases (KDBs) [4] that are essentially based on often manually generated `if ... then ... else ...` rules [119].

Todays car OEMs offer extensive possibilities to customize the car upon order. Different engines, equipment packages, colors, and other optional extra equipment cause that only a few vehicles are identical. In addition, vehicles are used in different regions (e.g. with regard to fuel quality) differently (depending on driving style). A preliminary data analysis[2] shows that even a way less customizable hybrid vehicle offers approximately 9500 unique configurations. This means that there are on average three vehicles with the same configuration in the data set. Representing and considering this high variety of possible configurations using classical KDB-based systems, and ensuring that they are up-to-date, e.g. when new software updates are released, is tremendously challenging.

Evolving countries continue to gain importance for automotive OEMs. In 2016, 43% of new cars were sold in evolving markets such as China, India, and Brazil [121]. Since it is not standard in these countries (unlike Germany) to complete several years of training before working as a mechanic, it is difficult to find qualified personnel for the workshops. This, however, is a crucial requirement to perform tests and follow the recommendations given by the KDB properly.

The results can be severe: Low customer satisfaction due to wrong or unnecessary expensive repairs, and high warranty costs as well as damage to the premium image of the OEM. At the same time, modern vehicles produce a rich set of data that has the potential to revolutionize how cars are diagnosed and repaired in a more generic, precise and autonomous way. This data is already available today, but is rarely used because of the major challenges

[2] Performed on the *Automotive 2L* data set from Chapter 6.

involved, as described in the following. However, its use is promising and will be investigated in this thesis.

1.2 Data Sources

The starting point to enable data-driven car diagnostics is historical, non-personal, labeled data. For this work, the data sets were collected from a fleet of modern hybrid vehicles. In general, every time a vehicle is brought to a BMW dealership and read out using the OBD-II port or using telematics, a sample is created. Each sample consists of the following feature groups (that consist of multiple features):

- Readout data (RO): This holds basic information such as when or in which dealership the readout was performed. Also, software version information of the car and the mileage are included.
- Car Parameters (CPs): This is static information about the car that was defined during production, such as the type of car or the engine.
- Extra Equipments (EEs): This, foremost `Boolean` typed feature group indicates whether a certain optional extra equipment is present or not. Examples include leather seats, light and comfort packages, fast charging or if a trailer hitch is available.
- Diagnostic Trouble Codes (DTCs): A car constantly compares measured values (e.g. the gasoline consumption) to a model-based prediction of the same value. Based on the self-diagnostic capabilities of Electrical Control Units (ECUs), a DTC is flagged autonomously if the discrepancy grows too large.
- Environmental conditions (ECs): These are usually logged at the same time when a DTC is flagged and allow to reconstruct the state of the car at the time of the error later on.
- Measurement Values (MVs): MVs allow for assessing the internal state of the car, e.g. the coefficients of adaptive controllers, and other values to enable conclusions to be drawn about how the car was moved.

The potential targets of a model include parts that were switched (Switched Parts (SPs)), counter actions that were taken (Taken Action (TA)), and Diagnostic Codes (DCs). The latter

is a hash like ID that summarizes potentially multiple SPs and TAs. In addition, a variety of publicly available data sets tailored to the specific aspects of the corresponding chapters were identified: Chapter 3 uses the *golub* [48] and *secom* data sets [77], Chapter 4 includes the *vehicle* [117], *vowel* [125], and *forest* data sets [32], Chapter 5 uses the *PHM08* [104], *turbofan* [103], and *SML2010* [146] data sets, and Chapter 6 relies on the *Credit Card* [34] and *NIPS* [53] data sets.

1.3 Problem Formulation

Among others, *four* major challenges need to be overcome to enable data-driven, autonomous, and *generic* automotive diagnostics that are not tailored towards a specific component but able to diagnose basically any part where data is available (*generic*). These are:

(1.) *The high dimensional and heterogeneous feature space*: As mentioned earlier, a multitude of data of data is already available today. However, this poses the challenge to identify the few (usually up to 20, see Chapter 3) useful features required to create meaningful and well generalizing models from the several thousand features available. This is specifically challenging, if the feature space is heterogeneous, as shown in Table 1.1: In the automotive context, a variety of different features types occur. The various column header prefixes match the abbreviations introduced in Section 1.2. Feature datatypes include: Ordinal integers or floating values, categorical integers, and strings. They all need to be treated differently. Also, features differ in their sparsity (consider, e.g., `SC_IP` in Table 1.1 which is very sparse).

Table 1.1: Example data set for automotive diagnostics.

MV_S	MV_1	MV_3	MV_BG	EE_TP	SC_IP	SC_1	SC_2	DTC_PU	DTC_1	DTC_2	CP
44	3	20	-0.06	False	2	77	27	False		True	"v.10a"
72	36	73	-0.01	False		16	29			False	"v.10a"
100	4	16	-0.02	True		45	1		False	False	"v.10b"
44	14	54	-0.02	True		76				False	"v.10b"
95	34	73	-0.07	False		80	22		False	False	"v.10a"
16	50	33	-0.02	True		61	93	False	False	False	"v.11x"
	4	27	-0.09	False		59	91			False	"v.10a"
	48	20	-0.07	False		32	31			False	"v.10a"
88	60	72	-0.01	True	1.9	96	53	True	False	True	"v.10a"
27	14	88		False		73	14			False	"v.11x"

(2.) *The highly imbalanced data sets*: Thanks to the high quality standards of todays cars, errors are not the norm but the exception. The circumstance that every time a vehicle is read out in the workshop or via telematics a data sample is created (represented by one row in Table 1.1) causes a strong imbalance in the data set. The small number of samples in which the problem occurred compared to the high number of samples where another or no problem occurred leads to a high imbalance in the data set. This makes model training extremely difficult. In addition, due to the aforementioned variety of vehicles offered by each OEM, the possible causes for faults and fault patterns are extremely diverse. This further increases the imbalance and makes it even more difficult to train meaningful and well generalizing models.

(3.) *Short time series of varying length*: Generally speaking, the automotive data sets are panel data sets: Multiple objects (cars) are measured over multiple time periods. Especially for predicting the RUL (and thus enabling *predictive maintenance*), considering temporal connections during model creation promises an increased model accuracy[3]. However, time series data from cars in diagnostic contexts tends to be very short (e.g. only three samples "long"). In addition, the length of the time series depends extremely on the observed component and the type of data collection (workshop or telematics). For these two reasons, classical time series approaches such as windowing [92] are out of the question. In order to meet the generic requirement, however, approaches are required that can cope with short and variable-length time series.

(4.) *Large, continuously growing data volumes*: This work examines (among other data sets) two automotive diagnostic data sets in different versions. While the first one originated from workshop readout processes, the second one was collected via telematics. With 3107 features and 121900 samples, the first automotive data set was already larger than all publicly available data sets evaluated in the context of this work. However, the second data set was even larger. The latter was not processable in-memory on a single computer. Should data collection via telematics gain in importance – which is to be assumed due to the advantages such as the higher sampling rate – the data volume will continue to increase. This places demands on data-driven diagnostics with regard to their scalability in order to cope with presumably ever-increasing data volumes.

Many of the above mentioned challenges apply not only to automotive diagnostics. As

[3] This is referring to the accuracy in general, not the definition of accuracy derived from a confusion matrix.

more and more data becomes available every day, challenges such as imbalance and being able to process large-scale data sets apply to many industries and products.

1.4 Objectives

The overall objective of this thesis is to develop, implement, and extensively evaluate a machine learning system which is able to classify whether a counter measure is suitable or to predict the RUL of a component (regression). Based on the aforementioned challenges, the following main objectives are derived:

1. The definition of a meaningful error score that allows to accurately assess the classification performance in the given, imbalanced scenario as discussed in challenge 2 (imbalanced data sets). This also affects challenge 1 and 4 since a meaningfull error score is a prerequisite for all subsequent considerations.

2. Also, to tackle challenge 1, the combination and adaptation of techniques that are able to select relevant features from a high dimensional feature space is required.

3. The development and comparison of techniques to tackle the high imbalance of the automotive and other, publicly available data sets (challenge 2).

4. The development of techniques to accurately estimate the RUL based on short, variable length time series according to challenge 3.

5. The identification and adaptation of a multi node in-memory computing frame work to deal with large data volumes according to challenge 4. The scalability and performance in comparison to single node frameworks shall be examined. Also, potential caveats and corresponding countermeasures need to be identified.

6. All objectives defined above must meet the “generic” requirement. The declared objective of this thesis is to create the foundation for automated diagnosis of errors and prognosis of RULs independent of the examined or modeled component. Thus, evaluation shall be based on a variety of different data sets.

7. The evaluation and optimization shall be laid out in respect to both, classification or regression grade *and* run-time complexity.

1.5 Structure of this Thesis

The rest of this work is structured as follows. Chapter 2 introduces the notation (Section 2.1) and machine learning models (Section 2.2) used across all following chapters. Also, objective 1 is tackled and a meaningful error measure for classification is identified (Section 2.3).

In Chapter 3, a feature selection pipeline is proposed to tackle objective 2. Chapter 4 will evaluate various techniques to cope with imbalanced data sets thus satisfying objective 3.

Chapter 5 will propose and evaluate techniques to process short and variable time series (objective 4).

In Chapter 6, an in-memory cluster computing framework will be evaluated regarding it suitability for data-driven workshop diagnostics (objective 5).

Chapter 7 will summarize the results of this work and give an outlook for further research. All chapters will also consider objectives 6 and 7. Generic applicability (objective 6) is ensured by evaluating on a variety of different data sets.

For all experiments, the runtime is also logged so that this can be taken into account in the subsequent run-time complexity evaluation (objective 7). The relevant state of the art will be presented in Chapters 3, 4, 5, and Chapter 6, respectively. All existing and proposed approaches are evaluated based on the results originating from extensive experiments which are described in the corresponding chapters.

1.6 List of Relevant Publications

The following publications directly emerged from the work on this thesis:

- B. Schlegel and B. Sick. Design and Optimization of an Autonomous Feature Selection Pipeline for High Dimensional, Heterogeneous Feature Spaces. In Proceedings of the 8th IEEE Symposium Series on Computational Intelligence (*SSCI16*), pages 1-9, Athens, Greece, [109].

- B. Schlegel and B. Sick. Dealing with Class Imbalance the Scalable Way: Evaluation of Various Techniques Based on Classification Grade and Computational Complexity. In 2017 IEEE International Conference on Data Mining Workshops (*ICDMW*), pages 69-78, New Orleans, USA, [108].

- B. Schlegel, A. Mrowca, P. Wolf, B. Sick, and S. Steinhorst. Generalizing application agnostic remaining useful life estimation using data-driven open source algorithms. In 2018 IEEE 3rd International Conference on Big Data Analysis (*ICBDA*), pages 102–111, Shanghai, China, [107].

- B. Schlegel and M. Kaminski. Next Generation Workshop Car Diagnostics at BMW Powered by Apache Spark. Presented on the 2017 Spark Summit, San Francisco, USA, [105].

In Schlegel and Sick [109] a feature selection pipeline based on several layers of differing run-time complexity is developed, evaluated and optimized. The results of Chapter 3 can be found in this article. Schlegel and Sick [108] surveys and adapts various techniques, and also introduces a novel technique to tackle imbalance in classification scenarios. This publication forms the foundation for Chapter 4. Chapter 5 is based on Schlegel et al. [107], where novel techniques to estimate the RUL are proposed and compared to current, state of the art RUL estimation techniques. The presentation by Schlegel and Kaminski [105] presents a machine learning pipeline for large scale data sets built on top of Apache Spark. Among other things, the findings presented there form Chapter 6.

Chapter 2

Preliminary Considerations

While each of the following chapters introduces techniques relevant only for the respective chapter, this chapter explains the techniques used across *multiple* of the following chapters. First, the notation used in all following chapters will be defined (Section 2.1). Then, relevant model types (classifiers and regressors) are introduced (Section 2.2). A question that is of high relevance to all classification focused following chapters is "how to reliably assess the model performance in imbalanced scenarios?" (in terms of classification grade). This question and its answer are addressed in Section 2.3.

2.1 Notation

The following notation is used to ensure a clearer understanding of the theoretical introduction in the next section and all following chapters. The common notation of N referring to the number of samples (observations) and P referring to the number of features (predictors) is used as also shown in Equation (2.1) :

- $\mathscr{X} = \{X_1, X_2, ..., X_P\}$ is a set of input variables, also called *features*, of a machine learning problem. In general, feature values can be classified into three types [24]: Continuous or discrete, categorical (nominal), and ordinal. Continuous features can take any value in a given interval, discrete features can only take integer values. Categorical features can take only a specific set of values from a set of possible categories (The size of the set is referred by $|\mathbb{D}_X|$). Ordinal features are categorical features with an explicit ordering. Here, the features' domains are given by $\mathbb{D}_{X_1}, \mathbb{D}_{X_2}, ..., \mathbb{D}_{X_M}$, respectively. The domain of $\mathscr{X}$ is thus given by $\mathbb{D}_{\mathscr{X}} = \mathbb{D}_{X_1} \times \mathbb{D}_{X_2} \times \ldots \mathbb{D}_{X_P}$.

- Y has the domain $\mathbb{D}_Y$ and is the output, also called label or target, of a machine learning problem. The domain can be either binary (Boolean) for classification (Chapter 3, Chapter 4, Chapter 6) or continuous/ discrete for regression (Chapter 5).
- $\mathscr{S} = \{s_1, s_2, ..., s_N\}$ refers to the set of N training samples.
- Every sample $s_n = (\mathbf{x_n}, y_n)$ consists of values of P features $\mathbf{x_n} = (x_{n,1}, ..., x_{n,P})$ and the target value (e.g., for binary target variables $y_n \in \{0, 1\}$ for a class $\mathscr{C}$).

$$\begin{array}{c} \\ s_1 \\ s_2 \\ \vdots \\ s_N \end{array} \begin{array}{c} \begin{array}{cccc} X_1 & X_2 & \dots & X_P \end{array} \\ \begin{pmatrix} x_{(1,1)} & x_{(2,1)} & \dots & x_{(1,P)} \\ x_{(2,1)} & x_{(2,2)} & \dots & x_{(2,P)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{(N,1)} & \dots & \dots & x_{(N,P)} \end{pmatrix} \end{array} = \begin{array}{c} Y \\ \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} \end{array} \tag{2.1}$$

2.2 Machine Learning Models

This section presents a short summary of the model types used extensively in the following. More details are are given in Duda [39], Marsland [82], Murphy [87] and the Bruce brothers [24].

2.2.1 Random Forests

Random Forests (RFs) are an *ensemble* of multiple Decision Trees (DTs) and can be used for both, classification and regression. A DT tries to subsequentially generate splitting rules to divide the presented data set into "purer" sub-data sets on the way from the trunk to the leaves. For classification, *impurity* is measured using the Gini impurity and negative cross–entropy (which is described in greater detail in Section 3.1.2) in the following. The Gini criterion[1] $\mathbb{G}$ for of a (sub-data) set is defined based on the label Y as follows [22, 8]:

$$\mathbb{G}[Y] \equiv -\sum_{k=1}^{K} p(Y = k) \cdot (1 - p(Y = k)) \tag{2.2}$$

with $K = |\mathbb{D}_Y|$ referring the number of states, $p(Y = k)$ refers the probability of label Y taking the value of $k \in \mathbb{D}_Y$. For regression, impurity is measured using squared deviations from the

[1] Despite being the same, the "Gini criterion" by Breiman [20] is also called "Gini index" by Bishop [8] and "Gini impurity" by Duda [39]. These terms refer all a impurity measure similar to the entropy (see Equation (3.2)), but save computational complexity due to the missing log.

mean in the corresponding sub-data set. The purity gain Δ achieved by splitting is defined as:

$$\Delta(D) = I(D) - \frac{N_L}{N} I(D_L) - \frac{N_R}{N} I(D_R), \tag{2.3}$$

with D referring the data set before the split, D_L and D_R the left and right sub-data sets after the split, respectively. I is one of the aforementioned measures of impurity.

The algorithm to construct a DT is called *recursive partitioning* [24]: Repeated partitioning of the sub-data sets based on feature values to create the most homogeneous sub-data sets possible while paying attention to the stopping criteria which avoids, e.g., that each leaf only holds a single example. To stop a DT to overfit to the (noise in the) training data set by creating too many rules yielding pure leaf nodes, several concepts exist: Splitting can be stopped if the sub-data set in the resulting terminal leaf is too small or if the new partitioning does not significantly decrease the impurity.

An upside–down visualization for a single DT for a multi–class classification is given in Figure 2.1: The original data set (the trunk) at the top holds three different types of beer. The first split, according to the alcohol level, splits the data set into two sub-data sets, consisting of Lager and Pale Ale, and Pale Ale and India Pale Ale (IPA), respectively. Another split is performed based on the International Bitter Unit (IBU) reducing the impurity even further (the sub-data sets in the leave nodes mostly consist of one class only).

The RFs used in the following are created from multiple DTs, based on the bootstrap aggregating (bagging) technique proposed by Breimann [21], where each tree is not only trained on a subset of samples drawn with replacement from the training set but also on a subset of features.

The RFs have serveral advantages: They are able to model non–linear relationships and they are interpretable by human experts (although the usually high number of trees make it harder in comparision to a single DT) which is important in the given, automotive diagnostic context. Also, DTs (and therefore RFs) can provide a probability estimate based on the class ratio in the leaf.

Disadvantages include the greedy optimization strategy (and therefore the risc of getting stuck in local optima, with the target function being the measure of impurity I), the issue that small errors close to the trunk yield big estimation errors in the leafes [87], and the risk of overfitting by relaxed stopping criteria yielding very deep trees.

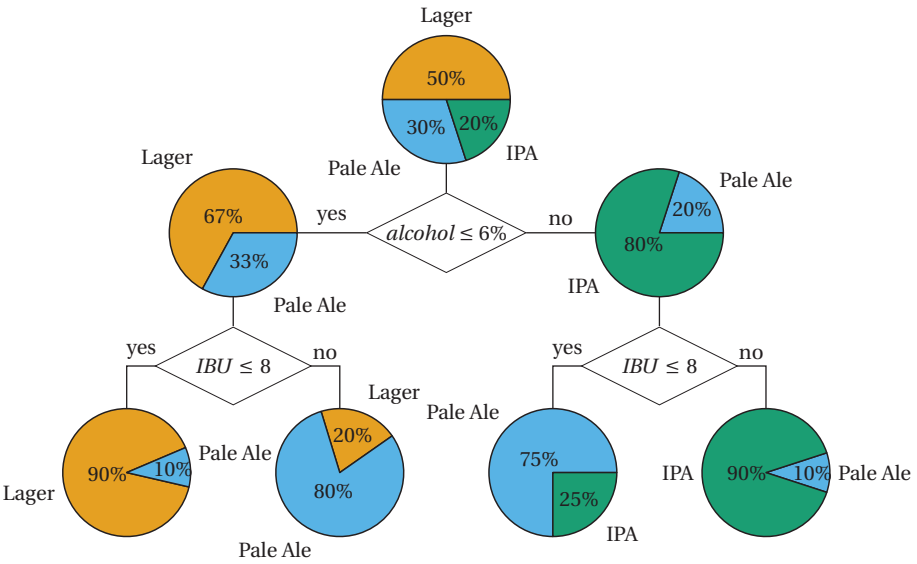


Figure 2.1: Example of a DT.

For cluster computing, as described in Chapter 6, the following optimizations are performed:

- *Partitioning*: The sampled data sets are, according to the used bagging approach, distributed among different workers.

- *Binning*: Continuous feature values are *binned* ("discretized") into a given number of bins. This enables the algorithm to identify potential splitting thresholds by looking at the bin border. This technique yields major performance gains by trading-off splitting accuracy, since it would be necessary to sort the data set by every feature otherwise (which is very expensive on a distributed data set).

- Additional performance is gained, by running the decision tree algorithm on all nodes for each level of a tree simultaneously.

2.2.2 Logistic Regression

The Logistic Regression (LR) is a special version of the Generalized Linear Model (GLM) to extend linear regression to other settings. It is characterized by two main components [24]: The probability distribution or family (binomial if LR is used for two-class classification), and a link function mapping the linear response to the binary target variable. This is done by using a logistic sigmoid function σ. The "logit" function σ is defined as

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.4}$$

yielding a probability, bounded between 0 and 1, that the label is a "1":

$$P(y|\mathbf{x}_n, \mathbf{w}) = \frac{1}{1+e^{-y\mathbf{w}^T\mathbf{x}_n}}, \tag{2.5}$$

with $\mathbf{w}$ being the model parameters ("weights") that need to be fitted to the data. This is done based on an maximum likelihood estimation (MLE). It aims to find the model that most likely produced the training data by maximizing the log-likelihood [88]. The log-likelihood function is defined as

$$L(\mathbf{w}; X, Y) = \sum_{n=1}^{N} \log P(y_n \mid \mathbf{x}_n; \mathbf{w}) \tag{2.6}$$

Due to its superiority as elaborated in more detail in Section 3.1.3, L_1 regularized LR will be used in the following. Regularization is achieved by adding an additional term to the function to be optimized (yielding the final weights $\hat{\mathbf{w}}$):

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w},C} \left(CL(\mathbf{w}) - \|\mathbf{w}\|_1\right), \tag{2.7}$$

with $\|\mathbf{w}\|_1$ denoting the 1-norm $\sum_{i=1}^{n}(w_i)$ and C indicating the trade-off between regularization (shrinking $\mathbf{w}$) and correct classification (maximizing the log-likelihood function). This

can be transformed using the following steps[2]

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w},C} \left(C \sum_{n=1}^{N} \log P(y_n \mid \mathbf{x}_n; \mathbf{w}) - \|\mathbf{w}\|_1 \right), \tag{2.8}$$

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w},C} \left(C \sum_{n=1}^{N} \log \frac{1}{1+e^{-y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n}} - \|\mathbf{w}\|_1 \right), \tag{2.9}$$

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w},C} \left(C \sum_{n=1}^{N} \log(1) - \log(1+e^{-y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n}) - \|\mathbf{w}\|_1 \right) \tag{2.10}$$

into the following constrained minimization problem which is solved using the coordinate descent algorithm[3] for all instance-label pairs $(\mathbf{x}_n, y_n), n = 1, ..., N, \mathscr{X}_p \in \mathbb{D}_{X_p}, y_n \in \{-1, +1\}$ during training (notation used by Fan et al. [41])

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w},c} \left(C \sum_{n=1}^{N} \log(1+e^{-y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n}) + \|\mathbf{w}\|_1 \right). \tag{2.11}$$

This is equal to a different notation (e.g. used by Ng [88]), where C is replaced by $\frac{1}{\alpha}$:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w},c} \left(\sum_{n=1}^{N} \log(1+e^{-y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n}) + \alpha \|\mathbf{w}\|_1 \right). \tag{2.12}$$

Advantages of LR include the output of a "predicted probability" (ranging from 0 to 1) which allows for a finer differentiation of the classification performance (as described in Section 2.3) and easy interpretability of the models (weights reflect feature importances). Moreover, it is being computationally extremely fast.

On the other hand, LR assumes that a parametric linear relationship between the features $\mathscr{X}$ and labels $\mathscr{Y}$ exists [24]. Also, LR is prone to severe overfitting for data that is linearly separable: When the hyperplane separating the two classes is defined by $\mathbf{w}^{\mathrm{T}}\mathbf{x}$, the magnitude of $\mathbf{w}$ can go to infinity [8].

2.2.3 K-nearest Neighbor

A concept that is often used as a baseline classifier, and also embedded into other approaches (see e.g. Section 4.1.1) used in the following is the k-nearest neighbors (k-NN) approach. k-NN can also be used for regression [24, 82]. It is an instance–based or non–

[2] Using $\log(a/b) = \log(a) - \log(b)$ and $\log(1) = 0$.
[3] For details please refer to Fan et al. [41].

generalizing[4] [91] algorithm that predicts class membership based on density estimation [8].

The basic idea is to find the k closest samples (in terms of similar feature values). The majority class of the neighborhood is then assigned to the new sample under consideration. Choosing the right k is the most important hyperparameter tuning to be performed when using a k-NN. While a low k is prone to overfit, a high k may oversmooth / underfit the data [24]. Another hyperparameter is the used *distance metric* that determines which the "closest samples" are. The most widely used is the Euclidean distance between two samples s_1 and s_2 ($dist = \sqrt{(x_{1,1} - x_{2,1})^2 + ... + (x_{1,P} - x_{2,P})^2}$).

The classification is formally based on the Bayes' theorem [39]: The class conditional density for a new sample $s = (\mathbf{x}, y)$ is estimated by

$$p(\mathbf{x} \mid \mathscr{C}_i) = \frac{k_i}{N_i V} \tag{2.13}$$

with k_i being the number of samples from class $\mathscr{C}_i$ among the k nearest neighbors, N_i being the total number of samples for $\mathscr{C}_i$ in the data set, and V being the hyper-sphere volume (defined by the most distant neighbor). The class priors are estimated by

$$p(\mathscr{C}_i) = \frac{N_i}{N}. \tag{2.14}$$

The unconditional density for sample $s = (\mathbf{x}, y)$ is estimated with

$$p(\mathbf{x}) = \frac{k}{NV}, \tag{2.15}$$

with N samples in total (see Section 2.1). Together, this leads to

$$p(\mathscr{C}_i \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid \mathscr{C}_i)\, p(\mathscr{C}_i)}{p(\mathbf{x})} = \frac{k_i}{k}. \tag{2.16}$$

An advantage of k-NN is the low amount of model hyperparameters: Aside from the distance measure (which is usually set prior to training and may be calculated using a kernel), k is the only one remaining to be tuned during, e.g. Cross-Validation (CV). k-NN is also able to predict probabilites. Instead of assigning the majority class, using the result from Equa-

[4] Since no generalizing decision rules are inferred but new sampes are compared only with already known ones.

tion (2.16) can yield class probabilities in multi-class scenarios. Another big advantage is that no model needs to be fitted. One one hand, this is beneficial, since this saves training time. On the other hand, the whole training set needs to be stored as reference. This can be very memory consuming or even infeasible on large data sets. Techniques introduced in Chapter 4 can remedy this effect. Also, all predictors need to be in numeric form, e.g. when using a Euclidean distance measure.

2.3 Measuring Classification Performance in Imbalanced Scenarios

In order to be able to compare the various methods and algorithms in the following, an objective performance measure is needed that is suitable for the imbalanced scenario at hand. This section introduces the metrics used across the following chapters (especially Chapter 3, 4, and 6). Great emphasis is placed on obtaining meaningful values even in imbalanced scenarios.

Suppose, e.g., an LR classifier which we would like to evaluate in the following, where the "predicted probability" is given by the output of the LR model: If the prediction equals to 0.99, the model considers a result of 1 to be very likely. If the output is 0.63 the model is definitely less "sure" what the output will really be.

All measures calculated from a confusion matrix (Table 2.1) such as accuracy, precision, recall, $F_{\beta=1}$ (F_1), etc. inherently disregard the predicted probability. This is due to the way the confusion matrix (Table 2.1) is constructed: All predictions need to be either assigned "predicted positive" or "predicted negative". If the used model outputs a predicted probability (e.g., ranging from $0-1$), this is achieved by setting a threshold, e.g. at 0.5. This means, all predictions higher than this threshold will be flagged as "predicted positive". It is obvious, that this process removes helpful information (how sure was the model about the prediction?) that could be used to asses the model quality.

Table 2.1: Confusion matrix.

	true positive	true negative
predicted positive	True Positive (TP)	False Positive (FP)
predicted negative	False Negative (FN)	True Negative (TN)

The formulas for important metrics used in this work are:

$$accuracy = \frac{TP + TN}{TP + FN + TN + FN}, \tag{2.17}$$

$$fall\text{-}out = ROC_x = \frac{FP}{TN + FP}, \tag{2.18}$$

$$precision = \frac{TP}{TP + FP}, \tag{2.19}$$

$$recall = ROC_y = \frac{TP}{TP + FN}, \tag{2.20}$$

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}, \tag{2.21}$$

with TP being the true positives, FN being the false negatives, FP being the false positives, and TN being the true negatives.

In contrast, the Receiver Operating Characteristic (ROC) curve and the Precision Recall Curve (PRC) are 2D curves which take the predicted probability into account. Both curves are created by systematically varying the classification threshold. This way, the same predicted probability can yield a positive prediction (e.g., if the threshold is low) or a negative prediction (e.g., if the threshold is high) depending on the threshold. Since this also results into an altered confusion matrix, each threshold yields a distinct point in the ROC (defined by (ROC_x, ROC_y)) or PRC space (defined by $(precision, recall)$). For detailed information how to draw an ROC curve or PRC based on these "predicted probabilities", Davis [36], Metz [83], and He and Garcia [57] are referred. Integrating these curves results in an one-dimensional characteristic, the area under Receiver Operating Characteristics Curve (auROC) or area under Precision Recall Curve (auPRC), respectively. A scalar value is preferable, as this property simplifies a comparison.

Also, a measure that measures, e.g., whether the right measure is among the three highest predictions would have been conceivable. However, self-defined key measures cannot usually be transferred to public data sets and make it difficult to compare the approaches

proposed in the following with existing literature, as this usually uses one of the measures defined above.

The widely used accuracy (Equation (2.17), [43]) does clearly not work in imbalanced scenarios [55, 138, 58, 64, 74]. If the minority class is only present in 0.1% of the samples, a classifier will achieve 99.9% accuracy by simply neglecting the existence of the minority class. Similar issues arise with any other metric using values from both columns of the confusion matrix (Table 2.1).

The auROC is a very popular metric [112] in imbalanced scenarios (e.g., used by Guyon et al. [54] and Kubat et al. [73]). However, there exist scenarios where even the auROC can be misleading. To illustrate this, multiple examples and the following notations are used: $\mathscr{S}_P$ refers to the set of all actually positive samples, $\mathscr{S}_N$ the set of all actually negative samples. $|\mathscr{S}_P|$ and $|\mathscr{S}_N|$ refer to the number of elements in the $\mathscr{S}_P$ and $\mathscr{S}_N$ set, respectively. $\mathscr{P} = pred(\mathscr{S})$ refers the set of all predictions, $\mathscr{P}_P$ and $\mathscr{P}_N$ the set of all predictions for all actually positive ($_P$) and negative ($_N$) predictions, respectively. For each prediction $p \in \mathscr{P}$ the following holds $0 \leq p \leq 1$.

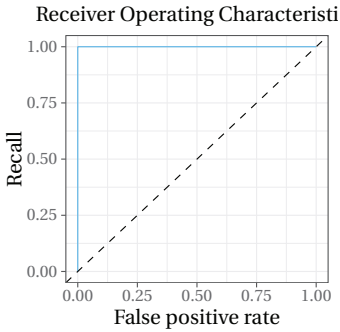
Figure 2.2 shows a perfect classification of a data set with an imbalance ratio of $1:1000$ holding $|\mathscr{S}_P| = 100$ samples. Since $\min(\mathscr{P}_P) > \max(\mathscr{P}_N)$, the ROC curve passes through the point $(0,1)$ and the PRC passes the point $(1,1)$.

In contrast, Figure 2.3 shows a very poor model based on an artificial data set with an imbalance ratio of $1:1000$. In this case, the $|\mathscr{S}_P| = 100$ are uniformly distributed in the top 0.1% percentile of all predictions $\mathscr{P}$. In the automotive context this would e.g. yield a component being replaced in 1000 cars while the component was only broken in 100 of them. The ROC curve does not reflect the definitely worse model performance compared to the previous example (Figure 2.3a); the PRC unambiguously reflects this (Figure 2.3a).

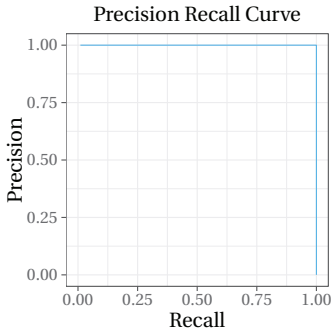
In the next data set, the imbalance ratio is reduced to $1:100$ (Figure 2.4). The data set now holds $N = 100 + 10000 = 10100$ samples. The $|\mathscr{S}_P| = 100$ actually positive samples remain uniformly distributed in the top 10% percentile of all predictions $\mathscr{P}$. The ROC curve starts to reflect the poor model performance (Section 2.3). The PRC continues to reflect the model performance the same way as before (where the imbalance was higher).

Reducing the imbalance ratio further to $1:10$ (Figure 2.5), the data set now holds $N = 100+1000 = 1100$ samples. The $|\mathscr{S}_P| = 100$ are now distributed among the top 91% percentile of all predictions. ROC curve now indicates an almost coin-flip model (identified by the

dashed line with a slope of one).

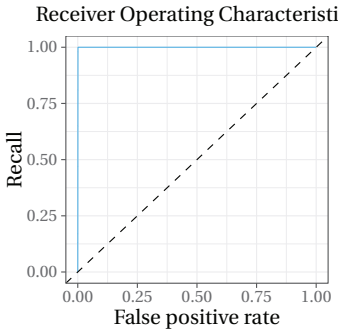


(a) ROC curve.

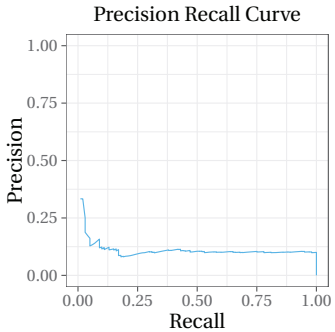


(b) PRC curve.

Figure 2.2: Model that identifies the 100 real positives samples without error. Imbalance 1 : 1000.



(a) ROC curve.

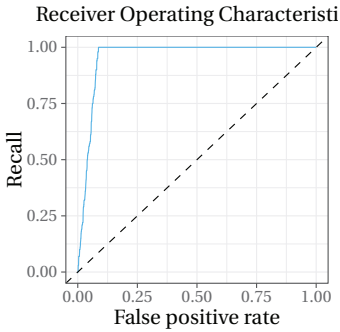


(b) PRC curve.

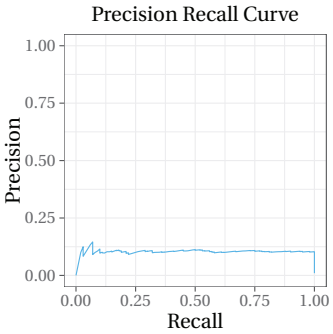
Figure 2.3: Poor model. Imbalance 1 : 1000.

In addition to the just exemplarily justified unsuitability, a model that dominates another model in terms of auROC does not necessarily dominate in the PRC space. Conversely, a model that dominates in the PRC space will always dominate in the ROC space [101, 57, 36].

Thus, the auPRC will be used as *primary* metric in this work. To ease the comparison with other research, it will be accompanied by the auROC and the F_1 score.

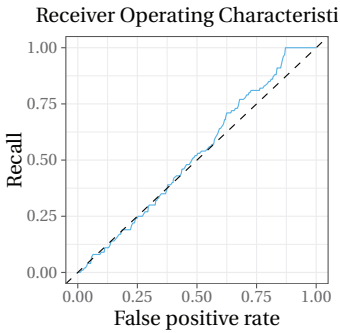


(a) ROC curve.

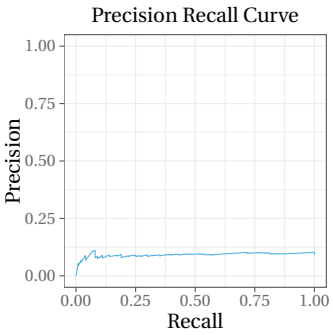


(b) PRC curve.

Figure 2.4: Worse than poor model. Imbalance 1 : 100.



(a) ROC curve.



(b) PRC curve.

Figure 2.5: Catastrophic model. Imbalance 1 : 10.

Chapter 3

Feature Selection

One would intuitively assume that not all of the features a data set offers are in general helpful or required to train and create working machine learning models. This suggests to isolate and select important influencing factors (encoded in features). If the data set offers more than one potential outcome of interest, multiple models may be required. In this case, relevant features shall be selected specifically for each model, respectivly. This is not only expected to speed up the training process, but also to improve the generalization of the model and to benefically influence the interpretability of the trained models by human experts, which is not only helpful in the automotive context.

The overall goal of this thesis, enabling highly automated data-driven automotive diagnostics, requires thousands of models being built – potentially on a daily basis to account for data updates using a growing amount of samples and features. In order to speed up model training and to ensure the interpretability of the trained models, the selection of relevant features from the multi-thousand dimensional feature space is indispensable. The implementation of a feature selection pipeline becomes additionally challenging in the given automotive context, where the features vary in their sparsity, noise level, datatype, and distribution.

Often, as the following results show, no more than 25 features are needed for modeling. Therefore, the pipeline must be able to reduce the high-dimensional feature space by up to 99.5%, taking into account the following framework conditions:

To ensure interpretability of the selected features and created models by human experts, this chapter focuses on feature subset selection techniques only [100]. This refers to the set of algorithms, that only choose the most promising features from the existing ones but do not

transform features into another (meta-)feature space. Techniques such as Principal Component Analysis (PCA) [39], deep learning using auto encoders [27, 2], or feature selection using compressed features based on information theory [137] are therefore, not part of this research.

The pipeline proposed in the following consists of three layers: a feature preparation layer, a filter layer that significantly reduces the feature space at low computational cost based on entropy and other statistical measures, and a wrapper layer that selects the final feature set for training based on simple models. Finally, high-performant LR models have been trained to generate the metrics used for evaluation.

This chapter is structured as follows: First, state of the art data-driven diagnostic techniques (Section 3.1.1) and different feature selection techniques (Section 3.1.2, Section 3.1.3) are presented. Then, given the aforementioned conditions, research questions are formulated (Section 3.2). Afterwards, the data sets used for evaluation are discussed (Section 3.3). Finally, a feature selection pipeline is proposed (Section 3.4) and evaluated (Section 3.5). The results are summarized in Section 3.6.

Preliminary results of the work presented in this chapter have been published in [109], where a cost-benefit survey of various feature selection algorithms was deducted. The code to run the experiments described in this chapter was entirely implemented using `R`.

3.1 State of the Art

3.1.1 Diagnostics

Most known data-driven approaches from automotive contexts based on machine learning are *specific* in the sense that they only model a single type of component. Examples include compressors failures modeled by RFs [93], pump bearings failure prediction based on Artificial Neural Networks (ANNs) [123], combustion engines faults modeled by ANNs [1] or wavelet networks to predict distinct (malicious) operation modes [122], turbochargers failures modeled ANNs [141], and lithium ion batteries failures modeled by Gaussian processes [95].

On the other hand, *generic* approaches are not tailored towards a specific component or problem and therefore able to predict different types of faults. The generic approach proposed by Azarian et al. [4] requires high manual effort to create “suspicious links”, pointing

from features (DTCs) to targets (parts).

Another generic approach is proposed by Müller et al. [86]. Their model is based on

- DTCs: Based on the self-diagnostic capabilities of ECUs, e.g., discrepancies between measured and calculated values are flagged as a DTC. This relies on the self-diagnostic capabilities of the ECU itself.
- encoded customer and workshop staff perception,
- software version numbers of ECUs, and
- part numbers of switched parts.

3.1.2 Filter Measures

Filter measures are the first type of feature assessment [79] techniques discussed in the following. Filters are fast, scalable, and independent from the classifier [100]. The following sections provide an overview of the used filter measures and will explain the basic concept. The used filter measures were selected based on a cost/benefit survey by Schlegel and Sick [109] and are aligned with a recent study by Huertas and Juárez-Ramírez [62] who evaluated several filter measures for homogeneous data sets. These are: Mutual Information (MI) (Section 3.1.2), the Gini coefficient (Section 3.1.2), Relief (Section 3.1.2), the Chi-Squared test (Section 3.1.2), and correlation (Section 3.1.2). Thus, a diverse range of mathematical concepts such as distribution similarity, unbalancedness of distributions, correlation, etc. is covered. The score was calculated using solely training samples based on the feature (unsupervised), or the feature and the label if required by the corresponding algorithm (supervised) [79]. A features' importance will be denoted by $\mathscr{I}$ in the following.

Mutual Information

Based on the information theory introduced by Shannon [114] the helpfulness of feature can be assessed by measuring the additional information gained by considering it. A helpful concept to do so is the Mutual Information (MI) [39, 8], which measures "the reduction in uncertainty about one variable due to the knowledge of the other variable" [39]. It is defined as for a given feature X_p [39, 8]:

$$\mathbb{I}[Y, X_p] = \mathbb{H}[Y] - \mathbb{H}[Y|X_p], \tag{3.1}$$

with $\mathbb{H}[Y]$ being the entropy of label Y and $\mathbb{H}[Y|X_p]$ being the conditional entropy. Using Equation (3.1), the MI resulting from a feature X_p can now be defined by the decrease of unpredictability of the label Y when the feature X_p is known [87, 39, 8]. To calculate the MI, two other measures need to be formally defined: The entropy and the conditional entropy.

The *entropy* $\mathbb{H}$ is a measure for the randomness or unpredictability of discrete variables (or in this case label Y) [87, 39]:

$$\mathbb{H}[Y] \equiv -\sum_{k=1}^{K} p(Y=k)\log(p(Y=k)), \tag{3.2}$$

with $K = |\mathbb{D}_Y|$ referring the number of states, $p(Y=k)$ refers to the probability of label Y taking the value of $k \in \mathbb{D}_Y$. The entropy according to Equation (3.2) reaches its maximum for an uniformly distributed random variable where $p(x=k) = 1/K$ and equals to zero for constant variables. This can be extended to the *differential* entropy $\mathbb{H}$ for a continuous X_{cont} variable

$$\mathbb{H}[X_{cont}] \equiv -\int_x p(x)\log(p(x))\mathrm{d}x, \tag{3.3}$$

with $p(x)$ being the distribution over the continuous variable X_{cont} (note: an integral replaced the sum). However, this is not relevant to the remainder of this section, since the function used to calculate the entropy[1] [99] internally discretizes continuous features[2].

The *conditional entropy* used in Equation (3.1) is defined as follows [8] for discrete variables:

[1] The `information.gain()` function from the `FSelector` package available in `R` was used.
[2] Details are available in the `FSelector` source code, see the `discretize.R` file.

$$\mathbb{H}[Y|X] = \mathbb{H}[X_p, Y] - \mathbb{H}[X_p] \tag{3.4}$$

$$= -\sum_{x \in \mathbb{D}_{X_p}} \sum_{y \in \mathbb{D}_Y} p(X_p = x, Y = y) \log(p(X_p = x, Y = y)) - \mathbb{H}[X_p] \tag{3.5}$$

$$= -\sum_{x \in \mathbb{D}_{X_p}} \sum_{y \in \mathbb{D}_Y} p(X_p = x, Y = y) \log(p(X_p = x, Y = y)) + \tag{3.6}$$

$$\sum_{x \in \mathbb{D}_{X_p}} p(X_p = x) \log(p(X_p = x)) \tag{3.7}$$

$$= -\sum_{x \in \mathbb{D}_{X_p}} \sum_{y \in \mathbb{D}_Y} p(X_p = x, Y = y) \log(p(X_p = x, Y = y)) + \tag{3.8}$$

$$\sum_{x \in \mathbb{D}_{X_p}} \sum_{y \in \mathbb{D}_Y} p(X_p = x, Y = y) \log(p(X_p = x)) \tag{3.9}$$

$$= \sum_{x \in \mathbb{D}_{X_p}} \sum_{y \in \mathbb{D}_Y} p(X_p = x, Y = y) \left(\log(p(X_p = x) - \log(p(X_p = x, Y = y)\right) \tag{3.10}$$

$$= \sum_{x \in \mathbb{D}_{X_p}, y \in \mathbb{D}_Y} p(X_p = x, Y = y) \log\left(\frac{p(X_p = x)}{p(X_p = x, Y = y)}\right). \tag{3.11}$$

The mutual information according to Equation (3.1) can now be calculated using Equation (3.2) and Equation (3.11), with

- $p(X_p = x)$ referring the prior probability of a discrete feature X_p being equal to $x \in \mathbb{D}_{X_p}$ when being drawn randomly,
- $p(Y = y|X_p = x)$ is the conditional probability of a random sample belonging to the discrete class $y \in \mathbb{D}_Y$, given that the discrete feature X_p being equal to $x \in \mathbb{D}_{X_p}$,
- $p(Y = y)$ is the prior probability of a random sample belonging to the discrete class $y \in \mathbb{D}_Y$ and
- $p(X_p = x, Y = y)$ is the joint probability of discrete feature X_p being equal to $x \in \mathbb{D}_{X_p}$ and discrete class Y being equal to $y \in \mathbb{D}_Y$ when being drawn randomly.

The following example shall explain how the probability for a random, discrete feature X_p is estimated. If the feature is continuous, discretization is performed beforehand using the technique by Fayyad and Irani [42][3]. This technique recursively selects splits that minimize the entropy for each resulting partition until the Minimum Description Length (MDL) [97]

[3] This technique is used by the `Discretize` function provided by the `RWeka` package which was used.

or an optimal number of intervals is achieved [84]. Suppose the feature X_p has 4 levels (if necessary, after discretization): $\{a, b, c, d\}$ for which the respective probabilities are given by $\left(\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{2}{5}\right)$ based on the distribution of the feature. This yields the prior probability of, e.g., $p(X = c) = \frac{1}{5}$.

MI can take values in the range of $[0, \mathbb{H}(Y)]$ for discrete (or discretized) features. The higher the value, the more information is gained from feature X_p. This filter measure requires both, the feature value and the label and assesses each feature individually.

Gini Coefficient

Originally proposed by Sen [113] to measure the inequality of incomes, the "Gini coefficient" (not to be confused with the Gini index which is a purity measure similar to entropy, see Section 2.2.1) for a feature X_p is given by [38, 35]

$$Gini(X_p) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} |x_{i,p} - x_{j,p}|}{2N^2 \mu(X_p)}, \tag{3.12}$$

with N being the number of samples (values) available, $\mu(X_P)$ being the mean of the feature X_P, and $x_{i,p}$ being the value of ith sample from the pth feature X_p. If the feature X is ordered, which is ensured by the used implementation[4], the following formula can be used [35]

$$Gini(X_p) = \frac{\sum_{j=1}^{N} (2j - N - 1) x_{j,p}}{N^2 \mu(X_p)}, \tag{3.13}$$

which decreases the run-time complexity to from $\mathcal{O}(n^2)$ (two, nested for loops) to $\mathcal{O}(n \cdot \log(n) + n) = \mathcal{O}(n \cdot \log(n))$ (sorting and one loop).

Due to the fact that its roots lie in economics, the Gini coefficient ranges from zero (a feature X_p is constant or economically put "all individuals earn the same") to one (a feature is always zero except for one sample, "an infinite population in which every individual except one has no income" [35]).

Listing 3.1 shows how the Gini coefficient for a single feature X_p is calculated using zero-indexed pseudo code: First, required information is gathered in lines 1 through 3. After sorting the feature vector X_p in line 4, the loop in line section 3.1.2 calculates the sum according to Equation (3.13). This process is repeated for all features $X_p \in \mathbb{D}_X p$. The label Y

[4] The function `ineq(..., type="Gini")` provided by the `ineq` R package was used.

Table 3.1: Examples for different feature values and their corresponding Gini coefficients.

$Gini(X_p)$	feature X_p
0.75	$(0,0,0,1)^T$
0.8	$(0,0,0,0,1)^T$
0.8	$(0,0,0,0,10)^T$
0.2	$(1,1,1,1,0)^T$
0.4	$(0,1,2,3,4)^T$

is not required for this filter measure.

```
mu := mean(X)
N := length(X)
temp_sum := 0
X := sort(X)
for j:= 1 to N do
  temp_sum := temp_sum + (2j - N - 1) * X[j]
gini = temp_sum/(N*N*mu)
```

Listing 3.1: Pseudo Code to calculate the Gini coefficient [113].

The Gini coefficient can yield counter intuitive results, as shown in Table 3.1. Suppose, feature X is indicating the income distribution, as this this was the original purpose of use for the Gini coefficient. In the example, the vector (or feature) $(0,0,0,0,1)^T$ refers five people, where only one of them earns all the money. This yields a Gini coefficient of 0.8 referring an "unfair" income distribution (second row in Table 3.1). Considering the opposite case ($(1,1,1,1,0)^T$, fourth row in Table 3.1), four persons exactly earn the same (1), and one person has no income (0). This is less "unfair", since the income is more equally distributed across a higher number of people. The Gini coefficient reflects this: In the second scenario the lower Gini coefficient indicates less unfairness.

However, these results, which are comprehensible from an macroeconomic point of view, are *not* comprehensible from the point of view of information theory [114]: E.g., both income distributions would yield the same entropy: $H[(1,4)] = H[(4,1)] \approx 0.5$.

Relief

Unlike all aforementioned filter measures which assume that the features are independent (each feature is assessed in isolation), the Relief filter originally proposed by Kira and Rendell [70] and enhanced by Kononenko [71] is not based on this assumption. The key idea

of this supervised filter measure is to rate features based on their capabilities to distinguish samples that are near to each other. Thus, the feature values and class labels are required.

```
1  set all weights w[0:P] := 0.0
2  for i:= 1 to m do
3    randomly select a sample s_rnd
4    find nearest hit s_hit and nearest miss s_miss
5    for j := 0 to P do
6      w[j] := w[j] - diff(j, s_rnd, s_hit) / m +
7        diff(j, s_rnd, s_miss) / m
```

Listing 3.2: Pseudo Code of the Relief algorithm [70, 71].

The original algorithm [70] works as shown in Listing 3.2: First, all feature importances are initialized $w(X_p) = 0$ for all features $X_p \in \mathscr{X}$ (with P features total) as shown in line 1. The variable $m \leq N$ can be used to subsample the data set and thus decrease the run-time complexity. This thesis used $m = 5^5$. Then, in line 3, a random sample is selected from the set of all samples ($s_{rnd} \in \mathscr{S}$). For the randomly selected sample, the nearest hit s_{hit} with same class label and nearest miss s_{miss} with a different class label are selected (line 4). Using the nearest hit and miss, all weights w of all features $\mathscr{X}$ are updated according to the following rule (line 7):

$$w(X_p) := w(X_p) - \frac{diff(j, s_{rnd}, s_{hit})}{m} + \frac{diff(j, s_{rnd}, s_{miss})}{m}. \tag{3.14}$$

The function $diff(j, s1, s2)$ calculates the difference between the values of jth feature for samples $s1$ and $s2$. For discrete features, $diff()$ returns 0 if they are equal, and 1 else. For continuous attributes, $diff()$ returns the actual normalized difference (ranging from $[0, 1]$):

$$diff(j, s1, s2) = \frac{|value(j, s_1) - value(j, s_2)|}{\max(X_p) - \min(X_p)}, \tag{3.15}$$

with $value(j, s)$ returning the value of feature j from sample s. The loop in line 2 is repeated $m - 1$ more times.

The intuition behind the weight update formula (Equation (3.14)) is that the feature is more helpful if the closest sample of the same class is very close, while the closest sample of a different class is very distant, since this eases the identification of a decision boundary.

The enhancements proposed by Kononenko [71, 98] that have been used for this thesis are the following:

First, Kononenko extends the algorithm to take $k > 1$ nearest hits and misses into con-

[5] This was the default value of the `relief()` function from the `FSelector` package available in R.

sideration instead of just a single one. This increases the reliability in noisy scenarios (in the following $k = 5$ was used). This is achieved by averaging the contribution to the weight update in line 7 in Listing 3.2.

Second, the *diff*() function was enhanced to deal with missing feature values. The best results were achieved by using Equation (3.16) if one sample misses a value (e.g., s_1) and using the *diff*() calculation according to Equation (3.17) if both samples have missing values. Both versions rely on the conditional probabilities p that are approximated with the relative frequencies from the training set. The sum in Equation (3.17) denotes an iteration over all possible values v of attribute j.

$$diff(j, s1, s2) = 1 - p\left(value(j, s2)|class(s1)\right) \tag{3.16}$$

$$diff(j, s1, s2) = 1 - \sum_{v}^{values(X_p)} \left((p\,(v|class(s1)) \cdot (p\,(v|class(s2))\right) \tag{3.17}$$

Third, a strategy to deal with multi-class problems was introduced: Instead of picking *one* near miss from any different class, one near miss s_{miss} for *each* different class is picked and their contribution to the feature weight averaged. However, the third improvement is, unlike the first two, not relevant for this thesis since this chapter deals with two-class classification only.

χ^2 Test

The Chi-Squared (χ^2) test [81] is used here to determine if two random categorical variables are independent from each other. This is the null hypothesis. In the feature selection context this translates to checking the independence of feature $X \in \mathcal{X}$ and a label Y. How the χ^2 metric is calculated, is described below. As the MI measure described above, the used implementation[6] internally discretizes continuous features. The number of bins used for discretization might affect the ranking of the feature and must be further investigated before transferring the techniques described in this chapter to practical use.

To ease following along, an example based on a categorical feature X "Exercise" with three different levels and a target class Y "Pulse" (heart rate) with two levels and $N = 192$ samples is used (Table 3.2). Now:

[6] To calculate the χ^2 ranking, the `chi.squared()` method from the `FSelector` package [99] was used.

Exercise	Pulse
Freq	high
None	low
Freq	low
Some	low
Some	high
Some	low
Freq	high
Freq	high
...	...

Table 3.2: Sample data.

	high	low
Freq	38	57
None	9	8
Some	49	31

Table 3.3: Contingency table.

1. Transform the data set into a contingency table (Table 3.3), yielding the total number of instances for each feature-target pair, $Q_{x,y}$. High dimensional feature spaces can cause many zero entries in the contingency table. This, however, does not cause any issues, since $Q_{x,y}$ only appears as numerator in the following equations.

2. Calculate the expected occurrence $E_{x,y}$ for all variable pairs (Table 3.4). This is performed under the assumption of the null hypothesis ("feature and label are independent"):

$$E_{x,y} = N \cdot P(X = x) \cdot P(Y = y), \tag{3.18}$$

with N being the total number of samples. In our example, this would e.g. yield $E_{X=Freq,Y=high} = 192 \cdot \frac{38+57}{192} \cdot \frac{38+9+42}{192} = 47.5$

3. The final step is to calculate the Chi-square-points (Table 3.5) and to sum them up (yielding $\chi^2 = 7.9$ in the example):

$$\chi^2 = \sum_x \sum_y \frac{(Q_{x,y} - E_{x,y})^2}{E_{x,y}}. \tag{3.19}$$

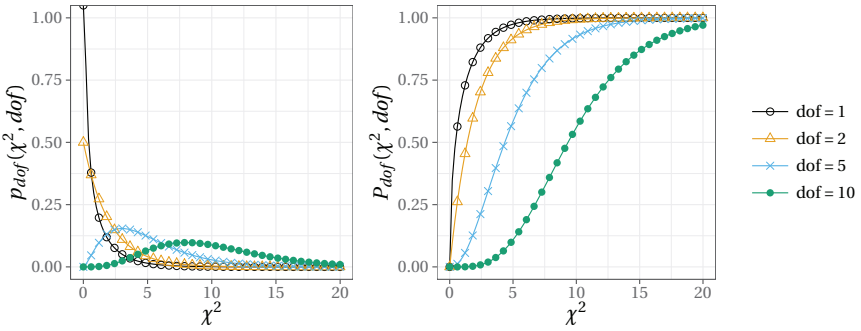
Based on the χ^2 calculated above, p can be calculated. The latter refers the probabil-

	high	low
frequent	47.50	47.50
no	8.50	8.50
some	40	40

Table 3.4: Expected frequencies.

	high	low
frequent	1.90	1.90
no	0.03	0.03
some	2.03	2.03

Table 3.5: Chi-square points.



(a) PDF of χ^2 distribution.

(b) CDF of the χ^2 distribution.

Figure 3.1: The χ^2 distribution.

ity of obtaining results as unusual or extreme as the observed results [24]. To acquire the corresponding p from the χ^2, the χ^2 distribution (or a lookup table) is required. The χ^2 distribution is created by summing *dof* squares of independent, normally distributed random variables. *dof* refers the "degrees of freedom" which is defined as $dof = (|\mathbb{D}_{X_p}| - 1) \cdot (|\mathbb{D}_Y| - 1)$ for two categorical random variables, with $|\mathbb{D}|$ referring the number of distinct levels of feature X and label Y (in our example $dof = 2$). The χ^2 distribution is depicted in Figure 3.1a based on *dof*. Since the χ^2 distribution results from squaring a normal distribution, it ranges from 0 to infinity. The corresponding $p - value$ is determined based on the CDF according the following formula

$$p(\chi^2) = 1 - P_{dof}(\chi^2, dof), \tag{3.20}$$

yielding $p = 0.01917$ in the example, see the orange line in Figure 3.1b where the cumulative density function is plotted. A value of χ^2 close to zero (corresponding to a $p - value$ close to 1) tells us that it is very unlikely that the variables are completely independent. Given the example and a significance level of 0.05 one would reject our null-hypothesis of "working out has no effect on the pulse". Yet, this does not mean, that they are strongly dependent.

A similar measure is calculated based on the χ^2 measure that indicates the strength of the association is the Cramer's V [33] coefficient which is defined as

$$\Phi_C = \sqrt{\frac{\chi^2}{N(k-1)}}, \tag{3.21}$$

with N being the total number of samples and k being the value of rows or columns in the

contingency table (whichever is smaller). This normalized value allows to draw conclusions whether the label and feature depend on each other (Φ_C close to 1) or are completely independent ($\Phi_C = 0$). This value will be used in the following to perform χ^2 feature selection. The example yields $\Phi_C = 0.203$. This measure requires both, the label and the target. Discretization of continuous features is performed as layed out in the "Mutual Information" section above using the method by Fayyad and Irani [42].

Correlation

The term "correlation" is used for a range of measures, quantifying the dependence between two random variables. The most commonly used correlation measure is the Pearson coefficient of linear correlation (r), which measures the linear dependence between random variables, which is defined as

$$r(X_p, Y) = \frac{\sum_{n=1}^{N}(x_{(n,p)} - \mu(X_p))(y_i - \mu(Y))}{\sqrt{\sum_{n=1}^{N}(x_{(n,p)} - \mu(X_p))^2}\sqrt{\sum_{n=1}^{N}(y_n - \mu(Y))^2}}, \tag{3.22}$$

with $\mu(X_p)$ referring to the mean of the pth feature, $\mu(Y)$ referring to the mean of feature or target Y, and x and y being two continuous variables. However, r_{X_pY} requires the random variables to be linearly dependent and normally distributed. Especially in generic machine learning pipelines with thousands of features, this can not be guaranteed.

The Spearman coefficient of rank correlation ρ [61] relaxes these requirements by measuring the linear correlation for the ranks of the original observations. This relaxes the linear requirement, since the correlation is no longer dependent on the actual value but solely on the order (or monotonic correlation).

To achieve this, the two variables are first sorted by their values $x_{(n,p)}$ (y_n) and then assigned rank in ascending order $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(N)}$ ($y_{(1)} \leq y_{(2)} \leq \cdots \leq y_{(N)}$), with N being the number of samples and $x_{(1)}$ ($y_{(1)}$) referring to the lowest value of X_p (Y). If multiple ranks refer the same value, these ranks are averaged in a final step to yield the ranks $R_{n,p}$ (S_n) used in the following calculations. E.g., a feature $X_p = (0.4, 7, 7, 0.42)^{\mathrm{T}}$ yields the ranks $R_p = (1, 3.5, 3.5, 2)^{\mathrm{T}}$. Thus, the ranks $R_{n,p}, S_n$ are within the interval $[1, N]$. The Spearman coefficient is then defined by

$$\rho(X_p, Y) = 1 - \frac{6\sum_{n=1}^{N}(R_{n,p} - S_n)^2}{N(N^2 - 1)}. \tag{3.23}$$

Both coefficients (Spearman and Pearson) range from $[-1, 1]$, with $+1/-1$ indicating a strong positive or negative and 0 indicating no correlation. Also, both require ordinal variables that enable the calculation of a mean or a performing a division. Thus, categorical variables need to be transformed e.g. by using a one-of-k scheme ("one-hot-encoding", see Section 3.4.1).

In a feature selection context, $\rho(X_p, Y)$ can be used to measure the correlation between all feature-label pairs. The correlation was calculated for each pair, respectively.

3.1.3 Wrapper and Embedded Measures

While the aforementioned filter techniques require no model to be built, wrappers determine feature importances based on heuristics that use information from trained models. One advantage of wrappers is that they take greater account of more complex interdependencies between features. On the other hand, excessive training (e.g. selecting the best parameters using cross-validation) can have a high computational complexity, as shown later. In addition, the results may not be transferable to other model types. Three different implementations, namely RF (Section 3.1.3), k-NN (Section 3.1.3), and LR (Section 3.1.3), that are briefly described and extensively evaluated in the following. For all wrapper measures, both the feature and the target are required.

Random Forest Feature Importances

Having a "forest" of random decision trees in mind, the intuitive definition of a features' importance should be related to the number of occurrences of the feature under consideration in the forest, and the average position in the trees (the closer to the trunk, the more important). This intuition can be used to formulate an RF importance for a feature X_p [44] as

$$\mathscr{I}_{RF}(X_p) = \frac{1}{|d|} \sum_{d \in D_{X_p}} \Delta(d, X_p), \tag{3.24}$$

where D_{X_p} are all sub-data sets that evolved from splitting by X_p during training, $|d|$ is the number of samples in the respective sub-data set, and $\Delta(d, X_p)$ is the gain of splitting the sub-data set d by feature X_p (see Equation (2.3), it should be noted that more than two subtrees are possible).

Feature Importances from L_1 Regularized Logistic Regression

L_1 regularized logistic regression (LR) has been favored over L_2 for two reasons:

1. Results from Ng [88] suggest that L_2 regularized LR is "rotationally invariant"[7] and also needs more samples from the minority class compared to L_1 to perform equally. "Rotational invariance" of a stochastic learning algorithm (which the LR is) refers the circumstance that the predictions have the same distribution if the training set $\mathscr{S}$ and all remaining samples are rotated by the same rotational matrix $\mathscr{M} = \{M \in \mathbb{N}^{d\times d} \mid MM^{\mathrm{T}} = M^{\mathrm{T}}M = I, |M| = 1\}$.

2. L_1 outperforms L_2 regularization when a lot of irrelevant features are present [88] since L_1 results into very low (0) feature weights for important features [8].

The feature importances are represented by the vector of absolute weights $\mathbf{w} = (w_1, w_2, \ldots, w_P)^{\mathrm{T}}$ (recall Section 2.2.2 for more information on the training process) of the trained model:

$$\mathscr{I}_{LR}(X_p) = |w_p|. \tag{3.25}$$

Since one step in when predicting using LR is the multiplication of the model weights w with the presented sample s, it becomes obvious that a feature X_p ranging from 0 to $1 \cdot 10^6$ would be modeled with smaller model weight w_i, compared to an evenly important feature that ranges from 0 to $1 \cdot 10^{-3}$. Thus, for L_1 LR feature selection to work properly, feature normalization is required. A normalized feature has zero mean and a standard deviation of $\sigma = 1$.

k-NN greedy approach

Due to the high dimensional feature space, a tailored version of a k-NN wrapper inspired by Bolon-Canedo et al. [15] and Breker et al. [23] was implemented: For each feature, one k-NN was trained separately. A range of different values from for k was evaluated: $k = [2,3,4,5,10,15,20]$[8]. The best achieved accuracy of every feature was used to rank the latter. The benefit of this naive approach is that only P_{kNN} (P_{kNN} is referring the number of features evaluated by the k-NN wrapper, see Equation (2.1)) k-NNs have to be evaluated.

[7] For a definition of *rotational invariance* and the proof itself see Ng [88].
[8] In a two-class classification problem, uneven numbers should be preferred to eliminate the risk of ties.

Strategies by which the next feature is selected based on the accuracy gain it adds to the k-NN when being used in addition to the already selected features were not investigated because being to computationally expensive. The procedure just described would require $P_{kNN}!$ k-NNs to be evaluated.

3.2 Research Demand

The implementation, systematic evaluation and optimization of a highly autonomous feature selection pipeline for high dimensional, heterogeneous feature spaces causes the following research questions, elucidated in the remainder of the chapter.

First, the basic architecture of the pipeline needs to be defined (Section 3.4): How can available methods and algorithms be orchestrated in the most promising way in a pipeline, defining the basic structure of the pipeline? Which characteristics have to be obeyed when preparing the features for the pipeline (Section 3.4.1)? Which filter measures (Section 3.4.2) and wrapper algorithms (Section 3.4.3) should be evaluated? Which kind of model is suitable for generating a pipeline performance metric, used for evaluation and optimization (Section 3.4.4)? And: Which hyperparameters result from this to optimize the pipeline (Section 3.4.5 based on Section 2.3)?

Second, preliminary questions are answered: Is it possible, given the available features, to create models that produce results above a given performance threshold at all? Also, the automotive feature space as well as potential model outputs (targets such as switched parts) are evaluated (Section 3.5.1) in detail.

Third, a detailed evaluation of the different pipeline stages is carried out: Which filter algorithms rank features high in order to create highly performing models (Section 3.5.2)? Furthermore, a variety of wrapper algorithms (Section 3.5.3) are compared.

Fourth, a multidimensional optimization problem needs to be solved: How to identify the optimal hyperparameters and algorithms for the pipeline and what are the most valuable features (Section 3.5)? Thousands of results are analyzed to identify the optimum pipeline parameters. Finally, the question, how the proposed pipeline performs on data sets for which it has not been optimized for, is addressed (Section 3.5.5).

3.3 Data Sets

In this section the data sets are presented that have been used to optimize and evaluate the performance of the feature selection pipeline. Although the pipeline was developed for the automotive context (diagnostics data set in Section 3.3.1), it was also evaluated on publicly available data sets to ensure comparability.

3.3.1 Diagnostic Automotive Data

The *automotive* data set was collected from BMW Hybrid cars and consists of non personal data only. In total, 250.000 observations, each with 5000 features were used to predict over 3000 potential targets. These include the parts that were replaced in the workshop, taken counter actions (e.g., installing a software-update on an ECU), and Diagnostic Codes (DCs). DCs are hash-like values identifying the final result of a workshop repair. These should not be confused with DTCs that can provide indications of potential errors based on the self-diagnostic capabilities of ECUs. A detailed evaluation of the different feature types is given in Section 3.5.1.

Aside from being very high dimensional, the automotive data set poses another challenge for feature selection that will be tackled in the following: The heterogeneousness, referring the fact that each of the aforementioned feature types vary in terms of their datatype, sparsity, and information content (as we will see in Section 3.5.1). E.g., EEs is always Boolean typed (a customer either ordered leather seats or not), CPs are mostly categorical values (a car is only available with a certain set of unique wheel rims), and most of the remaining features are discrete/ continuous or nominal after preprocessing (e.g., the mileage of the read-out, the value of a specific MV, etc.). Additionally, the sparsity differs between the different feature groups: DTCs have a sparsity of 99% (since the occurence of errors is rare on average), whereas MVs only have a sparsity of 65% (measurements are always present, but still hold the value “zero” for a considerable amount of samples), and RO less than 1%. More details on the various feature types available in the automotive context are given in Section 1.2.

Also, many potential model targets only appear at a very low rate (e.g., a BMW-engine part that was only switched a couple of times in the whole data set). This imbalance must also be taken into account during model training, corresponding procedures are presented in Chapter 4. To obtain meaningful results, the minimum number of samples, where a target

was true (“part number 117 was switched”) for the pipeline to select features and create a model was set to five. This minimum is only exceeded by 69% of the parts, 56% of the taken actions, and 40% of the diagnostic codes. Only targets that matched this criterion (e.g., a part that was switched at least five times) have been processed by the pipeline.

In addition to the features used by Müller et al. [86] as described in Section 3.1.1, the selection pipeline proposed in the following is developed and evaluated on the following feature types which are offered by the *automotive* data set (see Section 1.2):

- Long time Measurement Values (MVs): MVs capture slowly changing car parameters, these include but are not limited to: Slowly drifting parameters of proportional-integral-deriva-tive controllers integrated into the engine control, the number and strength of misfires, the total number of hours driven, etc.

- Environmental conditions (ECs) from Diagnostic Trouble Codes (DTCs): Every time a DTC is flagged, values describing the engine or car-state such as the engine rounds per minute or the vehicle speed and certain temperatures at the time of error occurrence are recorded. The idea behind is to ease the subsequent analysis.

- Optional Extra Equipment (EE): Customers often make use of the possibility of adapting the car to their personal needs when ordering by adding optional extras. Examples are aerodynamic kits, rims, and leather seats.

- Basic Car Parameters (CPs): This information includes, e.g., the engine horse power, steering type, the type of gearbox (automatic or manual), etc.

- Readout data (RO): A readout contains, e.g., the current odometer value, software version information, the readout date, etc.

3.3.2 Publicly Available Data Sets

Also, two publicly available classification data sets with a high dimensional feature space were selected.

The *Golub* data set consists of 72 samples, each representing one patient, with 7129 features each. The features in this case are gene expressions. The target is to classify which of patients indicate type one leukemia (acute lymphoblastic leukemia, 47 patients) or type

two (acute myeloid leukemia, 25 patients). This data set is linearly separable and can not be considered "imbalanced" (the ratio is 0.532 : 1).

Secom contains 1567 observations from a semi-conductor manufacturing process. The 591 features represent sensor values used for monitoring the process. The goal in this case is to classify a "pass" of the fabricated wafer, which is recorded 1463 times. A "fail" is recorded 104 times, yielding an imbalance of 0.071 : 1.

3.4 Proposed Solution

The proposed pipeline for feature selection is depicted in Figure 3.2. The vertical axis represents the number of features being processed. As expected, during processing from the left to the right this number is reduced throughout the pipeline yielding the funnel like shape of the pipeline. The latter consists of a preparation layer (Section 3.4.1), two feature selection layers, and a model building layer (Section 3.4.4). All layers were implemented in `R`. The feature selection layers (orange) include a filter layer (Section 3.4.2) and a wrapper layer (Section 3.4.3).

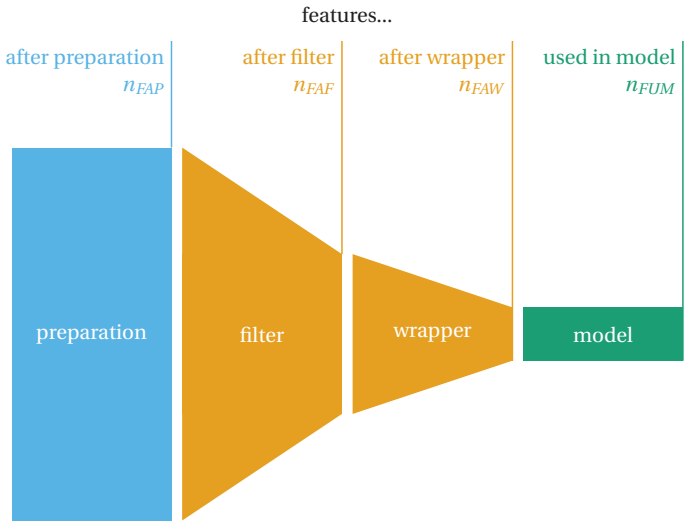


Figure 3.2: Overview of the feature selection pipeline.

3.4.1 Preparation Layer

This and all subsequent layers must meet the generic requirements. This means that since a large number of models are to be created *parameter free*, no kind of manual influence must be required after implementation.

The preparation includes:

- The removal of constant features,
- the removal of $n-1$ features from a group of n strongly linearly correlated features,
- encoding of textual features (e.g., the software-version of an ECU which is encoded such as "BMW3-16-13-100") into machine readable numbers,
- one-hot-encoding of nominal variables (every level is encoded in a new feature which equals to 1 if the level under consideration is present and 0 otherwise), and
- normalizing all features such that mean $\mu(X) = 0$ and standard deviation $\sigma(X) = 1$ for all $X \in \mathcal{X}$.

Also, an outlier filter was implemented. By default, outliers are filtered out by calculating the feature specific mean and standard deviation and considering all values that are more than $3 \cdot \sigma$ from μ away as "outliers" [76]. This, however, requires the considered feature to be normally distributed, which can not be guaranteed in the automotive context at hand. A visualization is given in Figure 3.3 where the presumably interesting values (on the very right of the distribution) would be marked counter intuitively as "outliers" since the feature is not normally distributed. To overcome this, all feature values within the first $q_{0.01}$ or last $q_{0.99}$ percentile were set to the closest, valid value. This yields two key characteristics: First, a broad range of values is kept. Second, values initialized with extreme values (e.g., 16384) do not stitch all other feature values when being normalized prior to model training, which would otherwise happen quite often in the automotive context. Also, this percentile based approach does not require any manual intervention. Outliers are filtered out before discretization of the feature is performed.

Furthermore, a variety of nominal features exists that are ordinally encoded, although there is no linear relationship. So, although being discrete per se, there exists no linear relationship. As an example consider to two dealerships referred by an integer. Although being

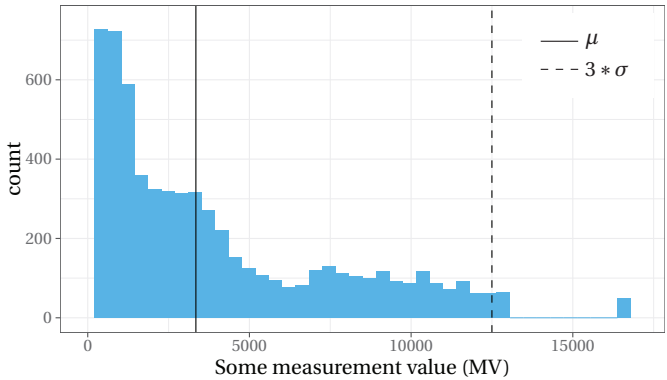


Figure 3.3: Example why standard outlier filters fail, μ is marked as solid, $3 \cdot \sigma$ as dashed line.

discrete, a dealer referenced by 21788 may reside on a different continent compared to a dealer that is only "1" unit away. A translation to latitude and longitude coordinates fixes this problem in most cases[9]. Through these coordinates a linear connection is established, making the geographical information (e.g., dusty areas or areas with low gasoline quality) accessible to the model.

After being preprocessed as described above, the data set was split into a training/validation and a test set according to the pipeline hyperparameter *SPLIT*, see Section 3.4.5 for more details.

3.4.2 Filter Layer

Filters reduce the dimensionality at low computational cost and are therefore an essential building block of the pipeline. Each of the filter algorithms presented in Section 3.1.2 was evaluated in terms of its capabilities to select helpful features and its computational complexity. Each of the aforementioned filter rankings is calculated completely "parameter-free".

The final ranking of each feature based on this layer was calculated by summing all filter measure rankings, each scaled to a range from 0 to 1. This yields a list of ranked features, sorted descendingly by the sum over all filter rankings. Alternatively the proposed pipeline offer the possibility, that top ranked feature from each filter measure are guaranteed to be passed to the following wrapper layer.

[9]The only exception is the area around the Chukchi and Bearing Sea (coordinates $(68, -180)$), but considering the BMW dealership density in this area this can be neglected.

Instead of picking features from the top of the aforementioned list, a different strategy to treat the filter rankings was evaluated: Here, the N top ranked features from every filter were prepended to the list of feature rankings. This way, a feature that received a high score from only one filter algorithm would be passed to the next pipeline stage, even if all filter measures ranked it low.

However, filter measures have two major caveats. First, they ignore feature dependencies between features since features are evaluated one by one. Second, they ignore classifier dependent feature preferences: Examples include the total number of features used to create a classifier or the polynomial degree that approximates the relationship between the features and the target (a LR will, e.g., tend to prefer features with linear influence on the target variable).

3.4.3 Wrapper Layer

Once the feature space has been "filtered" at low computational cost, the wrapper layer evaluates the remaining features in greater detail. Three concepts were implemented and evaluated: The information-gain-based ranking of a random-forest (RF) wrapper, the feature-weight-based ranking of a LR wrapper, and a k-nearest neighbors (k-NN) embedded approach as described in Section 3.1.3.

While "embedded" approaches perform the feature selection as a part of their internal learning procedure (e.g., RF and LR), "wrapper" approaches (such as the k-NN), on the other hand, are based on model types that do not rank features inherently. Instead, features are selected by evaluating the classification performance of the corresponding model on different feature subsets (e.g., k-NN) [15]. For simplification, both approaches (wrapper and embedded) will be referenced to as "wrappers" due to their similarity.

In contrast to the filter layer, where the final feature ranking was formed by taking the ranking from multiple filter algorithms into account, the wrapper algorithms were evaluated separately due to their high computational complexity.

While LR and RF wrappers consider feature dependencies, this is not the case for the greedy k-NN embedded approach: Here, a k-NNs is trained for each feature and evaluated in isolation.

3.4.4 Model Layer

The primary purpose of the model layer is to generate a metric allowing for the evaluation of the pipeline. A L_1 regularized LR based on the `LiblineaR` implementation – which serves as interface for the `LIBLINEAR C/C++` library [41] – was used.

Based on the unconstrained optimization problem given in Equation (2.11), seven different costvalues C are evaluated and optimized (ranging from 10^3 to 10^{-3}) with regard to the auPRC (see Section 2.3) using a 5-fold cross validation.

3.4.5 Hyperparameters

The pipeline described in the aforementioned sections offers a lot of hyperparameters that span up a search grid when being systematically varied:

- n_{FAF}: The number of features left after the filter layer. The parameter n_{FAF} was varied in discrete steps: $25, 50, 100, 200, 500$.
- n_{FAW}: The number of features left after the wrapper layer, also varied in discrete steps: $5, 10, 15, 25, 50, 500$.
- *NPR*: The positive to negative observations sample ratio defines how many negative samples (e.g., part was not switched) are sampled from the data set depending on the number of available, positive (e.g., part was switched) samples. Discrete values that were evaluated: 1, 3, and 5 times as many negative samples compared to the number of positives samples. A more detailled explanation of positive and negative samples is given in Section 4.3. Values $NPR \leq 1$, where fewer negative samples were used than positives, were not evaluated in this chapter. A detailed evaluation of imbalanced data sets and their effect on classification grade and performance is given in Chapter 4.
- *SPLIT*: Refers to the training/validation to test split. E.g., 0.7 refers to a split, where 70% of the samples were used for training and 30% were used for testing. Evaluated values were 0.7 and 0.8.
- *WA*: The used wrapper algorithm (Section 3.1.3) – either k-NN, RF, or LR.
- *TFA*: Select features based on the summed ranking over all filter measures (0) or prepend the one top ranked feature from each filter measure in any case (1) to the

list of descendingly important features.

3.5 Evaluation

Varying the aforementioned hyperparameters yields $n_{sp} = 25 \cdot 3 \cdot 2 \cdot 3 \cdot 2 = 900$ sampling points, since only 25 features-after-filter-wrapper-pairs match the condition $n_{FAW} \leq n_{FAF}$. One sample point is, e.g., $n_{FAF} = 200$, $n_{FAW} = 15$, $NPR = 3$, $SPLIT = 0.8$, $WA = KNN$ and $TFA = 1$. Every point is sampled 10 times for smoothing, where e.g. the set of randomly subsampled negative samples varies. Multiplied by the total number of at least partially successfully evaluated targets (1378), where more than 5 positive samples were available, this yields $n_{modellingAttempts} = n_{sp} \cdot 10 \cdot 1378 = 12.4 \cdot 10^6$. For each sampling point, minimum, maximum, standard deviation σ, mean μ, and median of auPRC, auROC, and F_1 have been recorded for evaluation. All experiments in this chapter have been conducted on an HP™ Z-840 equipped with two Intel® Xeon® E5-2640 v3 2.60GHz CPUs and 96GB of RAM in March 2018.

The pipeline can only be evaluated if the available features allow for a creation of models at all. Only potential targets from the diagnostic data set (Section 3.3.1) with five or more observations have been evaluated. This only holds true for ≈ 35% of all potential targets. Figure 3.4 shows the percentage of models for the targets with more than five observations that performed above a given average auROC, auPRC and F_1 score in dependence on target type. As mentioned earlier, target types are SPs, TAs, and DCs. As an example, 97.9% all of the models exceed $auROC = 0.9$, 47.8% of the models an $F_1 = 0.9$ and only 3.9% of the models an auPRC of 0.9. This can be interpreted that there are definitely targets that can be modelled using the given features, but there are also a lot, where the trained model does not perform well.

Across all measures, SPs allow for better models to be trained (the percentage of models exceeding all minimum performance thresholds shown in Figure 3.4 is always higher in comparison to DCs and TAs), and – with only a few exceptions – TAs can be better modelled than DCs.

An explanation for low AUC and F_1 values is, aside from low feature quality, that some potential targets are too generic. Take, e.g., a bolt (an SP) that is used in a variety of different situations, the clearing of all DTCs (a TA) which is common after a fault has been found and fixed, or a planned routine maintenance (a DC) that has no unique triggers other than

an approximate mileage or other specified usage. An explanation for the generally lower scoring models of DCs compared to SPs may be the hash-like nature of DCs that summarizes potentially multiple SPs and TAs which makes it "blurry", and thus hard to model.

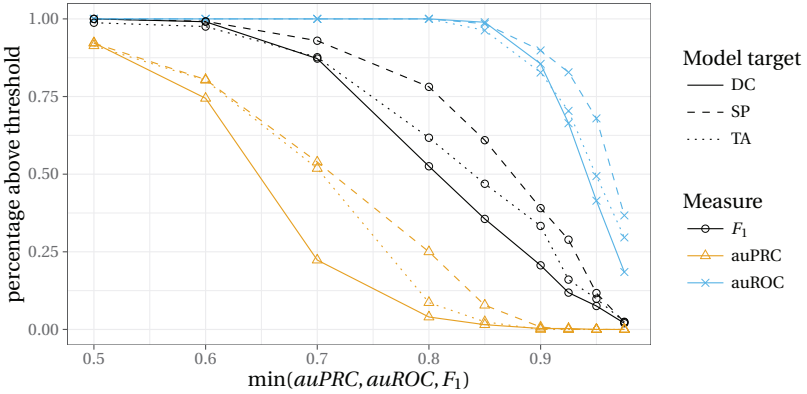


Figure 3.4: Average model performance depending on target type.

In Section 2.3 it was already argued for the superiority of auPRC under the given, foremost imbalanced, circumstances. Figure 3.4 underlines this statement: While 100% ("all") the models exceed an auROC of roughly 0.8, both the F_1 score and auPRC allow for a finer distinction of the model quality. Also, percentage of models performing above a given auPRC threshold starts decreasing at $auPRC = 0.5$ while even the percentage of models performing better than $F_1 = 0.5$ score is roughly the same (99.6%) as for $F_1 = 0.6$ (98.6%).

To conclude: There are targets that can be modeled given the available features. The remaining sections of this chapter will therefore examine the effect of the pipeline hyperparameters on classification performance and computational complexity, evaluate the algorithms, and give a measure of the overall pipeline performance.

3.5.1 Detailed Look on Feature Group Importance

Evaluating all successfully trained models, the used feature groups (FGs) are evaluated in Table 3.6. The row "Bias" references the linear offset of the LR model (used in the final model layer), being part of all created models.

The column "used" shows the group affiliation of the used features averaged across all models. For example, a value of 66.5% of the MV row means that on average, the features

used in the final model belonged in 66.5% of the cases to the MV group. This can be formalized:

$$
\begin{aligned}
used(FG) &= \frac{\sum_{m=1}^{M}\sum_{f=1}^{F_m}\mathscr{C}(f,FG)}{\sum_{m=1}^{M}F_m}\text{, with}\\
\mathscr{C}(f,FG) &= \begin{cases} 1, \text{if } f \text{ is a feature of } FG \\ 0, \text{otherwise} \end{cases}\\
M &: \text{number of models,}\\
F_m &: \text{number of features in model } m.
\end{aligned}
\tag{3.26}
$$

Suppose, $M = 2$ models: The first model $m = 1$ consists of $F_{m=1} = 6$ features, while only 3 of the features used by the model are from the feature group MV. The second model $m = 2$ consists of $F_{m=2} = 11$ features, with 7 of them being a MV. Equation (3.26) would thus yield (assuming that the MVs features are the most important and thus come first):

$$
used(MV) = \frac{(1+1+1+0+0+0)+(1+1+1+1+1+1+1+1+0+0+0+0)}{6+11} = 58.8\%. \tag{3.27}
$$

Whereby "used" only considers if a feature from the corresponding group is used or not, the "weighted" column also takes the feature weight into account, used by the L1 regularized LR model after training. The summed feature weight of each feature group is divided by the total weight of all features used by the model. Again, this number is averaged across all models for all FGs:

$$
\begin{aligned}
weighted(FG) &= \frac{\sum_{m=1}^{M}\sum_{f=1}^{F_m}\mathscr{W}(f,FG)}{\sum_{m=1}^{M}\sum_{f=1}^{F_m}\sum_{g}^{G}\mathscr{W}(f,g)}\text{, with}\\
\mathscr{W}(f,FG) &= \begin{cases} w_{m,f}, & \text{if } f \text{ is a feature of } FG \\ 0, & \text{otherwise} \end{cases}\\
w_{m,f} &: \text{weight of feature } f \text{ in model } m,\\
G &: \text{number of feature groups.}
\end{aligned}
\tag{3.28}
$$

In total, $G = 6$ feature groups exist (MV, RO, CP, EC, DTC, and EE). The term $\sum_{f=1}^{F_m}\sum_{g}^{G}\mathscr{W}(f,g)$ yields the summed weight of all features for a given model.

With a weighted importance of roughly 60%, MVs turned out to be the most informative

Table 3.6: Feature group importance.

FG	used	weighted
MV	66.5%	58.6%
Bias	5.2%	15.4%
RO	4.6%	6.9%
CP	5.1%	6.3%
EC	4.7%	4.5%
DTC	5.1%	4.3%
EE	8.9%	4%

feature groups in the model creation process and also the only one being of higher weighted importance as the bias. Surprisingly, they have not been leveraged in prior work [86]. The low impact of DTCs can be explained by the complex set of conditions by which they are triggered: Often, multiple (even timed) conditions are joined by different logical operators (`OR`, `AND`, `XOR`, etc.) to finally cause a DTC to be flagged. Despite this wealth of information that is needed to flag a DTC, the way back from a DTC to the conditions upon which it was triggered is surjective and not bijective - different sets of conditions can cause the same DTC. Also, the feature groups type RO or CP had low impact. This can be explained by most of the features from this group having the same value within the evaluated data set. They may become important if the data set is extended to include more than one car model and engine. Any FG with a lower weighted importance than the "Bias" is arguably useless on average since a constant value (the Bias) is more "informative" to the model, but may provide useful information in specific scenarios.

3.5.2 Filter Layer Evaluation

The implemented filter measures are evaluated to identify those filter measures that positively affect the performance of the pipeline. On the other hand, measures that only contribute little to the overall classification performance but significantly increase the computational complexity can be eliminated. This way, the pipeline is optimized for future research.

To rank the filter measures, more than 39000 filter rankings have been matched with the resulting LR models trained in the last stage of the pipeline (green box on the right of Figure 3.2) to yield the results listed in Table 3.7. The corresponding columns shall be explained

in the following.

First, the total number of features $n_{GTw_{min}}$ with an absolute weight higher than w_{min} = 0.001 across all LR models trained in the last stage is calculated:

$$n_{GTw_{min}} = \sum_{m=1}^{M} \sum_{f=1}^{F_m} \mathcal{N}(feature_f, model_m) \text{ , with}$$
$$\mathcal{N}(feature, model) = \begin{cases} 1, & \text{if } \left|\mathcal{I}_{model}(feature)\right| > w_{min} \\ 0, & \text{otherwise} \end{cases} \tag{3.29}$$

In the above formula, $\mathcal{I}_{model}(feature)$ refers the weight of the given *feature* in the LR *model*.

Every time, a filter measure ranks a feature higher than zero that is used in the LR model created in the final stage with an absolute feature weight higher than w_{min}, a "hit" is counted. The column "hit" in Table 3.7 is the number of hits (every hit counts as 1) from each measure divided by the total number of features with an absolute weight higher than w_{min} ($n_{GTw_{min}}$), across all trained models M.

$$hit(filter) = \frac{\sum_{m=1}^{M} \sum_{p=1}^{P} \mathcal{H}(X_p, filter, model_m)}{n_{GTw_{min}}} \text{ , with}$$
$$\mathcal{H}(X, filter, model) = \begin{cases} 1, & \text{if } \left|\mathcal{I}_{filter}(X)\right| > 0 \text{ and } |\mathcal{I}_{model}(X)| > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.30}$$

The term $\mathcal{I}_{filter}(feature)$ refers the importance of *feature* according to *filter*. Therefore, a high "hit percentage" does not necessarily indicate an useful filter measure: A filter measure ranking all features passed to the next layer with an "importance" ranking of 0.01 would score 100%.

Thus, in addition to solely counting the number of "hits", the "contribution" column takes the filter measure ranking and the final feature weight of the trained model into account. In case of a "hit", the filter measure importance ranking is multiplied by the feature weight in the model. This number is summed over all models and divided by $n_{GTw_{min}}$. This allows to assess the correlation between the filter measure scoring and the final feature weight in the

Table 3.7: Filter measure performance.

measure	hit	contribution	runtime [s]
χ^2	82.025%	0.858	2.178
Correlation	98.533%	0.697	0.469
Gini	99.890%	0.549	0.245
Mutual Information	82.025%	0.733	2.611
Relief	99.800%	0.463	70.577

model in a much better way.

$$\begin{aligned} contribution(filter) &= \frac{\sum_{m=1}^{M}\sum_{p=1}^{P}\mathscr{C}(X_p, filter, model_m)}{n_{GTw_{min}}}\text{, with} \\ \mathscr{C}(X, filter, model) &= |\mathscr{I}_{model}(X)| \cdot \left|\mathscr{I}_{filter}(X)\right| \end{aligned} \tag{3.31}$$

Since the feature weights in final model $\mathscr{I}_{model}(feature)$ are not scaled to a $[0,1]$ range, the "contribution" is not normalized. To quickly recapitulate the filter measures introduced in Section 3.1.2: *Correlation* is used to measure the correlation between the feature and the target variable. *Chi-squared* χ^2 measures the dependence of the feature and the target distribution. The *Gini coefficient* measures the inequality for the features frequency distribution. The *information* filter is an entropy-based (H) measure measuring the mutual information of the feature and the target. And finally: *Relief* estimates features based on how well their values can be used to distinguish instances that are close to each other in the feature space, while considering their class membership [71].

Column "runtime [s]" of Table 3.7 shows the time it took to perform all necessary computations as described in the corresponding subsections of Section 3.1.2. This includes not only the actual feature assessment but also computations such as binning that may be required in dependence on the filter measure.

Considering Table 3.7, the meaningfulness of the Relief measure may be questioned due to its extraordinary high computational cost, and its inability to identify relevant features: A high percentage of hits, but low contribution is indicating that the Relief is ranking most of the features equally high.

Also, the results of χ^2 and the MI-based filter yield the exact same results regarding the hit percentage. Therefore, the MI measure may be dropped due to its longer runtime compared

to the χ^2 measure if the computational complexity needs to be reduced. Additionally, the χ^2 achieved the highest contribution in this comparison.

An explanation for the relatively low contribution of the Gini coefficient can be explained by considering how the algorithm actually works, as explained in Section 3.1.2: The fact that a $X_1 = [0,0,0,0,1]$ holding five samples yields an extremely high score ($Gini(X_1) = 0.8$) and $X_2 = [1,1,1,1,0]$ yields a very low score ($Gini(X_2) = 0.2$), which makes perfect sense in an economic scenario: X_1 could, e.g., describe an income situation where one person in a company earns all the money. However, this does not make sense, in a machine learning context, where both features supposedly carry the same amount of information. Thus, the following formula is proposed to tackle this issue:

$$Gini_{FS}(X) = \left| \frac{Gini(X) + Gini(X - \max(X))}{2} \right|. \tag{3.32}$$

Which would yield $Gini_{FS} = 0.3$ for both features alike (X_1 and X_2). Also, this is the only filter measure evaluated in this thesis, that does not take the class label Y into consideration. This is a serious issue as the following example demonstrates. Suppose the following data set, consisting of two features X_1 and X_2 as well as the label Y:

$$\begin{matrix} X_1 & X_2 \end{matrix} \quad Y$$
$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \tag{3.33}$$

Both features would score the same ranking according to the Gini coefficient $Gini(X_1) = Gini(X_2) = 0.5$. This is counter-intuitive, since feature X_1 correlates perfectly with label Y, while feature X_2 is completely useless to determine between the different outcomes of Y.

3.5.3 Wrapper Layer Evaluation

Three different wrapper algorithms were evaluated (hyperparameter *WA*): A greedy k-NN, RF, and an LR. Table 3.8 shows the achieved auROC, auPRC and F_1 scores as well as the runtime (see below) for the corresponding wrapper algorithms averaged across all LR models built in the final pipeline stage. A *auPRC* = 0.644 for the k-NN wrapper thus refers the aver-

Table 3.8: Influence of the wrapper algorithm on the pipeline performance.

algorithm	auROC	auPRC	F_1	runtime [s]
k-NN	0.927	0.644	0.775	31.931
LR	0.955	0.691	0.850	0.286
RF	0.934	0.654	0.801	10.471

aged auPRC achieved by the final LR model if a k-NN wrapper was used to select the features. Similarly to the previous section, "run-time [s]" includes all steps necessary to rank the features using the corresponding wrapper algorithm as described in Section 3.1.3.

The k-NN and RF wrapper algorithms yield comparable results in terms of averaged model performance, with the RF wrapper yielding a slight advantage. The LR wrapper yields slightly higher auROC, auPRC, and F_1 scores. This may be due to the fact that the model layer also uses a LR model. In terms of run-time, LR outperforms all other approaches by several orders of magnitude. This can be explained by the much less expensive training: LR is optimized by gradient descent, while RF requires iterative branching and k-NN storing and iterating over a huge amount of samples as stored reference.

Usually one would remove features that only are have a particularly low weight in the model and train the model again. Since the actual value of "particularly low" may vary, this – usually manual – process is reflected by various thresholds t in Figure 3.5. Here, the number of actually used features by the LR model (final stage) n_{FUM} is shown in dependence on n_{FAW}. Especially for high thresholds, where a feature is only counted if its feature weight w in the final model exceeds the threshold $t \geq 1$, n_{FUM} starts decreasing after more than $n_{FAW} \approx 50$ features are passed to the model. Averaged across all trained models, the feature weights inferred by the LR model of the final stage after training are defined by the following metrics: $mean(w) = \mu(w) = 0.012$, $median(w) = 0$, $\sigma(w) = 2.41$, $min(w) = -680$, $max(w) = 413$. Thus, the number of actually relevant features which would remain after the manual removal of the irrelevant ones (e.g. for $t = 0.1$) would increase only slightly if more than $n_{FAW} = 50$ are passed to the final stage by the wrapper layer, and for $t = 1.0$ even decrease.

3.5.4 Remaining Hyperparameters

The influence of n_{FAF} on the pipeline's performance is given in Table 3.9. Results regarding n_{FAW} are presented in Table 3.10. Both, n_{FAF} and n_{FAW} positively correlate with the pipeline's

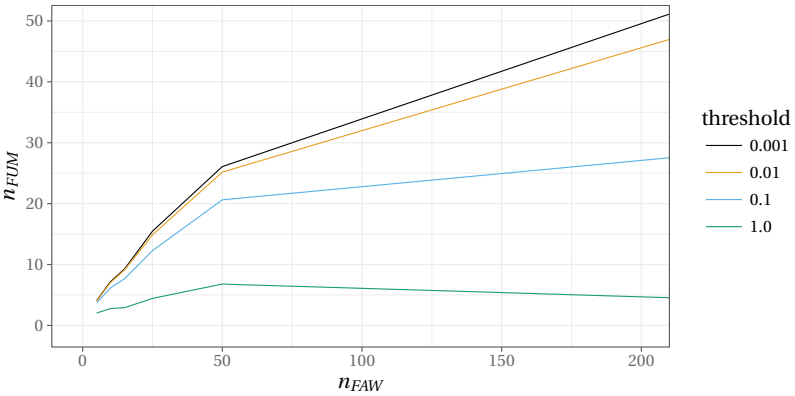


Figure 3.5: Number of features used in final model above threshold.

Table 3.9: Influence of n_{FAF} on pipeline performance.

n_{FAF}	auROC	auPRC	F_1
25	0.933	0.650	0.790
50	0.939	0.662	0.809
100	0.940	0.665	0.812
200	0.939	0.665	0.811
500	0.940	0.669	0.816

performance (e.g., measured in auPRC: $corr(n_{FAF}, auPRC) = 0.025$, $corr(n_{FAW}, auPRC) = 0.076$). This means, the more features are passed from the filter layer over the wrapper layer to the model layer, the higher the performance (in terms of AUC and F_1).

Although a higher n_{FAW} leads to a higher model performance, as shown in Table 3.10 the training time of the LR model in final pipeline stage increases disproportionately: While the auPRC increases by only 13.1% (F_1 increases by 18.1%) when increasing the number of features used by the model from 5 to 50, the model training time increases by 65.2%. Since the increase of training time is 5 times (3.5 times) as high as the increase of the auPRC (F_1), the optimum number of features passed from the wrapper layer to the model is set implicitly by the desired model performance in terms of auPRC and F_1.

The next evaluated hyperparameter is *NPR*: According to the results in Table 3.11, the most relevant measure (auPRC, as shown in Section 2.3) decreases with a higher *NPR*. Also, the F_1 decreases. The only exception is the auROC score, which neither in- nor decreases

Table 3.10: Influence of n_{FAW} on model performance and training time.

n_{FAW}	auROC	auPRC	F_1	average training time [s]
5	0.909	0.617	0.728	0.118
10	0.932	0.651	0.794	0.156
15	0.941	0.666	0.819	0.177
25	0.950	0.681	0.841	0.191
50	0.960	0.698	0.860	0.195
500	0.964	0.709	0.866	0.257

Table 3.11: Influences of *NPR* on the pipeline performance and training time.

NPR	auROC	auPRC	F_1	model training time [s]
1	0.939	0.724	0.915	0.090
3	0.937	0.648	0.791	0.159
5	0.940	0.613	0.714	0.267

significantly. Note that (unlike the more sophisticated upsampling techniques discussed in Chapter 4, which are discussed separately) only naive downsampling is applied in this experiment. This means, *NPR* = 1 undersamples all samples that are not labeled with with the target class, until there is the same number of target and non-target samples in the training set. Since a higher *NPR* only increases the training time according to the this experiment on the given, automotive data set, *NPR* values larger than 1 are not recommended.

An explanation, why F_1 does not increase, can be explained by the way F_1 is calculated, see Equation (2.21) [39]: It represents the *harmonic mean* of *precision* and *recall*. Thus, a lower *precision* always causes a lower F_1 if the *recall* remains the same. *Precision* and *recall* are calculated as shown in Equation (2.19) and Equation (2.20).

An increase of the number of negative samples in the used data set will always lead to a higher number of FPs under the assumption that the FP to TN ratio is constant. This leads to a smaller *precision* while the *recall* is not dependent on the number of negative samples. Therefore, increasing the number of negative observations in the used data set, will always lead to a smaller or equal F_1. Similarly, this applies to the auPRC, which is also dependent on precision and recall. Under the assumption, that the recall values for different decision thresholds remain the same, the precision is lowered as described above, which will reduce the area under the curve.

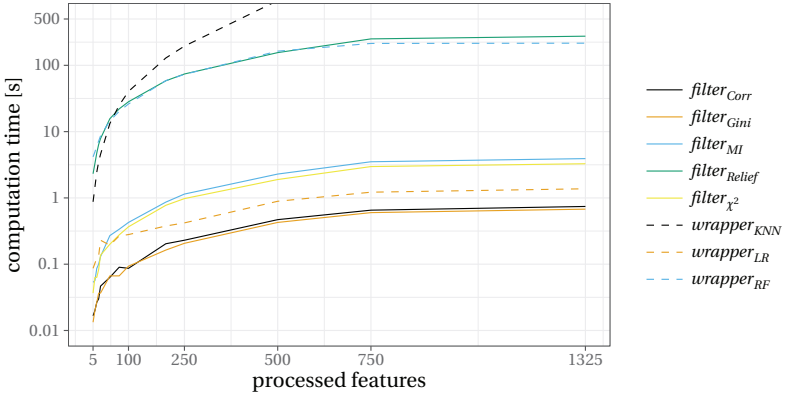


Figure 3.6: Computation time: k-NN, RF, LR, and all filter measures.

The influence of *SPLIT* and *TFA* can be neglected. Regarding *TFA*, this may be explained by the fact that a feature that is ranked high by a specific filter measure will be very likely part of the feature set passed to the wrapper layer if 20 or even 200 features are selected. However, this is only a conjecture. The neglectable influence of *SPLIT* may be explained by the similarity of the two values evaluated (0.7 and 0.8).

Aside from evaluating the given hyperparameters, the computational complexity and scalability of the proposed pipeline was evaluated. Figure 3.6 shows the logarithmically scaled computation time in dependence on the number of processed features. For the continuous values, Table 3.12 is referred. To collect the displayed values, all algorithms were measured one after another three times in immediate succession and the results were averaged. Important things to note are: The LR wrapper measure computes extremely fast, even outperforming most of the filter algorithms except for correlation and Gini coefficient. Unfortunately, the Gini coefficient is the filter measure, that performs worst in terms of "contribution" according to Table 3.7. Correlation ($filter_{Corr}$), on the other hand has a reasonable high "contribution", one of the highest "% hit", and is faster than the LR wrapper. At first glance, it may not make sense to compare filters with wrappers. However, situations in which one has to choose between filters or wrappers (e.g. because both in combination show too high computational runtime complexity) are possible and thus the comparison makes sense. The relief filter is also not advisable, since an RF is faster. Both, χ^2, and the MI ($filter_{gain}$) filter run faster than the RF wrapper. The χ^2 filter also scored the first rank in terms of "contribu-

tion" (Table 3.7) and thus may be the primary choice in scenarios with a nonlinear decision boundary (where a LR wrapper may not suitable). Lastly, the computation time of the k-NN algorithm grows disproportionally ($\mathcal{O}(N^2)$). Thus, this technique is useless in the given context.

To approximate the time savings gained by using the proposed pipeline, the following conservative formula was used (where 100 features are processed by the RF wrapper). The processing time of a standalone wrapper approach is compared to a combined filter/wrapper approach, where the χ^2 and correlation filters are used. The number of features fed into does not exceed $n_{FAP} = 1325$ in this example (for details on how features are filtered out prior to the filter layer please refer Section 3.4.1). This limitation may not hold in future or other applications, where the time savings would be even greater:

$$\begin{aligned} t_{saving,RF} &= t_{RF,unfiltered} - (t_{filter_{\chi^2},unfiltered} + t_{filter_{corr},unfiltered} + t_{RF,100}) \\ &= 215.790s - (3.277s + 0.743s + 26.140s) \\ &= 185.63s \end{aligned}$$

which results in a saving of

$$t_{saving,RF} / t_{RF,unfiltered} = 86.023\%$$

However, as shown in Table 3.12, there are combinations where no time is saved. E.g., if the filter layer utilizes all filter measures (except the Relief filter) and the wrapper layer is using an LR in comparison to just building the final model using an LR:

$$\begin{aligned} t_{saving,LR} &= t_{LR,unfiltered} - (t_{filter_{all},unfiltered} + t_{LR,100}) \\ &= 1.373s - (0.743s + 0.677s + 3.917s + 3.277s + 0.280s) \\ &= -7.521s \end{aligned}$$

Table 3.12: Overview over the layer runtimes in seconds depending on the number of processed features.

$n_{feature}$	$filter_{Corr}$	$filter_{Gini}$	$filter_{MI}$	$filter_{Relief}$	$filter_{\chi^2}$	$wrapper_{KNN}$	$wrapper_{LR}$	$wrapper_{RF}$
5	0.017	0.013	0.053	2.313	0.037	0.873	0.087	4.163
10	0.020	0.020	0.057	3.307	0.063	1.460	0.103	4.947
15	0.027	0.027	0.087	4.783	0.063	2.463	0.123	6.203
20	0.030	0.037	0.100	6.400	0.083	3.500	0.133	7.187
25	0.047	0.037	0.130	7.850	0.137	4.543	0.230	8.617
50	0.063	0.067	0.270	15.657	0.200	13.230	0.193	14.563
75	0.090	0.067	0.337	22.037	0.277	25.007	0.267	20.177
100	0.087	0.093	0.427	28.213	0.367	40.400	0.280	26.140
200	0.203	0.163	0.863	58.430	0.773	128.987	0.377	58.773
250	0.230	0.207	1.143	74.060	0.977	194.883	0.420	73.087
500	0.470	0.427	2.290	155.487	1.893	948.083	0.890	163.650
750	0.653	0.600	3.510	250.477	2.970	16016.837	1.220	213.970
1325	0.743	0.677	3.917	275.097	3.277	19330.133	1.373	215.790

3.5.5 Evaluation on Publicly Available Data

For the *golub* data set, a multidimensional hyperplane exists, separating the two classes without classification errors. For certain parameterizations, the pipeline scores an averaged $auPRC = auROC = F_1 = 1$, which has also been achieved by Guyon et al. [54]. Table 3.13 shows the top and last five pipeline parameterizations, sorted by $\epsilon = auPRC + F_1$. Sorting by ϵ ensures that results with both, a high auPRC and F_1 will appear. This eases the comparision with Guyon et al. [54] who used F_1. Keeping the original 833 features in mind, especially the rows where $n_{FAW} = 5$ represent a major benefit in terms of computational complexity because of an enormously reduced feature space. The proposed pipeline was able to reduce the feature space by $1 - \frac{5}{833} = 99.4\%$ without compromising model classification performance. The optimum solution in terms of *AUC* is possible with other parametrizations as well. Since an LR model was used, this means that there exists a linear decision boundary.

Generally, the k-NN wrapper leads to higher performing models on the *golub* data set ($corr(KNN, auPRC) = 0.343$), while the RF wrapper does not perform well ($corr(RF, auPRC) = -0.399$). A high *NPR* does not yield better performance ($corr(NPR, auPRC) = -0.210$). *TFA* is also a beneficial technique ($corr(TFA, auPRC) = 0.129$). The LR wrapper does not increase all

Table 3.13: Performance of the pipeline on the *golub* data set.

ϵ	*auROC*	*auPRC*	F_1	n_{FAF}	n_{FAW}	*NPR*	*WA*	*TFA*
1.929	1.000	0.929	1.000	100	5	5	LR	1
1.929	1.000	0.929	1.000	100	10	5	LR	1
1.929	1.000	0.929	1.000	200	5	5	LR	1
1.929	1.000	0.929	1.000	500	10	5	LR	1
1.917	1.000	0.917	1.000	50	5	3	KNN	0
1.279	0.909	0.682	0.597	25	15	10	LR	1
1.277	0.889	0.627	0.650	100	5	10	LR	1
1.223	1.000	0.500	0.723	500	500	5	RF	0
1.218	0.896	0.642	0.577	25	10	10	LR	1
1.071	0.835	0.611	0.460	200	5	10	LR	0

classification measures on the *golub* data set ($corr(LR, auROC) = -0.112$, $corr(LR, auPRC) = -0.056$, $corr(LR, F_1) = -0.176$) in contrast to the results based on the automotive data set.

Table 3.14: Performance of the pipeline on the *secom* data set.

ϵ	*auROC*	*auPRC*	F_1	n_{FAF}	n_{FAW}	*NPR*	*WA*	*TFA*
1.527	0.786	0.814	0.713	100	15	1	LR	1
1.525	0.840	0.855	0.670	50	5	1	LR	1
1.502	0.815	0.836	0.666	50	10	1	LR	1
1.494	0.796	0.734	0.760	500	50	1	KNN	0
1.481	0.809	0.809	0.672	25	25	1	LR	1
0.096	0.646	0.096	0.000	500	25	10	LR	1
0.096	0.644	0.096	0.000	200	50	10	RF	0
0.093	0.637	0.093	0.000	100	50	10	RF	0
0.086	0.527	0.086	0.000	500	10	10	KNN	1
0.081	0.513	0.081	0.000	500	5	10	KNN	1

The scores in terms of ϵ on the *secom* data set are promising as well (again, the top and worst five results regarding ϵ are shown in Table 3.14) but lower compared to the $F_1 = 0.947$ achieved by Arif et al. [3] with a higher manual effort. An explanation may be – aside from the positive influence of the manual effort – that the data set requires a nonlinear model to correctly represent the relationship between the features and the labels: While Arif et al. [3] used a non-linear RF based model, the LR used in this work is a linear model. Also, since an LR model was used to generate the classification grade metrics, this could also mean that

there simply exists no linear decision boundary. The five highest ϵ scores have been achieved after reducing the 591 dimensional feature space to $5-50$ features. The RF wrapper was included in this evaluation and scored $\epsilon_{RF,avg} = 0.711$ averaged over all parameterizations ($\sigma_{\epsilon,RF} = 0.404$). This is higher compared to LR and k-NN in terms of the average ($\epsilon_{LR,mean} = 0.647$ and $\epsilon_{KNN,mean} = 0.658$), and lower in terms of the standard deviation ($\sigma_{\epsilon,LR} = 0.434$ and $\sigma_{\epsilon,KNN} = 0.424$).

A high *NPR* decreased the classification performance ($corr(NPR, auPRC) = -0.210$) in this case – which is also outlined by Table 3.14, where the top 5 lines have a low value of *NPR* in common. The performance was affected negatively by the *TFA* technique ($corr(TFA, auPRC) = -0.129$) and the RF wrapper algorithm ($corr(RF, auPRC) = -0.399$).

3.6 Summary and Conclusion

The final scenario discussed in the following only considers the RF and LR wrappers due the extremely high computational complexity of the k-NN wrapper as visualized in Figure 3.6. The averaged runtimes accompanied by the respective classification grades, are shown for the RF (Table 3.15, Table 3.16, and Table 3.17 for auROC, auPRC and F_1) and the LR wrapper (Table 3.18, Table 3.19, and Table 3.20 for auROC, auPRC and F_1), respectively. First, the feature space was prepared and features with variance close to zero were filtered out. This already reduced the multi-thousand dimensional feature space to 1325 features after preparation (n_{FAP}). These were processed and ranked by the filter layer, consisting of all filter measures.

According to the filter measure ranking, the n_{FAF} top ranked features after the filter layer were passed to the wrapper. As expected, the wrapper training time increases when more features are being passed by the filter. Except for LR, the wrapper layer is usually computationally more expensive (Figure 3.6) when processing the same amount of features compared to the filter layer. The model training time is given in the last row.

Together with n_{FAW} features after the wrapper layer, this spans up a matrix with model classification performance scores. For each column of n_{FAW} the corresponding model training time is given – which also increases when more features are processed.

To adapt the pipeline to other applications, the following recommendations may be helpful: Which wrapper to choose may be strongly dependent on the data set. In case of the

Table 3.15: Overview of pipeline performance and runtime using an RF wrapper and auROC.

n_{FAF}	wrapper training time [s]	5	10	15	25	50	500	n_{FAW}
25	8.617	0.902	0.926	0.937	0.948	–	–	
50	14.563	0.904	0.926	0.936	0.946	0.958	–	
100	26.140	0.906	0.927	0.937	0.946	0.957	–	
200	58.773	0.906	0.926	0.936	0.945	0.956	–	
500	163.650	0.903	0.924	0.934	0.944	0.955	0.962	
	model training time [s]	0.103	0.107	0.127	0.223	0.197	0.847	

Table 3.16: Overview of pipeline performance and runtime using an RF wrapper and auPRC.

n_{FAF}	wrapper training time [s]	5	10	15	25	50	500	n_{FAW}
25	8.617	0.603	0.637	0.655	0.673	–	–	
50	14.563	0.608	0.638	0.656	0.673	0.692	–	
100	26.140	0.612	0.643	0.659	0.675	0.693	–	
200	58.773	0.611	0.642	0.657	0.673	0.693	–	
500	163.650	0.607	0.639	0.655	0.671	0.690	0.705	
	model training time [s]	0.103	0.107	0.127	0.223	0.197	0.847	

Table 3.17: Overview of pipeline performance and runtime using an RF wrapper and F_1.

n_{FAF}	wrapper training time [s]	5	10	15	25	50	500	n_{FAW}
25	8.617	0.708	0.774	0.809	0.836	–	–	
50	14.563	0.718	0.777	0.809	0.836	0.857	–	
100	26.140	0.729	0.788	0.814	0.837	0.854	–	
200	58.773	0.730	0.788	0.814	0.835	0.853	–	
500	163.650	0.723	0.785	0.809	0.832	0.852	0.865	
	model training time [s]	0.103	0.107	0.127	0.223	0.197	0.847	

Table 3.18: Overview of pipeline performance and runtime using an LR wrapper and auROC.

n_{FAF}	wrapper training time [s]	5	10	15	25	50	500	n_{FAW}
25	0.230	0.939	0.948	0.950	0.951	–	–	
50	0.193	0.942	0.955	0.958	0.960	0.960	–	
100	0.280	0.941	0.957	0.962	0.965	0.966	–	
200	0.377	0.938	0.957	0.962	0.966	0.968	–	
500	0.890	0.932	0.953	0.959	0.964	0.966	0.967	
	model training time [s]	0.103	0.107	0.127	0.223	0.197	0.847	

Table 3.19: Overview of pipeline performance and runtime using an LR wrapper and auPRC.

n_{FAF}	wrapper training time [s]	5	10	15	25	50	500	n_{FAW}
25	0.230	0.657	0.673	0.676	0.677	–	–	
50	0.193	0.664	0.687	0.692	0.695	0.696	–	
100	0.280	0.664	0.695	0.702	0.707	0.710	–	
200	0.377	0.659	0.696	0.706	0.711	0.715	–	
500	0.890	0.653	0.692	0.701	0.709	0.712	0.715	
	model training time [s]	0.103	0.107	0.127	0.223	0.197	0.847	

Table 3.20: Overview of pipeline performance and runtime using an LR wrapper and F_1.

n_{FAF}	wrapper training time [s]	5	10	15	25	50	500	n_{FAW}
25	0.230	0.803	0.831	0.837	0.838	–	–	
50	0.193	0.817	0.850	0.856	0.859	0.859	–	
100	0.280	0.817	0.856	0.864	0.868	0.871	–	
200	0.377	0.817	0.859	0.866	0.870	0.872	–	
500	0.890	0.813	0.854	0.863	0.866	0.867	0.868	
	model training time [s]	0.103	0.107	0.127	0.223	0.197	0.847	

automotive data set, the LR wrapper yielded a high auROC, auPRC, and F_1 while requiring a minimum of computational power. In contrast, LR yielded low results as wrapper for the *golub* data set. Also, the wrapper evaluation may be biased towards the LR wrapper, since the LR algorithm was also used in the final model layer of the pipeline. This represents the common scenario. Using the same algorithm for both, the feature selection and the model building may be beneficial. In this case, combining wrapper and model layer might be possible.

Selecting the highest ranked features of every filter algorithm used in any case (*TFA* parameter, see Section 3.4.5) is an useful technique, if the number of features passed to the wrapper has to be very low (e.g. 5). In most cases, the available computing power allows to process 200 or more features. This feature space will very likely be a superset of the feature space selected by *TFA*, making this technique unnecessary in most scenarios.

In general, *feature selection speeds up model building*. In scenarios, where the data set is reduced once, and a lot of different models are evaluated on the reduced data set, this might save large amounts of time, while yielding comparable model classification performances (e.g., auPRC) compared to the original feature space. Procedures in which the optimal model is first selected on a reduced number of features and then trained on all features are conceiv-

able.

After evaluating all filter measures introduced in Section 3.1, using a subset of filter measures that is tailored to the needs of the application is recommended. If too many filter measures are used at the same time, the filter layer can be computationally more expensive than "cheap" wrapper layers, e.g., wrappers using LR. In case of an LR wrapper, the necessity of using a filter layer at all, must be evaluated on a case-by-case basis.

Averaging the pipeline performance in terms of auPRC across all trained models (see e.g. Table 3.16) of the automotive data set, the best results are achieved with a minor selection before the wrapper took place when $n_{FAF} = 200$ and $n_{FAW} = 50$ were used, respectively. In this case (RF wrapper based on auPRC), further increasing n_{FAF} from 200 to 500 did only yield slightly higher auPRC results ($\frac{0.705}{0.693} - 1 = +1.73\%$) but a much longer RF wrapper selection ($\frac{163.65}{58.773} - 1 = +178\%$) time.

There may be scenarios where the wrapper or filter layers are computationally more expensive than the model layer. In case an increased number of features does not affect the generalization of the final model negatively, the wrapper and / or filter layer may be unnecessary. As described above, using $n_{FAF} = 200$, $n_{FAW} = 50$ represents a good compromise between a high model performance, a low training time, and a low feature selection time.

Although the optimization of the pipeline took more than 2 weeks on a 32 core, 96GB workstation, the pipeline can be considered absolutely scalable once the hyperparameters are set, especially when a subset of filter measures is picked based on the insights gained in this chapter. A possible usage scenario is that all hyperparameters are tuned once, and can then be used multiple times for the similar application or data set within a cooperation.

In this chapter, a variety of different algorithms were evaluated and chained in the most efficient way to create an automotive feature selection pipeline. The proposed pipeline involves multiple layers of different computational complexity, does not require any manual effort, and (once tuned) is computationally unexpensive. The pipeline already yielded promising models, already successfully applied in real life scenarios for predictively maintenancing an engine component used in more than 20,000 cars. The possibility to autonomously select features and create models for more than 3000 potential targets showed that only 200 well-selected features out of the 5000 available can be sufficient to successfully create models (see, e.g., Table 3.15). Also, considering Figure 3.4, in many cases, well-performing (in terms of auPRC) models can not be created given the currently available data.

Future work could evaluate if there is a way to assess potential model performance before actually computing the whole pipeline. E.g., if a link between selected filter measures and the expected model performance in terms of auPRC could be established, this would allow the definition of a “early stopping” criterion. E.g., if all available features receive a “poor” (“unimportant”) ranking by all filter measures, a “successfull” (e.g. in terms of a high auPRC) will be unlikely. This would allow to save huge amounts of processing time by skipping unpromising modeling targets.

Chapter 4

Dealing with Imbalance

Highly imbalanced data sets are still a challenge in many data mining and machine learning applications, in both academia and industry. Many state-of-the-art techniques countering class imbalances are usually very computationally expensive and therefore unscalable. Most research effort has been directed towards enhancing those techniques, e.g., by focusing on borderline examples or combining multiple techniques. This inherently increases the computational complexity, reducing the scalability even further. Since the overall theme of this work is to repeatedly deal with large-scale data, this chapter examines how to deal with class imbalances the scalable way. This evaluation is layed out in isolation. Thus, techniques to select features as laid out in the previous chapter are not considered. Also, many authors claim that their technique outperforms other techniques, which leads to contradictions that can only be resolved by a neutral and objective comparison which is done in this chapter. The focus lies on two-class classification problems, since every multi-class problem can be broken down to multiple two-class problems.

In general, an "imbalanced data set" refers to a data set where samples for one potential outcome are by far outnumbered by other potential outcomes. E.g., given a two-class classification problem, a data set is referred as "imbalanced", when the class with fewer records, the *minority* class, is highly under-represented in contrast to the class with more samples, also known as the *majority* class.

This imbalanced class distribution causes several, mostly noise related issues: A k-NN could assign the wrong (majority) class label to a minority class sample due to "near" noisy majority class samples [6, 74]. A Bayesian classifier might be biased towards assigning a majority class label because of a high prior probability of the majority class [74]. In general,

classification algorithms might overfit the majority class noise in the minority class feature region [6]. Lastly, noisy (majority) class samples having little informative value can cause an increased training time and storage costs (e.g. when using a k-NN).

Applications suffering from (at least partly noisy) imbalanced data sets include automotive car diagnostics (where a fault occurs only a few times among hundreds of thousands of cars, see Section 4.3), the prediction of company bankruptcy [148], analyzing cDNA microarray time-series [90], oil-spill detection from satellite images [73], as well as fraud/intrusion detection, text classification, and medical diagnosis/monitoring [28].

Section 4.1 gives an overview over various existing preprocessing techniques (PTs) and presents a novel technique that impact the following training of under-represented classes in noisy, imbalanced data sets (NIDs). The novel technique called Class Sensitive Scaling (CSS) (Section 4.1.1) partially scales samples (non-) linearly to the corresponding class center.

In Section 4.1.2, a parameter free class overlap and noise measure is introduced to complement the existing measures such as, e.g., the balance ratio to assess the data sets' properties. Section 4.3 will present the data sets, used for the extensive evaluation performed in Section 4.4. All introduced PTs will be evaluated regarding their computational cost *and* influence on classification performance in combination with a variety of classifiers.

Section 4.5 will derive general recommendations regarding the suitability of the different techniques in dependence on data sets' properties, the used classifier and other requirements such as scalability.

4.1 State of the Art

Weiss' "unifying framework" [138] divides the different techniques to cope with NIDs into the following categories.

1. Using an appropriate evaluation metric,
2. incorporating expert knowledge,
3. learning an One-Class Classifier (OCC),
4. applying sampling,
5. using cost-sensitive learning or weighting,

6. creating ensembles using boosting.

The use of a proper evaluation metric, as proposed, e.g., by Weiss [138] and Han et al. [55], was extensively addressed in Section 2.3 and is taken as a basis for this chapter.

Although promising, the *incorporation of expert knowledge* is not evaluated in the following for two reasons: First, its effectiveness is hard to prove, especially on publicly available data sets. Second, expert knowledge conflicts with the overall goal to model thousands of potential errors that can occur in a car with as less human intervention as possible.

Training an *OCC* to learn only the minority or majority class has been tested successfully by Japkowicz et al. [63] using ANNs and Raskutti et al. [94] using Support Vector Machines (SVMs). According to Raskutti et al. [94], SVMs are particularly useful for extremely unbalanced data sets. The two most popular SVM-based approaches for one class classification according to Khan and Madden [68] are: The approach by Tax and Duin [120] tries to fit the smallest possible hyper-sphere around the majority class data. Missing some majority class samples is tolerated, if this sufficiently decreases the volume of the hyper-sphere. Schölkopf et al. [110] try to fit the data using a hyper-plane that is maximally distant from the origin with all majority class samples residing on one side of the hyper-plane. More complex decision boundaries can be created in both approaches using kernel functions. Since both of these approaches have been shown to be equivalent by Rieck [96], this chapter will focus on the hyper-plane-based approach by Schölkopf.

Another option to tackle NIDs is to apply *sampling techniques*. Various forms of sampling exist, an overview over existing techniques is given, e.g., by Chawla et al. [28] and He et al. [57]. Techniques include random under- or over-sampling, informed sampling using a heuristic that defines which samples to select, and synthetic sampling (where new samples are artificially generated). Sampling can also include data cleaning, e.g., using Tomek links [124] (see Section 4.1.1 for details). Sampling can be performed in three ways: over-sampling the minority class, under-sampling the majority class, or a combination of both. Over-sampling might increase the likelihood of overfitting [6] and add computational complexity [138]. A drawback of under-sampling is that potentially useful data is thrown away [6, 138].

In contrast to sampling, where the class distribution is balanced by following a specific strategy to “pick” a subset of samples from the data set, *cost-sensitive learning or weighting* utilizes all samples. This is intended to avoid loss of information. Cost-sensitive learning

focuses on adjusting the losses for misclassified samples associated with the different cells of the confusion matrix. Weighting adjusts the influence of every sample (e.g. class specific) on the model parameters during model training, e.g. by adjusting weights inversely proportional to class frequencies in the training data [91]. Japkowicz et al. [64] state that weighting outperforms sampling in terms of error rate in most especially artificial scenarios[1].

Another technique proposed by Weiss [138] to cope with NIDs is *boosting*. According to Joshi et al. [65], where boosting is evaluated for NIDs, there is no guarantee that boosting improves the classification performance of a minority class. Furthermore, the analysis shows that the best base learner can be identified by comparing the standalone (non-boosted) performance of various base learners. Therefore, this chapter will not consider and evaluate boosting since this would yield the same trends as just comparing the (non-boosted) base learners and just raise the simulation time.

Chawla et al. [28] also propose feature selection as an useful technique, since selecting the features that capture the high skew in the class distribution can lead to a better separability between the two classes (e.g. by selecting features for the minority and majority classes separately and combining them afterwards [147]). Since feature selection has already been addressed in Chapter 3 based on Schlegel and Sick [109] it is therefore not addressed again in this chapter. Weiss [138] is referred for more information.

To summarize: Expert knowledge is extremely context sensitive and creating ensembles does not change the ranking of the base learner. Therefore, this chapter is focused on the remaining points for a "unifying framework".

4.1.1 Preprocessing Techniques for Imbalanced Data Sets

This section explains the most common (in terms of citations) *foundational techniques*, evaluated in the simulation in greater detail. If available, the algorithms from the `imbalanced-learn` package [75] were used, which is implemented in `Python`. Regarding the novel technique proposed in Section 4.1.1 a pull-request has been submitted. If not, details are given in the corresponding section.

Recapitulate Section 2.1: $\mathcal{S}$ denotes the original training set, holding N samples $\mathcal{S} = \{s_1, ..., s_N\}$, with each sample s_n for $n = 1, \ldots, N$ consisting of a vector of feature values

[1] Only in one experiment (out of 25 artificial experiments) oversampling was more accurate. On real world datasets, weighting outperformed sampling in two out of eight cases (three cases yielded a tie, sampling outperformed weighting in three cases).

$\mathbf{x}_n$ and a label y: $s_n = (\mathbf{x}_n, y)$. The target variable $y \in \mathbb{D}_Y$ is always binary in this chapter and therefore in the domain $\mathbb{D}_Y = \{0,1\}$. Minority class samples are labeled $y = 1$.

In addition, the set $\mathscr{I} \subseteq \mathscr{S}$ references the set of all m**i**nority class samples: $\mathscr{I} = \{s \in \mathscr{S} \mid y = 1\}$. $\mathscr{A} \subseteq \mathscr{S}$ references the set of all m**a**jority class samples $\mathscr{A} = \{s \in \mathscr{S} \mid y = 0\}$. The number of samples in each set are denoted by $N_\mathscr{I}$ and $N_\mathscr{A}$, respectively. Furthermore, $rnd(k,l)$ returns an uniformly distributed random scalar r with $k \leq r \leq l$ which is either a rational ($\mathbb{Q}$, $rnd_\mathbb{Q}$) or a natural ($\mathbb{N}$, $rnd_\mathbb{N}$) number.

SMOTE

The Synthetic Minority Oversampling TEchnique (SMOTE) [29] combines minority oversampling as well as random majority class under–sampling (only over-sampling was performed in this chapter). As of March 2018, SMOTE was cited over 5500 times according to Google Scholar, making this algorithm the most popular algorithms for dealing with imbalanced data sets studied in this work.

To oversample minority class samples, the following steps are performed for each minority class sample until the data set is balanced[2]:

1. For any minority class sample $s_\mathscr{I} \in \mathscr{I}$, identify its $k = 5$ nearest (Euclidean distance) minority class neighbors[3] $s_{nn,k_i} = NN(s_\mathscr{I}) \in \mathscr{I}$ with $k_i = \{1,2,3,4,5\}$. Nominal features must be encoded as discrete features prior to applying SMOTE.

2. Calculate the Euclidean difference (Kernels are not supported by `imbalanced-learn`) from the minority class sample ($s_\mathscr{I}$) under consideration and one randomly chosen of the five nearest neighbors: $v = s - s_{nn,rnd_\mathbb{N}(1,5)}$.

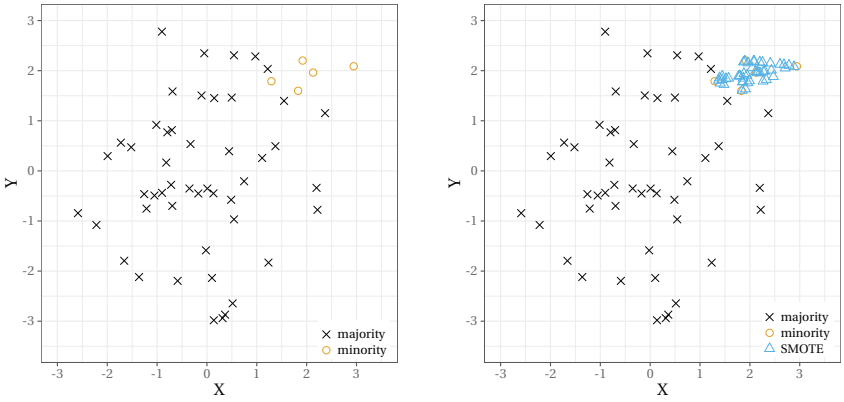
3. Multiply the resulting vector v by a random, scalar number between 0 and 1: $v_{rnd} = v \cdot rnd_\mathbb{Q}(0,1)$.

4. Add this randomized vector to the original sample $s_{SMOTEd} = s_\mathscr{I} + v_{rnd}$. This leads to a new, synthetically generated sample on the line segment between the original minority class sample $s_\mathscr{I}$ and its randomly chosen nearest neighbor $s_{nn,rnd_\mathbb{N}}$.

5. Add this artificial minority sample to the training set.

[2] If necessary, the target class ratio can be explicitly set.
[3] This value ($k = 5$) can be changed as well.

This causes the classifier to create "larger and less specific decision regions" [29], since the input space for the classifier is less sparse, resulting in better generalization. An example for $k = 4$ (since there are only 5 minority class samples), where the minority class of the original data set (Figure 4.1a) has been oversampled to match the number of majority class samples is shown in Figure 4.1b. Note, how SMOTE only generates samples (marked by a blue triangle) between original samples because of $rnd_{\mathbb{Q}}$ being bounded by 0 and 1 zo ensure that the minority class region is not enlarged by applying SMOTE.



(a) Original data set.

(b) Data set after applying SMOTE.

Figure 4.1: Comparison of the original data set before and after applying SMOTE.

ADASYN

ADaptive Synthetic Sampling Approach (ADASYN) [58] is another technique specifically designed for imbalanced learning. As of March 2018, it has been cited over 430 times according to Google Scholar. The key idea of ADASYN is to use density information as the heuristic to decide for every minority class sample how many synthetic samples need to be generated. ADASYN consists of the following steps:

1. Count the number of majority class neighbors $0 \leq \Delta_i \leq k$ for each minority class sample $s_{\mathscr{I}} \in \mathscr{I}$ among the $k = 5$ nearest neighbors.

2. Divide Δ_i by the number of nearest neighbors, k. This is repeated for all minority class samples $s_{\mathscr{I}} \in \mathscr{I}$ resulting in a $1 \times N_{\mathscr{I}}$ ratio vector $\mathbf{r}$.

3. Normalize this vector: $\hat{\mathbf{r}} = \mathbf{r} / \sum_{i=1}^{N_{\mathscr{I}}} r_i$.

4. The number of synthetic samples for a sample $s_{\mathscr{I}}$ are given by multiplying the corresponding entry from $\hat{\mathbf{r}}_i$ by the total number of necessary synthetic samples $g = N_{\mathscr{A}} - N_{\mathscr{I}}$ to generate a balanced data set. Minimal imbalances due to rounding errors of the multiplication ($\hat{\mathbf{r}}_i \cdot g$) are tolerated. Samples are generated the same way as described in Section 4.1.1 (SMOTE).

In comparison to applying no sampling at all this not only leads to a more balanced data set, but also forces the classifier to focus on regions that are hard to classify. This is shown in Figure 4.2b: ADASYN creates more samples close to the presumed decision boundary compared to SMOTE, where the created sampled are randomly spread in the minority class region (compare Figure 4.1b).

With one exception, the generation of synthetic samples when using ADASYN is equal to SMOTE, as described above (Section 4.1.1): The only difference between the used SMOTE and ADASYN implementations is that ADASYN softens the condition for the choice of the $k = 5$ nearest neighbors for generating the vector v: While SMOTE requires all of them to be samples of the minority class, ADASYN utilizes samples of both, the minority- and majority-class to synthesize minority-class samples. As depicted in Figure 4.2b this causes new samples not only to be generated on line segments between minority samples, but also "outside" the minority-class sample region (between minority and majority samples).

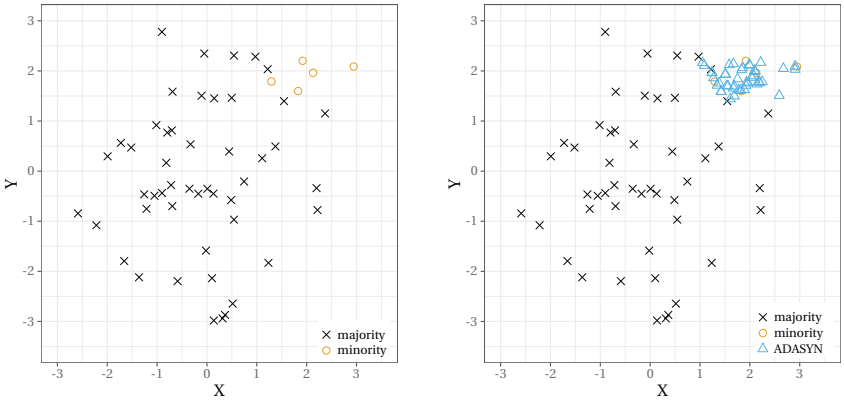
Tomek Links

The concept of Tomek links [124] (cited 633 times according to Google Scholar as of March 2018) aims to remove noisy and/or borderline samples from a given data set and therefore to clarify the border between classes. A "Tomek link" consists of two samples s_1 and s_2 that belong to different classes and are each other's nearest neighbor according to the Euclidean distance. Nominal features must be encoded as discrete features priorly.

To identify and remove samples that are part of Tomek links, the following steps are performed:

1. For each sample $s \in \mathscr{S}$, identify its nearest neighbor $s_{nn} = NN(s) \in \mathscr{S}$.

2. Loop over all samples $\mathscr{S}$ once. If the sample s is a member of the minority class ($s \in \mathscr{I}$), skip this item. If it is a majority class sample ($s \in \mathscr{A}$), and its closest neighbor is a



(a) Original data set.

(b) Data set after applying ADASYN.

Figure 4.2: Comparison of the original data set before and after applying ADASYN.

minority class sample ($s_{nn} \in \mathscr{I}$), these two samples form a Tomek link. In this case, the majority class sample under consideration is stored in a list $\mathscr{L}$.

3. The final training set $\mathscr{T}$ is created by removing all majority samples that are part of a Tomek link from $\mathscr{S}$: $\mathscr{T} = \{s_o \in \mathscr{S} \setminus \mathscr{L}\}$. These steps are only executed once.

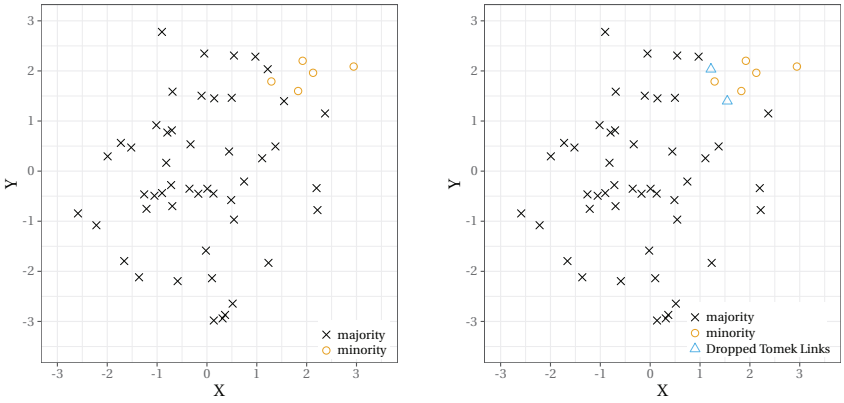
The identification of Tomek links is parameter free. As Figure 4.3b shows, two majority class samples were part of a Tomek link and have been dropped.

CNN

The Condensed Nearest Neighbor (CNN) [56] rule aims to identify a subset of samples which is a *consistent* subset $\mathscr{C} \subseteq \mathscr{S}$ of the original data set $\mathscr{S}$. A subset is called *consistent* if, when being used by an 1-NN (one nearest neighbor) classifier as stored reference, all other samples (non-stored samples) are classified correctly. In Figure 4.4b, all non removed samples form the stored reference. If these are used as the training set for a 1-NN classifier, all removed samples will be classified correctly. This technique has been cited over 1900 times according to Google Scholar as of March 2018.

CNN uses a greedy approach to identify one possible consistent subset:

1. Move a random sample s from $\mathscr{S}$ to $\mathscr{C} = \{s\}$, while the grab bag is initially empty $\mathscr{G} = \emptyset$. This results into a deletion from $\mathscr{S}$.



(a) Original data set.

(b) Tomek links removed.

Figure 4.3: Comparison of the original data set before and after the removal of Tomek links.

2. The subsequent, random sample from $\mathscr{S}$, $s = (\mathbf{x}, y)$ is then classified using the 1NN rule given the element(s) in $\mathscr{S}$ yielding the predicted class y_{pred}.

3. If the classification is correct ($y_{pred} = y$), the sample is moved from $\mathscr{S}$ to $\mathscr{G}$. If not, it is moved into $\mathscr{C}$. This way it is ensured, that this sample would be correctly classified on the next iteration.

4. Steps 2 – 3 are repeated for all remaining samples $s \in \mathscr{S}$.

5. After one pass through $\mathscr{S}$, all samples in $\mathscr{G}$ are classified using the elements in $\mathscr{C}$.

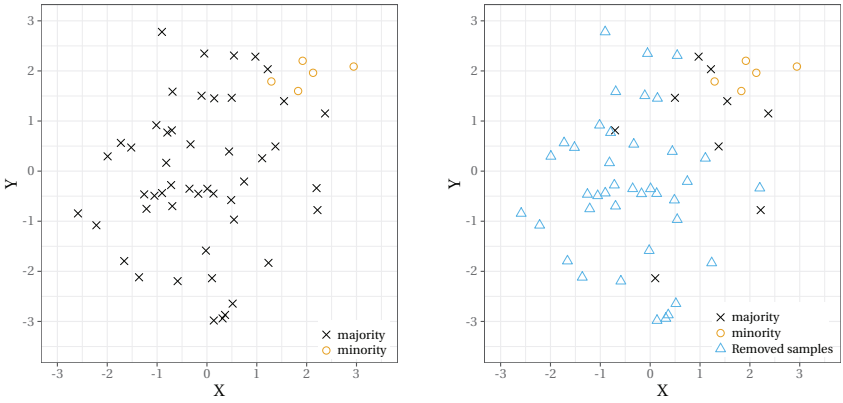
6. If all samples are classified correctly, the algorithm terminates. If not, all misclassified samples will be moved from $\mathscr{G}$ to $\mathscr{C}$ and step 5 is repeated.

7. The consistent subset $\mathscr{C}$ is used as the training set and represents the output of this algorithm.

The consistent subset $\mathscr{C}$ resulting from applying CNN is not unique: For every data set, multiple consistent subsets exist. One possible solution is depicted in Figure 4.4b.

OSS

One-Sided Selection (OSS) [74] (146 citations according to Google Scholar as of March 2018) also aims at creating a consistent subset of the original data set $\mathscr{C} \subseteq \mathscr{S}$. OSS picks up the



(a) Original data set.

(b) Consistent subset (CNN).

Figure 4.4: Comparison of the original data set and the consistent subset according to CNN.

idea of CNN, but introduces several, major modifications:

1. First, instead of randomly picking *one* sample to serve as stored reference $\mathscr{C}$, OSS has been specifically designed to work with imbalanced data sets. Therefore, OSS uses *all* minority class samples, since they are "too rare to be wasted" [74], and selects *one* majority class sample for the initial store set $\mathscr{C} = \mathscr{I} \cup \{s_a \in \mathscr{A} | a = rnd_{\mathbb{N}}(0, N_{\mathscr{A}} - 1)\}$.

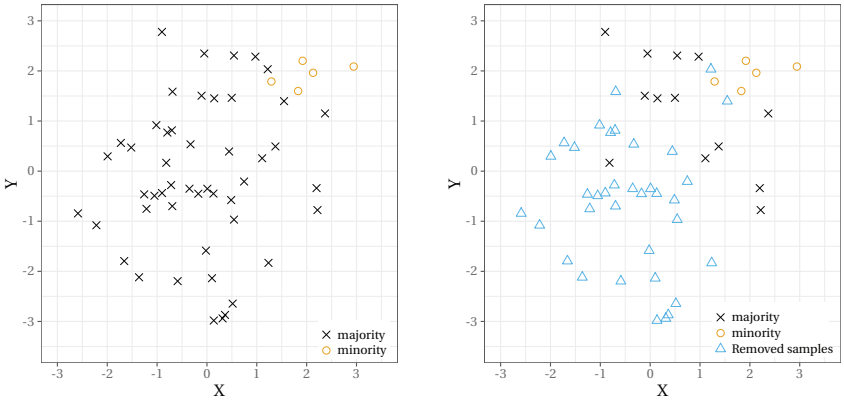
2. All remaining samples from $\mathscr{S}$ are then classified *once* (in contrast to CNN, where multiple loops are possible) using $\mathscr{C}$ as stored reference.

3. Misclassified (majority) samples are *added* (compare CNN: moved[4]) to $\mathscr{C}$. $\mathscr{C} \subseteq \mathscr{S}$ is now a consistent subset.

4. The last step is to remove all majority class samples from $\mathscr{S}$, that are part of a Tomek link.

This results in a consistent subset at less computational cost compared to CNN due to a reduced number of loops and therefore less classifications, see Figure 4.5b for a visualization.

[4] Given three features, one target, and 10000 samples, copying outperformed moving by a factor of 2.5 in `Python`.



(a) Original data set.

(b) Consistent subset (OSS).

Figure 4.5: Comparison of the original data set and the consistent subset according to OSS.

Class Sensitive Scaling

In addition to the existing techniques, CSS is proposed which was implemented in `Python`: It is based on the fact that areas with samples of different classes overlap in most, especially imbalanced, scenarios. The key idea is to move samples into the direction of the corresponding class center before training to enable a less complex (e.g. a lower depth when using a random forest as classifier) decision boundary, similar to the concept of Tomek links (Section 4.1.1). This is less computationally expensive compared to down-sampling techniques involving the application of simple models, such as a k-NN. Both ways, the identification of a decision boundary is eased.

CSS can be parameterized to scale either only one class (minority or majority), or both classes (referred by the hyperparameter "target"). Also, different scaling "modes" can be set: In the *constant* mode, all samples will be scaled by the same amount, set by a scaling constant c. In the *linear* mode, the amount increases with the distance to the feature-specific class center. Samples, that are one empirical standard deviation 1σ away from the corresponding class center μ will be scaled with the amount c, samples that are 2σ away will be scaled with $2c$, etc.

To avoid excessively optimistic classification performance during CV, CSS must be *only* applied to the training folds, while the validation fold (and the test data) must remain un-

modified. An explanation how constant scaling of the majority class works is given below:

1. Calculate the feature specific mean $\mu_{1,\mathscr{A}}, \mu_{2,\mathscr{A}}, ..., \mu_{P,\mathscr{A}}$ for all features $X_1, X_2, ..., X_P$ using *majority* class samples $s_A \in \mathscr{A}$ only. Since the averages of features need to be calculated, this technique requires – like all of the aforementioned techniques – discrete or continuous features.

2. Scale all feature values for all majority class samples, e.g. the new value of feature i from majority class sample $s_j = ([x_{j,1}, x_{j,2}, ..., x_{j,P}], y_j)$ is calculated using: $x_{j,i,scaled} = x_{j,i} \cdot (1-c) + \mu_{i,\mathscr{A}} \cdot c$. This yields scaled majority samples $s_{\mathscr{A},scaled} \in \mathscr{A}_{scaled}$.

3. Merge the scaled majority class samples $\mathscr{A}_{scaled}$ and the original minority class samples $\mathscr{I}$ into the new training set $\mathscr{T}_{scaled} = \mathscr{A}_{scaled} \cup \mathscr{I}$.

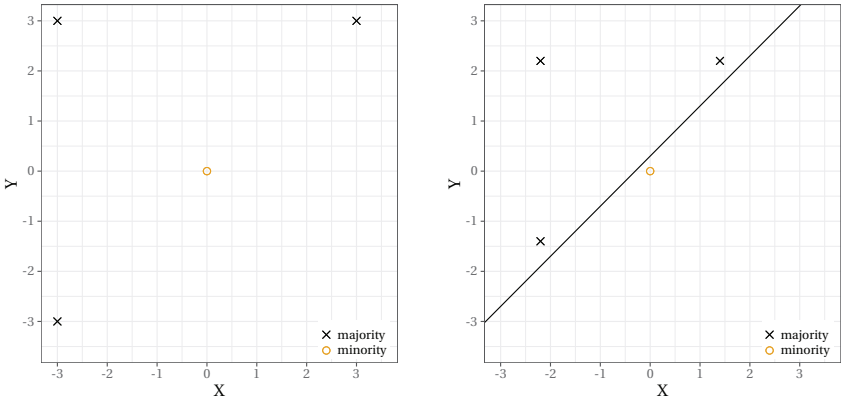
Figure 4.6b shows a synthetic data set with 4 samples. On the left, the unscaled data set is displayed. It is not possible to draw a decision boundary as straight line in the feature space, linearly separating all samples without misclassification. After applying CSS to each of the majority samples, a linear solution becomes obvious. With the assumption that more samples of class 0 will reside in the lower right corner, the pictured line would work just fine. A classifier trained on the unscaled data set (pictured left), would likely be overfit, e.g., using a second order polynomial as the decision boundary.

This concept is transferable to real data sets as well. Figure 4.7b displays the distribution of the vowel data set (Section 4.3) for class “1” after dimensions have been reduced by PCA to X and Y. In the unscaled data set classes “0” and “1” overlap more, presumably causing the classifier to overfit or misclassify. After scaling the majority class (Figure 4.7b, right), an easier solution to the classification problem becomes visible.

Since this technique is an entirely new approach towards dealing with NIDs (see Section 4.4), all 84 combinations of scaling modes (linear and constant), scaled targets (minority, majority, both) and 14 scaling constants $c = [0, 1]$, were simulated.

4.1.2 Objective Feature Noise, Borderline, and Overlap Measure

To measure the “noisiness” of the data set, the feature noise, borderline, and overlap measure (BS^2) is proposed based on Schlegel and Sick [108]. It does not require a model of the class-conditional densities: The BS^2 measure is calculated completely parameter free based



(a) Original data set.

(b) Scaled data set (CSS).

Figure 4.6: Demonstration of CSS with artificial data.

on the number of Tomek links [124] (Section 4.1.1) in relation to the number of minority class samples. The BS^2 measure is inherently normalized, since a value of 1 would mean *all* minority samples are part of a Tomek link and therefore noisy, borderline, or part of two overlapping distributions. BS^2 is calculated as follows:

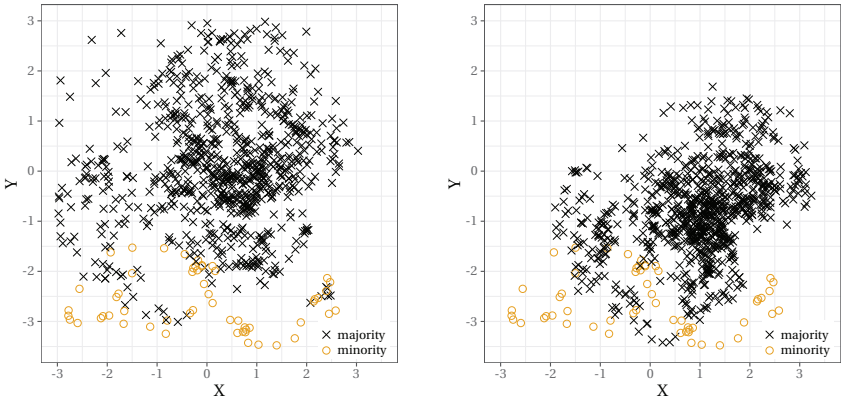
1. Count the number of all N and the minority $N_{\mathscr{S}}$ samples in the original data set $\mathscr{S}$.

2. Remove all majority class samples participating in a Tomek link, as described in Section 4.1.1. This results into a subset of samples $\mathscr{S}_{sub} \subset \mathscr{S}$.

3. Count the number of samples $N_{\mathscr{S},sub}$ in $\mathscr{S}_{sub}$. This leads to $N_{tomek} = N - N_{\mathscr{S}_{sub}}$ Tomek links.

4. Calculate the measure using $BS^2 = \frac{N_{tomek}}{N_{\mathscr{S}}}$.

The noisiness-overlap measure BS^2 complements the balance ratio, which is given by the class ratio $N_{minority}/N_{majority}$, the total number of samples, and the number of features to assess the nature of a data set, as laid out in Section 4.3.



(a) Original data set.

(b) Data set with majority class scaled by CSS.

Figure 4.7: CSS on Vowel data set.

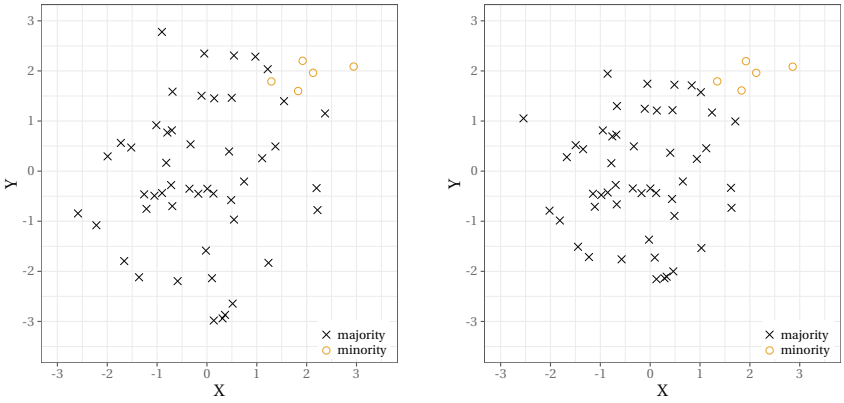
4.2 Research Demand

Many of the *foundational techniques* addressing NIDs presented in Section 4.1.1 have been modified, enhanced, or combined (e.g. [55, 5], and [30] in comparison to [29]). Yet, this chapter will focus on the investigation of the most popular, foundational PTs, because

(1) The improvements achieved by "derived" techniques reside in the lower single-digit percentage range and are highly dependent on the evaluated data set. Take, e.g., Barua et al. [5], who proposed a modification of SMOTE proposed by Chawla et al. [29]: Averaged over all evaluated data sets with a single Neural Network, their approach outperforms the unmodified one by only 1.14%.

(2) Also, many authors present opposing statements: Han et al. [55] argue for focusing on borderline samples (since being hard to classify), while Kubat et al. [74] and Batista et al. [6] propose discarding borderline samples (since being unreliable and noisy)[5]. Another example is Weiss [138] and Chawla et al. [28] who argue for and against sampling: Weiss [138] proposes "one should use all available training data, regardless of the resulting distribution of classes or cases". This way, "no information is lost". The opposite is stated by Chawla et al. [28]: "Clever re-sampling [...] methods [...] can provide new information or eliminate redundant information". In the following, techniques that focus on borderline samples (compare

[5]One could state that the evaluated data sets were too different, but the range of attributes (Han et al.: 2 – 36, Kubat et al.: 10 – 44) and samples (Han et al.: 306 – 1600, Kubat et al.: 38 – 990) clearly overlap.



(a) Original data set.

(b) Scaled data set (CSS).

Figure 4.8: Comparison of the original data set and scaled data set.

ADASYN, Section 4.1.1) and remove borderline samples (compare Tomek, Section 4.1.1 as well as OSS, Section 4.1.1) are compared.

(3) Avoidance of cross-influences between techniques to enable an isolated performance comparison of the foundational techniques, respectively.

(4) Combining PTs inevitably leads to a higher computational complexity that might restrict the applicability of the respective algorithm.

Therefore, this chapter will objectively assess classification grade improvements *and* computational complexity of various foundational PTs while considering interdependencies on various classifiers (such as LR, RF, etc.) in a data set agnostic fashion to ensure transferability to other data sets and scenarios.

4.3 Data Sets

Table 4.1 gives an overview of the data sets forming the testbed. Entries are sorted by their balance ratio. All data sets except for "Automotive" are publicly available on the UCI machine learning repository [77] or elsewhere as specified to enable other researchers to compare their results to ours. To generate the most imbalanced scenario, all classifiers are trained in a one versus all fashion. If a data set offers more than two classes, and the number of available samples for the respective minority classes vary within a data set, the "minority"

Table 4.1: Overview of the evaluated data sets.

Data Set	Features	Samples	Minority	Classes	BS^2	Ratio
Pima	8	768	268	2	0.205	0.54
Phoneme	5	5404	1586	2	0.062	0.42
Vehicle*	18	846	216	4	0.154	0.34
Glass*	10	214	38	6	0.080	0.21
Satimage*	36	3998	666	6	0	0.20
Vowel*	14	990	90	11	0	0.10
Abalone	8	731	42	2	0.095	0.06
Forest	12	517	24	2	0.583	0.05
Mammography	6	11182	260	2	0.154	0.02
Automotive*	100	4949**	48	15	0.423	0.01

count was averaged. These data sets are marked with an asterisk (*). Downsampled datasets are marked with two asterisks (**). All data sets have been normalized prior to modeling. Unless otherwise specified, a 70% : 30% training-test-split was used. Model hyperparameters are optimized using a five fold CV based on the training data.

The Pima Indian Diabetes Data set. The target of the "pima" two-class data set [77] is to predict diabetes. It has a total of 768 samples and eight features. There are 268 samples with no diabetes present, which is equivalent to an balance ratio of $268 : 500 = 0.54$.

Phoneme Data set. The "phoneme"[6] data set consists of five features to distinguish between 3818 nasal and 1586 oral sounds. The balance ratio is therefore $1586 : 3818 = 0.42$.

Vehicle Silhouette Data set. The purpose of the "vehicle" [117] data set is to classify the type of vehicle among 4 classes based on a set of 18 features extracted from the silhouette. The data set holds 846 samples. The averaged balance ratio is 0.34.

Glass Data set. The "glass" data set consists of 214 measurements with ten features each. The samples fall into six different classes (glass types). The features indicate the chemical composition. Each feature represents one element, e.g. magnesium, silicon, or aluminum. The averaged number of minority samples is 38, resulting in a $38 : 178 = 0.22$ balance ratio.

Statlog Landsat Satellite Data set. This data set includes 3998 samples with 36 features representing spectral values. The target is to classify the type of surface "displayed" in the corresponding sample. The averaged balance ratio is 0.2.

Vowels Data set. The "vowel" data set [125] holds 990 samples of 11 vowels. Each vowel

[6] https://www.elen.ucl.ac.be/neural-nets/Research/Projects/ELENA/databases/REAL/phoneme/

was spoken six times by each of 15 speakers. Features include – aside from sex and a speaker ID (which was not used in this article) – spectral data. The train-test-split is predefined in this data set: 528 samples form the training set, 462 are left for testing (53.33% : 46.66%). The balance ratio is $90 : 990 = 0.\overline{09} \approx 0.1$.

Abalone Data set. This data set consists of various physical measurements such as, e.g., length, height, or shell diameter, taken from abalone slugs. The target is to predict the age. As suggested by Guo et al. [52] and adopted by the authors of ADASYN [58], the class "18" was chosen as the minority class and class "9" as the majority class. This results into an balance ratio of $42 : 689 = 0.06$.

Forest Fires Data set. The "forest" data set [32] holds samples of various wooded areas, likely to catch fire. The target is to predict how much of the area was burnt down. Every forest is identified by X and Y coordinates and is usually observed more than once. Features include several weather and humidity-based metrics. In order to be able to classify the data set, the target target variable *burned area* was binarized using

$$area = \begin{cases} 1, & \text{if } area \geq 50 \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

This results in 493 majority class and 24 minority class samples. The balance ratio is therefore $24 : 493 = 0.05$.

Mammography Data set. The *mammography* data set [140] holds 11182 samples, representing digitized micro-calcifications in mammogram images. The target is to classify whether the pixel is cancerous (260 samples) or not (10922 samples).

Automotive Data set. The "Automotive" data set was collected from hybrid cars and consists of non-personal data only. Python and `sklearn` [91] were not able to cope with 5000 dimensions and over 100000 samples on the available hardware (for details see Section 4.4). Therefore, the majority class was randomly sampled down to an $100 : 1 = 0.01$ balance ratio prior to the following experiments. Also, the number of dimensions was reduced to 100 using PCA. This shows, that a sophisticated feature reduction technique such as the one proposed in Chapter 3 is not only helpful but also a necessity to be able to work with larger data sets of this kind. Although the data set contains more than 3000 potential classes, 15 classes that consist of at least 21 and no more than 155 minority class observations were used for the following evaluation. Five targets of each category were modeled: switched parts, taken

counter actions, and diagnostic trouble codes (indicating a certain fault in the car). The classifiers are trained in an one-versus-rest fashion. E.g., a certain part needs to be modeled: All samples where this part was replaced will be marked as the positive (minority class). All other samples – if a different or no part was switched – will be marked negative / majority.

4.4 Evaluation

To ensure that the results presented in this chapter are not biased by the used classifier, the following experiments are performed on a variety of different classifiers. Model hyperparameters are optimized with regards to the auPRC using a five-fold CV. The used classification algorithms are:

k-NN [8]: The k-NN (Section 2.2.3) was implemented using $k = \{3, 5, 10\}$ neighbors.

LR and Weighted Logistic Regression (WLR) [8]: Both techniques (Section 2.2.2) were optimized regarding their cost ($C = \{0.001, 0.01, 0.1, 1\}$) using a L1 regularized implementation provided by `liblinear`. Between LR and WLR is only one difference: In case of WLR, the weights were chosen to be inversely proportional to class frequencies.

OCC: The used implementation is based on the SVM concept proposed by Schölkopf et al. [110]. The following parameters were optimized to learn the minority class: the used kernel (linear, poly using a degree of three, and sigmoid), the kernel coefficient $\gamma = \{0.001, 0.01, 0.1, 1\}$ for poly and sigmoid kernel as well as the cost parameter $\nu = \{0.001, 0.1, 0.5, 0.75, 1\}$.

RF [19] and Weighted Random Forest (WRF): The number of estimators was optimized ($n_{est} = \{5, 10, 20\}$) as well as the splitting criterion (using Gini impurity or entropy). In case of WRF, the weights were chosen to be inversely proportional to class frequencies (similar to LR and WLR this is the only difference). More details on RFs can be found in Section 2.2.1.

All experiments have been conducted on an HP™ Z-840 equipped with two Intel® Xeon® E5-2640 v3 2.60GHz CPUs and 96GB of RAM. A total number of 122700 models has been evaluated to substantiate the results presented in the following. If the model supported multicore, all available threads were used. This was the case for k-NN, RF, and LR.

Since CSS is an entirely new technique, meaningful default parameters for this technique were identified in advance (Section 4.4.1), similar to, e.g., SMOTE (where $k = 5$ nearest neighbors is the default setting). Also, to avoid that all interesting results vanish by averaging over

all data sets, they were clustered into three clusters based on the information given in Table 4.1. The results from Section 4.4.1 will be used in all following sections, where the actual comparison between all PTs introduced in Section 4.1.1 and a "naive" approach, where no (pre-)processing takes place, is carried out.

Combinations of classifiers and PTs are compared with respect to three criteria: Robustness (Section 4.4.2), computational complexity (section 4.4.4), and classification performance in terms of primarily auPRC (Section 4.4.3). A PT-classifier combination is referred to "robust", if the performance in terms of auPRC, auROC, or F_1 calculated during training using CV is comparable to the test performance.

4.4.1 Preliminary Investigations

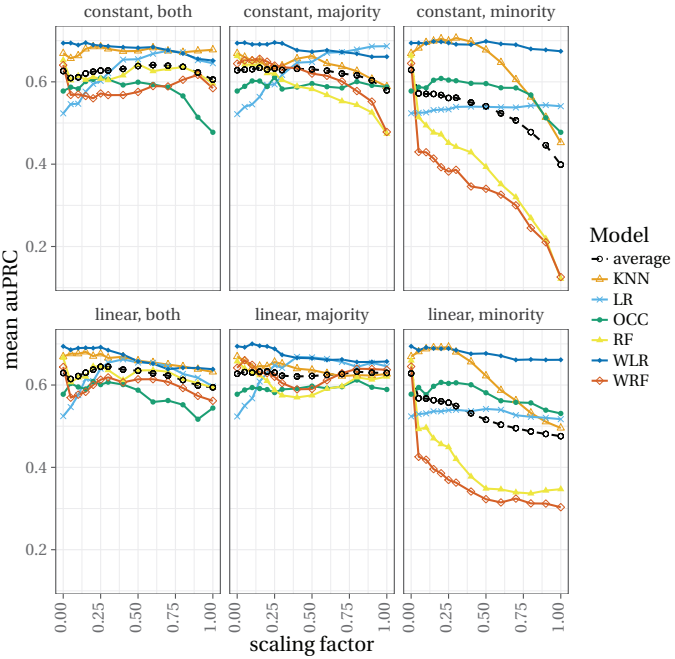


Figure 4.9: Comparison of different CSS modes based on auPRC calculated during CV.

Figure 4.9 shows the mean auPRC during training averaged over all models built with the different CSS modes. Several interesting observations should be noted.

First, scaling only the minority class is generally (on average) not advisable (see the black dashed line). This can also be observed in Figure 4.10, where the achieved auPRC values are averaged across all modeltypes and plotted in a single chart. However, a scaled minority (scaling factor $s \approx 0.2$) class in combination with a k-NN classifier can yield very promising results. A possible explanation for the degrading classification performance when scaling only the minority class is presumably a even smaller minority class feature region after scaling.

Second, best results are achieved according to Figure 4.9 and Figure 4.10 when applying constant scaling of both classes with $c \approx 0.625$ or linear scaling on classes with a scaling factor of $c = 0.25$. In this chapter, the latter configuration will be used due to slightly higher auPRC values and a less distorted feature space that is closer to the unscaled one for smaller values of c. The only exception is the k-NN classifier, where constant minority scaling with $c = 0.30$ was used.

Third, excluding minority class scaling, CSS is able to improve the auPRC performance compared to the "naive" scenario ($c = 0$, Figure 4.10 the leftmost point) in all cases on average. According to Figure 4.9, each of the examined classifiers' performance in terms of auPRC is increased for some c when CSS is applied.

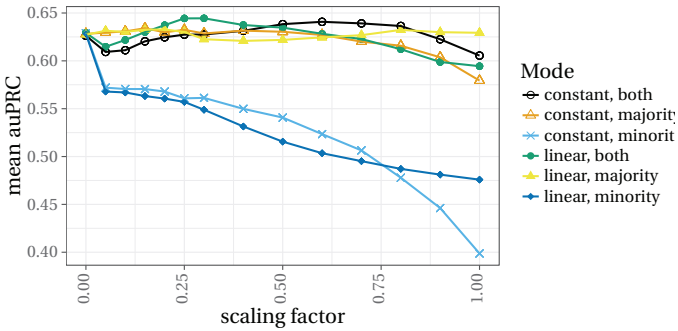


Figure 4.10: Comparison of different CSS modes based on auPRC averaged across all classifiers.

To avoid that all interesting insights vanish by averaging, the data sets have been clustered into three clusters. This not only allows for better visualization, but also enables to derive generally applicable, yet tailored recommendations in the final conclusion given in Section 4.5. The data sets (as shown in Table 4.1) have been clustered manually. The columns

"Features" (number of features), "Samples" (the number of samples), the "BS^2" measure, and "Ratio" (the balance ratio) were reduced to two principal components. The result is shown in Figure 4.11. If the data set holds multiple classes with differing properties (e.g. varying "Ratio"), every class is plotted individually. The clusters used in the following are:

- Cluster 1: low number of features ($5-18$), low number of samples ($214-5404$), low BS^2 ($0-0.27$), high balance ($0.04-0.55$).
- Cluster 2: high number of features ($11-101$), medium number of samples ($517-6868$), high BS^2 ($0.12-0.58$), medium balance ($0.01-0.05$).
- Cluster 3: high number of features ($6-101$), high number of samples ($10302-15655$), low BS^2 ($0.13-0.25$), low balance ($0.01-0.02$).

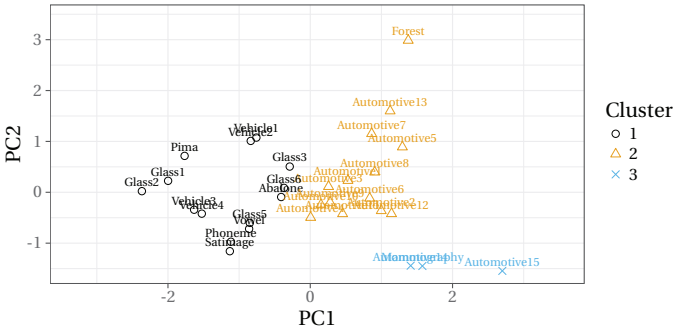
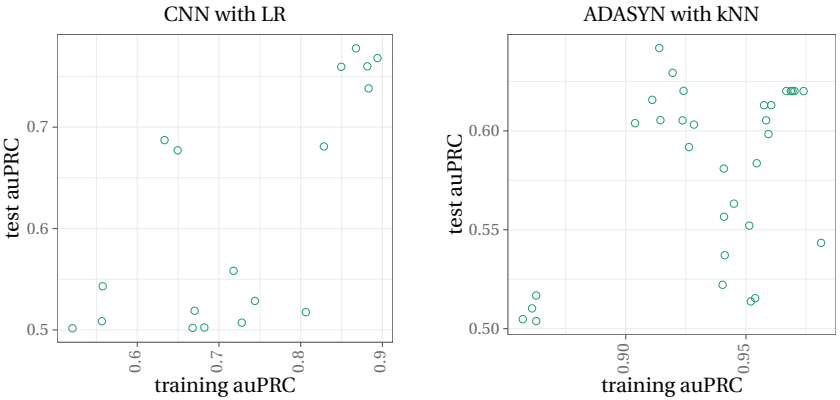


Figure 4.11: Clustering of data sets.

4.4.2 Robustness

To assess the robustness of PT–classifier combinations, the auPRC achieved during training is compared to the auPRC resulting from the test set. If the test auPRC differs too much from the auPRC calculated during CV-based training, model hyperparameters chosen during CV will not lead to well generalizing models.

An example is given in Figure 4.12b: It is impossible to determine which model generalizes well on the test set based on the overly optimistic training auPRCs. On the other hand, Figure 4.12a displays a PT–model pair, where all hyperparameter sets which resulted into a high training auPRC also yielded the best performance on the test set.



(a) Robust combination.

(b) Non–robust combination.

Figure 4.12: Comparison of a robust and non–robust PT-classifier combination.

To objectively assess the concordance of training and test auPRC, the Root-Mean-Square Difference (RMSD) will be used, which is given by the following formula:

$$RMSD = \sqrt{\frac{\sum_{t=1}^{T}(auPRC_{train} - auPRC_{test})^2}{T}}, \text{ with} \tag{4.2}$$

T referring the number of testing and training pairs, $auPRC_{train}$ reflecting the auPRC achieved on the test folds during CV based training, and $auPRC_{test}$ referring the auPRC achieved on the test set after training was finished.

Other approaches such as the correlation or fitting a linear regression and evaluating the slope to assess the concordance have been evaluated, but yielded counterintuitive results.

Table 4.2 (Table 4.3) summarizes the RMSD between the training and test auPRC (auROC) for each PT-classifier combination. The higher the RMSD value, the stronger the cell is colored in orange. Standard deviation and mean for Table 4.2 are: $\sigma_{RMSD} = 0.124$ and mean $\mu_{RMSD} = 0.361$. Especially when considering the auPRC, several noteworthy things surface:

First, ADASYN and SMOTE tend to cause high RMSD values. These are the (only) two techniques, that sample up by creating artificial samples. An explanation for the high RMSD value might be the simple way samples are generated, which yields an overly optimistic training auPRC. This theory is backed up by Figure 4.13, where all training and testing auPRCs

Table 4.2: RMSD of training and test auPRC.

	ADASYN	CNN	CSS	naive	OSS	SMOTE	tomek
KNN	0.552	0.167	0.238	0.259	0.285	0.505	0.283
LR	0.545	0.305	0.384	0.310	0.359	0.567	0.356
OCC	0.359	0.187	0.245	0.205	0.264	0.374	0.272
RF	0.536	0.251	0.304	0.222	0.305	0.518	0.302
WLR	0.542	0.328	0.407	0.432	0.488	0.571	0.495
WRF	0.571	0.260	0.266	0.229	0.299	0.510	0.298

for SMOTE and CNN (which scored the lowest RMSD scores in Table 4.2) are shown. While the CNN circles are forming a linear relationship, a large amount of SMOTE crosses are located at the very right. In case of SMOTE, this makes it impossible to transfer conclusions regarding hyper-parameters from the training onto the testset.

Second, CNN yields the lowest RMSD values. This is an interesting result, since even the "naive" approach, where no preprocessing takes place, yields higher RMSD values. CSS, OSS, and Tomek links yield comparable results, with slightly lower RMSD scores by CSS.

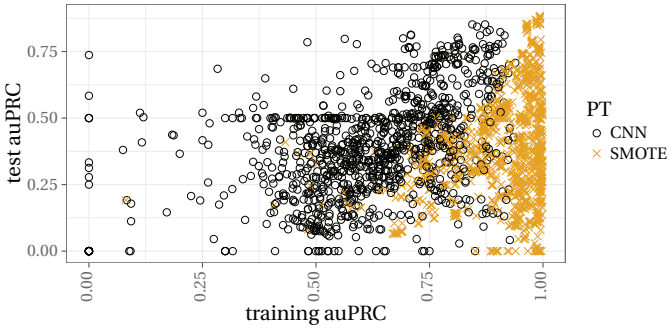


Figure 4.13: Test versus train auPRC of SMOTE and CNN.

4.4.3 Influence of Preprocessing Techniques on the Classification Performance

Figure 4.14 shows boxplots of the achieved auPRC performance depending on the cluster, PT, and used classifier. The rightmost column shows the performance for each classifier averaged across all clusters. The bottom row shows the results averaged across all classifiers.

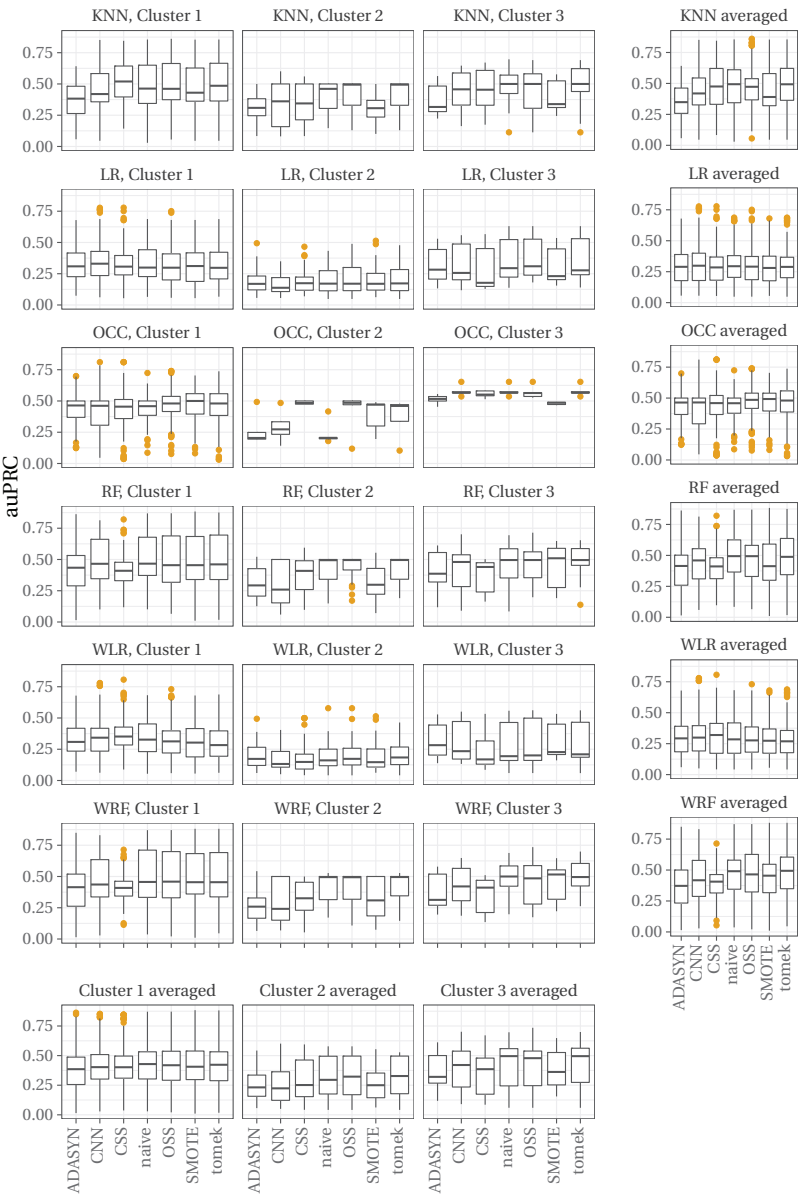


Figure 4.14: Achieved auPRC based on the data set cluster, PT, and classifier.

Table 4.3: RMSD of training and test auROC.

	ADASYN	CNN	CSS	naive	OSS	SMOTE	tomek
KNN	0.433	0.498	0.325	0.311	0.267	0.613	0.257
LR	0.466	0.583	0.452	0.472	0.429	0.481	0.426
OCC	0.318	0.387	0.211	0.218	0.195	0.321	0.195
RF	0.315	0.467	0.283	0.261	0.235	0.398	0.225
WLR	0.466	0.553	0.409	0.428	0.404	0.478	0.397
WRF	0.313	0.475	0.301	0.282	0.254	0.395	0.242

The following discussion will be, if not differently noted, based on median values of the auPRC scorings (referred as “performance”).

k-NN performance is below the naive approach[7], when ADASYN or CNN are applied. CSS, as well as Tomek increase performance on cluster 1. On cluster 2, Tomek and OSS yield the best results, while all other techniques perform significantly worse than the naive approach. This trend can also be observed on cluster 3, where OSS and Tomek techniques yield comparing results (Tomek is yielding the same median, but more consistently high results). To summarize: k-NN performs best when combined with the Tomek technique. All other techniques do not improve the performance.

LR performs on average lower compared to k-NN, independent on the used PT. The impact of using any PT is extremely low (if measurable at all). The only exceptions are CSS and SMOTE on cluster 3, that actually lower the performance.

OCC performance on cluster 1 is mostly unaffected, only SMOTE causes a slight increase. Results from cluster 2 suggest that applying CSS, OSS, SMOTE, or Tomek improves the performance. CSS and OSS should be preferred in this case, since the results are more consistent. On cluster 3, all techniques yield similar results, with an exception being SMOTE and ADASYN, which score slightly lower.

Performance of an RF is not increased by any PT. In fact, the opposite is true: Most PTs decrease the performance. This especially holds true for CSS, SMOTE, and ADASYN.

WLR performance can be foremost increased by CSS and CNN on cluster 1. However, these techniques tend to decrease the performance slightly on cluster 2, where Tomek and OSS increase the performance. On cluster 3, ADASYN, OSS, and SMOTE tend to increase the performance. On average, CSS is the only technique yielding improved results, however the

[7] “Naive” means that no PT is applied.

other techniques don not do any harm regarding the auPRC.

WRF works great out of the box without any PT applied. This technique not only yields the best results compared to all other classifiers compared in this chapter but also works well with unprocessed data sets (naive). Tomek is the only technique that yields a tiny advantage on one cluster (cluster 3), on every other cluster the performance is not improved by any PT.

To summarize the results in dependence on the cluster: On cluster one (low number of samples, low BS^2), no PT is able to yield consistently better results. Tomek is the only technique that does not noticeably decrease the performance. On cluster 2 (high BS^2, medium balance), OSS and Tomek yield an increased performance, while other techniques decrease the performance. Cluster 3, although being different in terms of the number of samples yields the same insights as cluster 2: Tomek and OSS are the only techniques that should be considered.

The following insights were gained in dependence on the classifier: The RF and WRF performance cannot be increased, they already yield the best classification performance in terms of auPRC without any PT being used. While the performance of the LR is unaffected, WLR can profit from applying CSS before training. An OCC can profit from any technique, the most beneficial being SMOTE according to this study. k-NN can not be improved much, but at least CSS and Tomek do not decrease performance.

Unlike concluded by Schlegel and Sick [108] the naive approach is *not always* outperformed when the performance is *not* measured in auROC but auPRC, and when hyperparameter tuning during CV is also performed using auPRC measure. On the contrary: Only two combinations tend to improve performance: WLR in combination with CSS and OCC combined with SMOTE. A possible explanation for this may be, that the auPRC is the more suitable measure in imbalanced scenarios as elaborated in Section 2.3 with less room for improvements. Therefore, the well-founded selection of the measure to assess model performance and to select hyperparameters (such as auPRC in the given scenario, see Section 2.3) should be preferred over using a PT. This especially holds true when computational complexity (as laid out in below, Section 4.4.4) is taken into consideration.

The following examines which PT can be statistically proven to be effective. The Student's t-test has been chosen using a significance level of $\alpha = 0.05$. The t-test is possible since all of the following requirements [17] are met: The error scores are ordinally scaled, the scores are unrelated, the score populations have approximately normal distribution (see e.g. Fig-

Table 4.4: Statistical comparison of the cluster mean F_1 from PTs compared to the naive approach based on the p–value.

Method	Cluster 1	Cluster 2	Cluster 3
CSS	1.000	1.000	1.000
SMOTE	0.014	0.999	0.653
ADASYN	0.287	0.999	0.968
OSS	0.175	0.578	0.430
CNN	0.968	0.991	0.128
Tomek	0.105	0.535	0.241

ure 4.15), and the populations have approximately equal variance.

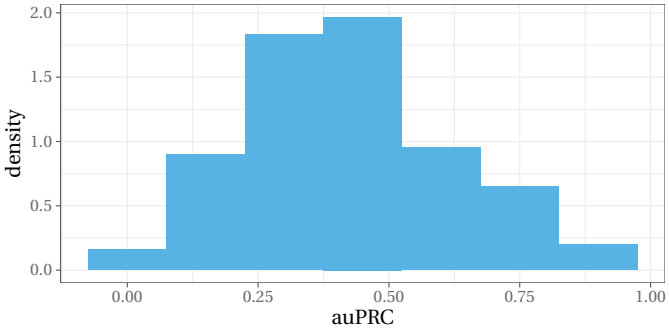


Figure 4.15: Distributions of the tested error scores.

The null hypothesis is, that the model performance (assessed in auPRC, auROC, and F_1) after a PT was applied to the data set is less or equal to the performance when no pre-processing took places (*naive*), $H_0 : \mu_{PT} \leq \mu_{naive}$. The alternative hypothesis is therefore $H_1 : \mu_{PT} > \mu_{naive}$.

The resulting *p-value*s (p) for each cluster are given in Table 4.4, Table 4.5, and Table 4.6 for all three measures, respectively. The hyperparameters have been tuned using auPRC in all three cases. If the *p-value*[8] is smaller than the chosen significance level $\alpha = 0.05$, H_0 can rejected. The cell is colored green in this case. This means that the PT indeed improves the performance measured in the corresponding method.

The following results emerged:

[8]The p was generated using the function `t.test(auc_array1, auc_array2, alternative = "greater")` available in R.

Table 4.5: Statistical comparison of the cluster mean auROC from PTs compared to the naive approach based on the p–value.

Method	Cluster 1	Cluster 2	Cluster 3
CSS	0.873	0.930	0.017
SMOTE	0.000	0.084	0.000
ADASYN	0.000	0.014	0.000
OSS	0.120	0.635	0.332
CNN	0.024	0.979	0.182
Tomek	0.103	0.663	0.282

Table 4.6: Statistical comparison of the cluster mean auPRC from PTs compared to the naive approach based on the p–value.

Method	Cluster 1	Cluster 2	Cluster 3
CSS	0.999	0.911	0.995
SMOTE	0.922	0.998	0.876
ADASYN	1.000	1.000	0.985
OSS	0.692	0.207	0.639
CNN	0.929	1.000	0.831
Tomek	0.808	0.152	0.340

- When measured in auPRC (Table 4.6), *none* of the tested PTs yield an increase that can be statistically proven.
- When measured in auROC (Table 4.6), CSS, SMOTE, ADASYN, and CNN are statistically proven to be effective. This is aligned with the results shown in Schlegel and Sick [108], where (except for CNN) the same results were obtained. However, what should be noted that the *p*s shown in Table 4.6 are even lower in comparison to the values given in Schlegel and Sick [108]. Schlegel and Sick [108] tuned the hyperparameters in respect to the auROC (in contrast to this chapter, where hyperparameters were tuned in respect to auPRC).
- Using the F_1, only SMOTE on cluster 1 can be proven to be effective.

Conversely, this means that the effectiveness of OSS and Tomek can not be proven statistically when the classifier performance is measured in terms of auROC. If the classifier performance is measured in terms of auPRC, *no PT can be statistically proven to be effective.* As mentioned earlier, hyperparameters were optimized in terms of auPRC in both cases.

4.4.4 Computational Complexity

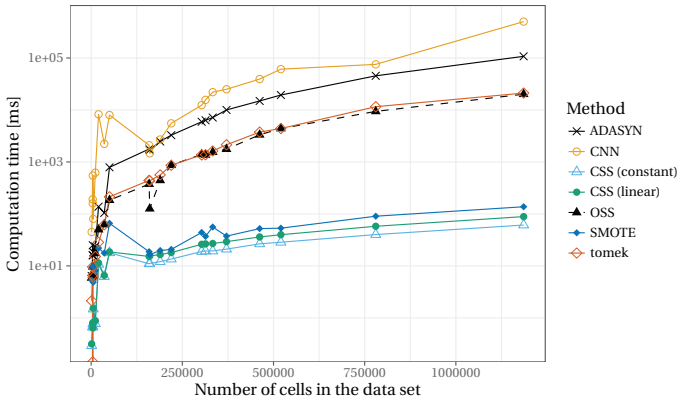


Figure 4.16: Computation time of different PTs.

Figure 4.16 shows the computation time of all examined PTs depending on the number of cells ($n_{cells} = n_{samples} \cdot n_{features}$) which varies across data sets. A clear trend can be observed especially for the larger values of n_{cells}. The bump for small values of n_{cells} (e.g. when using CNN) may be explained with specific characteristics of the data set that speed of the conversion of the (iterative) CNN algorithm resulting into a faster identification of a consistent subset. The *naive* approach is not displayed, since the processing time is zero (no preprocessing takes place). Averaged over all numbers of cells, CNN is by far the most computationally expensive method ($\mu = 37.35$s), followed by ADASYN ($\mu = 10.83$s). OSS ($\mu = 2.17$s) and Tomek ($\mu = 2.38$s) yield similar results. This is not surprising since both techniques rely on the same mathematical principles. With an average computation time of $\mu = 0.04$s, SMOTE is among the top three and only outperformed by both variants of CSS. The latter is clearly the best scalable approach, both versions (linear $\mu = 0.02$s, constant $\mu = 0.02$s) achieve the best performance in terms of computational complexity.

Figure 4.17 shows the mean training times for all evaluated classifiers based on the used PT. This figure is not cluster-specific since the same observations can be made independently from the cluster. SMOTE and ADASYN yield a strongly increased training time when combined with an OCC or k-NN classifier. These combinations should be avoided if training time is the only criterion. The increased training time can be explained by the additional, synthetic samples these two methods add to the training set. In contrast, after the data set

has been reduced by CNN, the training time is reduced across all classifiers.

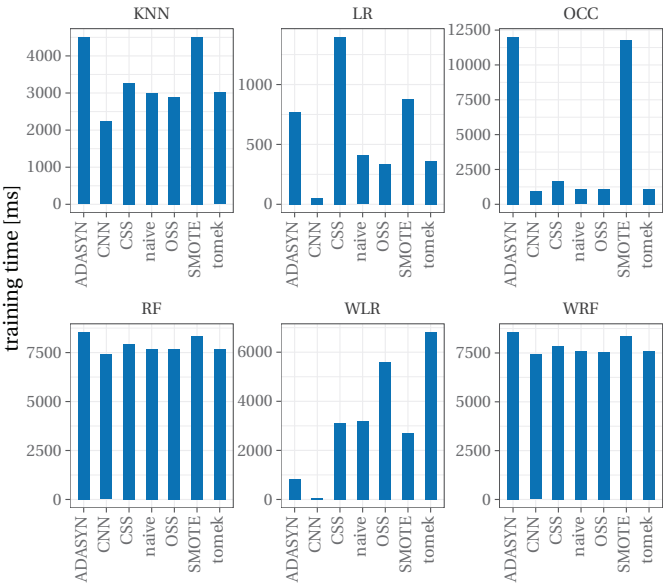


Figure 4.17: Training time of various classifiers dependent on the preprocessing technique.

4.5 Lessons Learned and Conclusions

The performance gain in terms of AUC of different PTs compared to the naive approach does not change across data sets with varying noise and class overlap (as measured with BS^2, see Section 4.1.2 for a detailled description) or imbalance levels. PTs that perform well, tend to yield the same level of performance independent on the data set or cluster as shown in the bottom row of Figure 4.14. Often OSS, CSS, and Tomek yield the best results (see Figure 4.14).

What separates the different PTs, is (1) their overall capability to positively influence the resulting model classification performance (e.g., in terms of auPRC), (2) their computational complexity, (3) their capability to lead to well generalizing models, and (4) their impact on the classifiers' training time .

(1) The positive effect on the achieved classification performance (auROC, auPRC) can only be statistically proven based on auROC for SMOTE (clusters 1 and 3), ADASYN (all clusters), CSS (cluster 3), and CNN (cluster 1). All other PTs do not result in an statistically sig-

nificant improvement of the achieved auROC according to the experiments layed out in this chapter. However, there might be other data sets, where a PT has a stronger, positive influence. This is even more sobering when statistical testing is performed using auPRC: In this case, *not a single* PT can be proven to have a positive impact on the resulting auPRC based on the deducted experiments.

(2) CNN, Tomek, and OSS should only be used if explicitly required by the circumstances, e.g., when the number of samples needs to be reduced due to storage restrictions (Figure 4.16). Although yielding similar results in terms of AUC (see Figure 4.14), ADASYN is way more computationally expensive than SMOTE. CSS is in general the best scalable technique, requiring the least computational effort. Also, SMOTE scales very well.

(3) According to Table 4.2, pairs that should be avoided due to overly optimistic training auPRC and therefore the lack of model hyper-parameter tuning capabilities are: ADASYN and SMOTE for all evaluated PTs, as well as WLR especially in combination with CSS, OSS, and Tomek.

(4) While SMOTE and ADASYN tend to increase the training time (Figure 4.17), PTs that pick a subset of samples such as CNN and OSS decrease the training time. Combinations, where the training time is increased significantly by the PT, are OCC and RF in combination with SMOTE and ADASYN. These should be avoided in any case.

Depending on the choice of metric to asses the classification grade, results as presented in Section 4.4 may vary. E.g., the main metric used by Schlegel and Sick [108] was the auROC, which led to the conclusion that applying a PT is helpful in many more classifier–PT combinations in comparison to the results obtained in Section 4.4. On one hand, this might be a sobering realization. On the other hand, this is an interesting observation: If the hyperparameters of a model are tuned with regards to the appropriate measure (auPRC), the classifier may work well out of the box, even in imbalanced scenarios. In contrast, when using an unsuitable measure (such as the accuracy) or even the auROC, using a PT can yield better testing performance. E.g., if accuracy is used, oversampling using SMOTE until the data set is balanced would yield huge increases.

The performance of CSS may be improved by performing a clustering analysis beforehand. This may be beneficial in scenarios where cluster centers are further apart than the corresponding centers.

Therefore, readers are advised to extensively think about the proper evaluation metric

before applying a PT. Furthermore, some models, such as the WRF, outperform other approaches (even when paired with an PT) out of the box.

Chapter 5

Estimating the Remaining Useful Lifetime

While the previous chapters focused on predicting the correct countermeasure for a problem that already occured, this chapter aims at predicting the Remaining Useful Lifetime (RUL). This provides the opportunity to replace diagnosed parts or to perform suitable counteractions *before* issues arise. This can beneficially influence a products' customer image, reduce costs otherwise caused by unplanned downtimes or off site repairs and has even the potential to avoid safety critical situations (e.g. failing breaking systems).

Forecasting the future failure of a component inherently requires to estimate the RUL of critical objects, parts or assets in general. In this chapter, four different algorithms to estimate the RUL will be surveyed: A "naive" regressor that serves as a baseline, an algorithm that scored first on the most cited RUL prediction data sets serving as benchmark, and two novel approaches. The first being a "bucketed random forest" which is able to accurately predict the RUL while requiring low computational effort once trained. Second, a similarity-based approach with an adjustable memory footprint of the trained model and runtime complexity for testing and training, which yielded very promising results.

All algorithms in this chapter are open sourced and evaluated for general validity on a variety of different data sets originating from complex and interdependent technical systems. Hereby, approaches are identified that offer the possibility to scale across various usage scenarios while requiring a minimized amount of manual effort.

The remainder of this chapter is structured as follows. In Section 5.2, the most popular related RUL estimation approaches found in the existing literature are briefly highlighted. A summary of data sets which are used in this chapter is presented in the Section 5.4. Section 5.5 gives a detailed overview on the RUL estimation approaches implemented in this

work. In Section 5.6, an experimental evaluation is conducted. Finally, a conclusion is given Section 5.7.

5.1 Notation and Definitions

In addition to the notation introduced in Section 2.1, the following symbols are needed to express the time series data used in this chapter. In general:

- $\mathcal{T}$ refers the set of time series,
- with each time series $\mathbf{t}_t \in \mathcal{T}$ consisting of L (length) samples: $\mathbf{t}_t = (\mathbf{s}_{t,1}, \mathbf{s}_{t,2}, ..., \mathbf{s}_{t,L})^{\mathrm{T}}$.
- The sample $\mathbf{s}$ is defined differently in this chapter: Instead of a Boolean target variable y_{tl}, the label or target variable in this chapter is continuous (y can be either the RUL measured in hours, or a risk value r). All data sets except for the *Automotive 1* hold equidistant samples. Thus, for all data sets but *Automotive 1*, a sample $\mathbf{s}_{tl}$ is defined as $\mathbf{s}_{tl} = (\mathbf{x}_{tl}, y_{tl})$. For *Automotive 1*, the samples are defined as $s_{tl} = (\mathbf{x}_{tl}, y_{tl}, ts_{tl})$, with ts_{tl} being the timestamp that allows to asses the temporal distance between two samples.

The risk for the lth sample of a given time series $\mathbf{t}_t$ is defined as follows:

$$r_{tl} = \frac{\max(RUL) - RUL_{tl}}{\max(RUL)} = 1 - \frac{RUL_{tl}}{\max(RUL)}, \tag{5.1}$$

with $\max(RUL)$ being the maximum RUL of the entire data set and RUL_{tl} being the current RUL of sample l from the time series. Thus, the r equals to zero at the beginning of a asset (turbofan, car, etc.) indicating a low risk, and $r_{tl} = 1$ indicating a failure ($RUL = 0$).

The Health Indicator (HI) is defined formally as

$$HI = \frac{RUL_{tl}}{\max(RUL)}. \tag{5.2}$$

5.2 State of the Art

Plenty of research has been conducted, that tries to predict the RUL. However, most existing approaches share the following caveats: First, they are built and optimized for a particular problem (which makes their adaptation to different scenarios cumbersome). Second,

they are often based on proprietary software or closed source software, preventing other researchers to use the algorithms out of the box.

The field of prognostics and health management (PHM) refers to a range of methodologies to tackle the challenge of predictively maintaining assets. A key requirement to be able to do so is real-time health estimation of an asset – be it a fridge, a car, or a space station – as well as the prediction of its future state [127, 49, 69]. Estimating the RUL is – as well as monitoring the assets state and predicting future trends – an important prerequisite to enable PHM. Known techniques for RUL estimation can be divided into three categories: physics-based, data-driven, or combinations of both (surveys are presented in [111, 60, 116, 49, 69]).

Physics-based model approaches represent the behavior of a system by building a dynamic model, making use of underlying physics and system knowledge. The upside of the resulting models is their usually high precision as well as their interpretability. On the other hand, these models are tailored towards very specific use cases and very hard to obtain for highly integrated and complex systems, such as cars [127, 60, 49, 69].

Data-driven approaches extract degradation behavior based on historical data from the modeled asset. Predictive models are typically trained offline. The upside of this approach is that a profound system knowledge including the underlying physics is not required (but may of course be beneficial). Hence, the underlying techniques and methods are easier to transfer to other scenarios or data sets, and therefore used for a wide range of RUL estimation problems. However, this generic applicability poses a trade-off in terms of model accuracy [127, 60, 49, 69].

Data-driven approaches usually involve the calculation of a HI or risk indicator (r) as given in Equation (5.1) and Equation (5.2). Since the approaches in this chapter are mostly data–driven machine learning approaches with some statistical concepts underneath, the following review highlights the most popular algorithms for data-driven RUL estimation.

Common, data–driven approaches based on *machine learning in combination with statistical algorithms* include the concept of Yan et al. [142], who combined LR with an Autoregressive Moving Average Model (ARMA) model to estimate the RUL of an elevator door motion system. In a three stage approach by van Tran et al. [128], an ARMA model and support vector regression was implemented to forecast the RUL of a low methane compressor in a petrochemical plant. Göbel et al. [47] compared three different data-driven algorithms on

the same data set in order to assess their predictive capabilities. All approaches, a probabilistic version of an SVM, a Gaussian Process Regression and an ANN-based approach yielded "sound" RUL estimates, despite the sparse and noisy data set originating from aerospace equipment with rotating parts.

Different architectures of ANNs are commonly used *machine learning techniques* for RUL estimation. Especially Recurrent Neural Network (RNN), which feeds back knowledge from the previous iterations to the next iteration works well to predict the RUL in dynamic settings. Vachtsevanos and Wang [126] proposed an RNN based on wavelets to predict crack evolution until the final failure of a rolling-element bearing. Heimes [59] used an RNN based on proprietary software to tackle the 2008 PHM conference challenge problem. Close to the results of Wang et al. [136] (see next paragraph), his approach achieved the second best result. Liu et al. [80] also proposed an RNN approach to predict the RUL of aging 18650-size lithium-ion cells. Three different types of RNNs (classical, adaptive, neuro fuzzy) have been applied on historical data to predict the RUL by an iterative one-step ahead procedure. More recently, a deep learning approach has been proposed by Deutsch and He [37] to estimate the RUL of rotating components. The authors chose a Deep Belief Feed-forward Neural Network algorithm to predict the RUL several steps ahead. The input data was based on NASA Glenn Spiral Bevel Gear Test Facility data.

Wang et al. [136] developed a similarity-based approach to solve the IEEE 2008 PHM conference challenge problem, scoring first place. Therefore, this approach was re-implemented and compared to the proposed approaches in Section 5.5. They deployed LR to generate an offline pool of degradation patterns. Through a comparison of the test units with the offline model pool (for a detailled description see Section 5.5.1), the current state of each test unit is derived and thus a RUL is estimated. A more detailed explanation how this algorithm works is given in Section 5.5.1.

A combination of LR and a relevance vector machine is proposed by Caesarendra et al. [25]. Bearing defect degradation of simulated and experimental data serves as an input to predict future failures and resulting RULs. Nuhic et al. [89] used Support Vector Regression (SVR) with a specifically developed data processing method to estimate the RUL of lithium-ion batteries. Battery data was gathered by investigating six high power lithium-ion cells for automotive application.

Another approach of applying several techniques on lithium-ion battery and *Turbofan*

data (which was also used in this chapter, see Section 5.4) is shown by Mosallam et al. [85]. Methods for feature selection and extraction were utilized to create an offline database of HIs and related RULs. A k-NN classifier determines the RUL by identifying the closest match of the test HI in the offline database.

To summarize, the majority of data-driven approaches in literature are applied and optimized on a single data set, which causes the approach by design to overfit a single scenario, thus lacking the capability to generalize well accross different data sets. Also, the algorithms proposed in literature and/ or the data sets used for evaluation are not publicly available. Further, similarity-based approaches are often not suitable for lightweight devices due to high memory and runtime complexity (as shown later), for both, training and inference.

5.3 Research Demand

Motivated by the state of the art just presented, this chapter includes the following contributions: First, to tackle the above mentioned limitations, a generally applicable machine learning approach and a memory-efficient similarity-based approach are introduced and compared to the RUL prediction benchmarks of Wang [136].

Second, both approaches are evaluated on a variety of data sets and modeling targets, which is necessary to ensure applicability in the automotive context where the “shape” of the data sets (in terms of the available number of features and samples) differs among targets. An evaluation of the algorithms on several data sets (Section 5.4) and applications with different characteristics has not been conducted yet.

The first of the proposed approaches is a similarity-based approach that yields extremely fast training times. In addition, it is very lightweight in terms of the storage requirements for the trained models and the lines of code used. Also, the model size can be parametrized to trade-off prediction accuracy for required storage. This ensures applicability on low-cost devices, making continuous RUL onboard monitoring in the car possible after the model has been trained.

Third, there is very few literature where RFs have been utilized to RUL prediction in technical applications. Only Frisk et al. [45] used Random Survival Forests to predict a battery lifetime function using fleet-management data from a heavy-duty truck manufacturer. RFs will be utilized in the second proposed approach.

Table 5.1: Overview on evaluation data sets.

Name	Samples for Training	Testing	Number of features	Average samples per object	max(*RUL*) [h]	Defect ratio	Split	Pre-clustered
Turbofan [103]	45047	28883	21	206.6	377.0	100%	61%:39%	yes
PHM08 [104]	45918	29820	21	210.6	356.0	100%	61%:39%	yes
Weather [146]	2670	858	19	80.9	48.0	30%	76%:24%	yes
Automotive 1	90031	532	27	3.7	4120.0	3%	99%:1%	no
Automotive 2	184176	59	1780	67.0	243122.0	< 1%	100%:0%	no

Fourth, the algorithms proposed in the following are publicly available on Github [106], including an open source implementation of the approach by Wang [136]. This is in contrast to e.g. Heimes [59], where "proprietary" software makes it impossible to reproduce the results.

Lastly, the *Automotive* data sets – although being technically time series from multiple objects – are too short (e.g. on average 3.7 samples for *Automotive 1*) to apply classsical time series techniques like windowing. Thus, the proposed approaches need to be capable to work well with short time series.

5.4 Data Sets

Table 5.1 gives an overview of the data sets forming the testbed. Entries are sorted by their defect ratio. The defect ratio indicates how many of the observed objects reached their End Of Life (EOL) in the data sets. All data sets except for the "Automotive" data sets are publicly available to enable other researchers to compare their results. To generate comparable results across data sets, the test-set is always filtered to only hold samples from objects that will reach their end of life (*EOL*) eventually. All data sets have been standardized prior to modeling. Unless predefined by the data set, a 75% : 25% training-test-split was used. All data sets consist of time series that vary in their length.

2008 PHM Data Challenge: The *PHM08* data set was originally published for the conference on prognostics and management by Saxena and Göbel [104]. Features represent sensor measurements from not further specified objects. This data set consists of equally spaced time series. The data set was split into three parts by Saxena and Göbel [104]: Training, testing, and final-testing set. The training set holds all samples of every training-object until failure ($RUL = 0$). The testing set holds only a subset: Records stop, before the *RUL* is zero.

The correct RULs of the test and final-test set are unknown. Predictions on the test data set can be submitted to the NASA website which calculates the PHM score s_{PHM} without revealing the actual RUL values. The final-test set can only be manually evaluated by sending an Email. The score s_{PHM} is defined as [102]

$$s_{PHM} = \begin{cases} \sum_{i=1}^{n} e^{-\left(\frac{d}{a_1}\right)} - 1 & \text{for } d < 0 \\ \sum_{i=1}^{n} e^{-\left(\frac{d}{a_2}\right)} - 1 & \text{for } d \geq 0 \end{cases}, \tag{5.3}$$

with n being the number of unique tested units (objects), $d = RUL_{pred} - RUL_{real}$, $a_1 = 13$, and $a_2 = 10$, respectively. Through the asymmetry caused by a_1 and a_2, late predictions are penalized higher. This score enhances the Root Mean Squared Error (RMSE) used for evaluation (Section 5.6).

As proposed by Wang et al. [136], the data set has been preprocessed: The operation mode (system state) is unambiguously identified by the first operational setting (which is a feature in the data set), yielding six clusters (operation modes). These operation modes were set manually prior to model building (column "Preclustered" in Table 5.1). The operation modes are thus discrete. The operation mode of a given object can change after every sample of the time series. Also, instead of using algorithms to select or transform features anonymously, Wang et al. [136] visually evaluated features by hand: Only features that had a clear trend indicating the failure of the object have been used to create their model. The following features were used: 2, 3, 4, 7, 11, 12, and 15. The features are non-further specified "sensor measurements".

NASA turbofan: The *turbofan* data set (Saxena and Göbel [103]) is similar to the *PHM08* data set in terms of value ranges, features, clusters, and equally spaced time series. The features are sensor readings from several turbofans. In contrast to the *PHM08* data set, the RUL of the last sample from every object is in a separate file. To ease the comparison with the *PHM08* results, the *turbofan* data set was modified as follows:

First, only data from *FD002.txt* and *FD004.txt* files (instead of all 4) were used, since only these two yielded the same number of clusters (operation modes). Second, 218 objects for both, the testing and training sets were randomly sampled to match the number of objects in the *PHM08* data set. This is necessary, since s_{PHM} linearly grows with the number of evaluated objects. Also, as pointed out in Section 5.4, the data set was manually preclustered into six clusters and the same features were selected (except for Section 5.5.4).

SML2010: Original aim of the *Weather* data set (Zamora-Martínez et al. [146]) was to forecast the indoor temperature, based on temperature sensors located in- and outside, as well as lightning-, wind-, and rain-sensors that are represented by 19 features. This data set was modified to suit the needs for RUL prediction: The "critical" temperature (that corresponds to $RUL = 0$) was defined as the 65% percentile of all inside temperatures measured. Days were considered "objects". The data was originally sampled every minute but smoothed to 15 minute intervals yielding equally spaced time series. The RUL is equivalent to the remaining time until the critical temperature is exceeded. This data set is available on the UCI machine learning repository (Lichman [77]).

Automotive 1 and 2: Both "Automotive" data sets were collected from hybrid cars and consist of non-personal data only. `Python` and `sklearn` [91] were not able to cope with the vast amount of data on the available hardware. Therefore, the data sets were reduced using the approach proposed in Chapter 3 to the dimensions given in Table 5.1. *Automotive 2* has a higher-dimensional feature space, and more samples per object than *Automotive 1*. Also, the RUL has been scaled differently. Each data set represents a specific component failure, which does only occur with a tiny fraction of examined cars (objects). A property worth to be pointed out is the comparably small average number of samples per object (3.7) in *Automotive 1*, since this represents a major challenge for methods that extensively rely on historical data that lead to a failure. Another difference between *Automotive 1* and *Automotive 2* is that while time series samples are distributed unevenly in *Automotive 1*, samples in *Automotive 2* are distributed evenly since they are automatically transmitted using telematics.

5.5 Proposed Solution

As a baseline for an evaluation of the algorithms developed in this chapter, a *naive* approach was implemented: In this case, a single RF is used to predict the RUL based on the normalized features. Aside from using cluster information indicating a certain operating mode (if available), no other optimizations were implemented.

Also, the similarity-based approach by Wang et al. [136] was implemented which is known to yield very accurate RUL estimations. The approach is outlined in Section 5.5.1. A more in depth explanation is presented in Wang et al. [136].

The novel approaches (Section 5.5.3 and Section 5.5.4) are in line with the generic re-

quirements: Both, Distribution-based Similarity Estimation v.2. (DBSE2), which is a completely newly implemented version inspired by the "Distribution-based Similarity Estimation" approach proposed by Schlegel et al. [107], and the RF based Bucketized RUL regression with trend-based feature selection (BRR) algorithm described in Section 5.5.4 are able to select features automatically based on a concept that is proposed in Section 5.5.2.

All approaches will be evaluated using the s_{PHM} (see Section 5.4) and the RMSE defined as [25]

$$RMSE = \left(\frac{1}{n} \sum_{t=1}^{n} (y_t - \hat{y}_2)^2 \right)^{1/2}. \tag{5.4}$$

5.5.1 Approach of Wang

The approach proposed by Wang et al. [136] won the IEEE 2008 PHM conference challenge with a total score of s_{PHM} = 5636.06 on the final test data set. The complete code implemented in `Python 3.6` as well as all parameters used in the algorithm can be reviewed in a public GIT repository [106]. The approach consists of two stages. In the first (training) stage, a database of degradation patterns is created. In the second (inference) stage, the observed degradation is compared to all models in the database (similarity based approach). Training involves the following steps:

1. Linear regression is used to map the (multi-dimensional) feature space (which represents the input of this modelling aproach) to a one-dimensional HI (Equation (5.2)). Linear regression was deliberately preferred over LR: LR is considered unsuitable in this case since it distorts the degradation pattern as the logit function is very flat when its output is close to 0 or 1. Yan et al. [142] propose LR in their RUL estimation pipelines, however, they do not model a continously scaled HI but a dichotomous target variable ("health" or "disease").

2. For each object (or time series $\mathbf{t}_t \in \mathscr{T}$) in the training set, a representative degradation model M is derived, which is essentially a function that maps a given HI to its respective RUL. This function can e.g. be an exponential function as proposed by Wang et al. [136]. The function is fitted to all HIs of a given time series in combination with their respective timestamps (RULs). Figure 5.1 shows a random time series (blue circles) and the respective quadratic model which was derived from the displayed time series (orange line).

3. The above step is repeated for all available time series which forms a pool of models $\mathscr{M} = \{M_1, M_2, \ldots, M_T\}$, with T being the number of time series. This results into a pool of models.

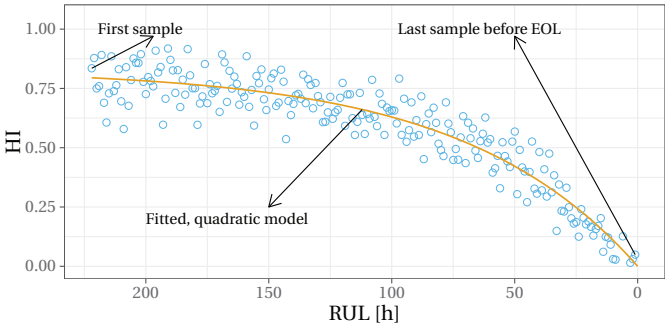


Figure 5.1: Example for a model which is derived from a time series.

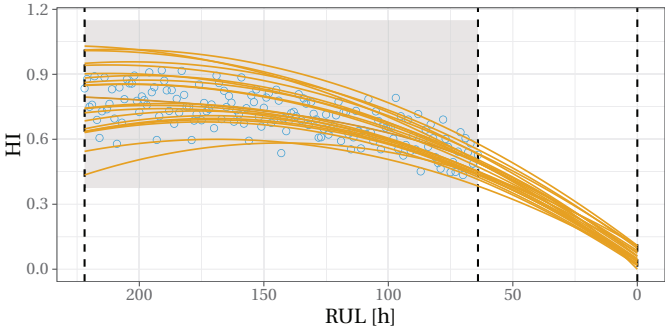


Figure 5.2: Visualization of mutliple models.

Estimation of a RUL for a given time series $\mathbf{t}_t$ works as follows:

1. A Euclidean distance measure is defined $d(\tau, \mathbf{t}, M)$, with τ being the number of hours that the time series (test object history) $\mathbf{t}$ is shifted away from cycle zero to minimize the distance d between the test object history and model M. "Shifting" can be best explained visually: At first (see Figure 5.3), the history of the test object (represented by the blue circles) would be aligned with the rightmost, dashed black line at $RUL = 0$. The orange line in Figure 5.3 represents one model $M \in \mathscr{M}$ of the model pool.

2. The history (blue circles) of the test object is now "shifted" ("moved") to the left by a certain amount of hours, to minimize the the distance d between the test object history and the given model. Once the RMSE is minimized (yielding e.g. Figure 5.4), the number of hours by which the test object history was shifted is equal to the RUL prediction of this specific model. This is repeated for all models $M \in \mathcal{M}$ in the model pool, created in the training stage. This is shown in Figure 5.2.

3. The final RUL is calculated using the weighted average (based on the distance d) across all models. This way, all known failure progressions are taken into account.

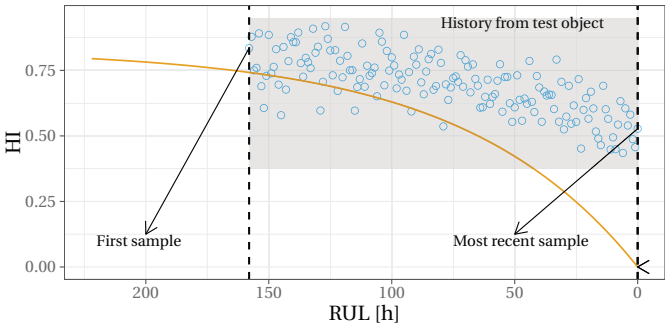


Figure 5.3: Example for a model using Wangs technique before the history is shifted.

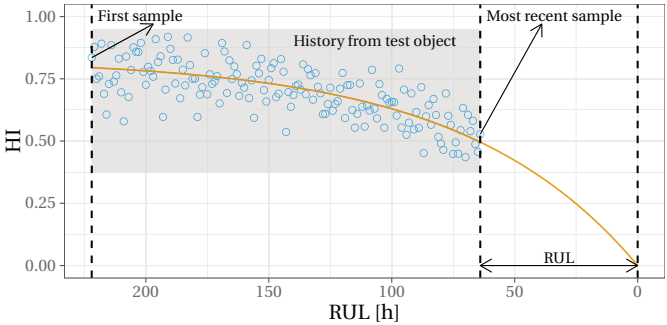


Figure 5.4: Example for a model using Wangs technique after shifting the history by the optimum number of hours.

A visualization based on the the open source implementation by Schlegel et al. [106] for an arbitrary time series T (that was mapped to a HI using linear regression) is given in

Figure 5.4. The pictured samples (blue circles) represent points in time, where the features already have been mapped to an HI. The orange line represents one model $M_i \in \mathcal{M}$ from the pool of available models. In the pictured case, the model M_i has a small distance d to the observed time series T, since it is fitting the samples well. The intersection with the x–axis represents $HI = RUL = 0$ and is used for prediction.

Although the hyperparameters of this approach can be tuned in theory, they were set explicitly based on the the recommendations by Wang et al. [136]:

- C_{min} and C_{max} define the thresholds for the minimum (and maximum) RUL that correspond to the minimum (and maximum) HI. As proposed by Wang et al. [136], $C_{max} = -5$ and $C_{min} = -300$ were used for the *Turbofan* and PHM data set. To adapt these values to other data sets, the corresponding RUL percentiles were identified: 2.612% (99.706%). Applying the percentiles to the range of RULs of the *Weather* data set yielded e.g. $C_{min} = -22.75$, $C_{max} = -1$.

- max_{cycles}: During the inference stage, each time series is shifted over each model from the model pool. max_{cycles} limits the maximum length of a time series which is shifted over the models. This decreases the run-time complexity by reducing the possible number of shifts. If the time series of a test object is longer than max_{cycles}, the oldest samples are removed, since the HI values at the beginning of a time series which is longer than max_{cycles} are mostly less informative. Based on the *PHM08* data sets and the recommendations by Wang et al. [136] the following heuristic was used: $max_{cycles} = 1.067 \cdot RUL_{max}$, with RUL_{max} being the maximum RUL in the training data set (1.067 was estimated based on Wang et al. [136] recommendations for the *PHM08* data set).

5.5.2 Polynomial-Based Feature Selection

As shown by Schlegel and Sick [109] and as a consequence of the results presented in Section 5.6, feature selection can positively influence machine learning: It can not only reduce training time, but also increase the classification/regression performance.

To select features for the time series data at hand, Polynomial-Based Feature Grading (PBFG) is proposed: As displayed in Figure 5.5, the HI ($[0,1]$) is plotted over all values of a single feature (scaled to $[0,1]$). These are fitted by a first-order ($f_1(x) = a_1 x + b_1$) and a second-

oder-polynomial ($f_2(x) = a_2x^2 + b_2x + c_2$). Since two polynomials were used, only the factor of the highest order monomial was taken into account to avoid mutual information (Fuchs et al. [46]).

This technique is visualized in Figure 5.5. On the left side, a non-informative (weak correlation between the HI and the feature $corr(HI, bad) = -0.09$) feature is displayed. In contrast to right, where an informative feature (stronger correlation $corr(HI, good) = -0.64$) is plotted. The intuition is that the clearer a trend of a specific feature is (and thus the larger the $|a_i|$s), the more helpful is it to estimate the RUL. Since there are cases, where $|a_i|$ is large, but the points are poorly represented by the fitted curve, the calculated feature specific weight w takes the RMSE of the fitted polynomials into account, yielding the final formula to assess the feature weight

$$w = \frac{|a_1| + |a_2|}{RMSE_1 + RMSE_2}, \tag{5.5}$$

with $RMSE_1$ being the error of the linear polynomial and $RMSE_2$ of the quadratic polynomial, respectively. The feature weight can be used for selecting the top n_{pf} features (as laid out in Section 5.5.4) or to weight the risk prediction according to the feature importance w (as laid out in Section 5.6.1).

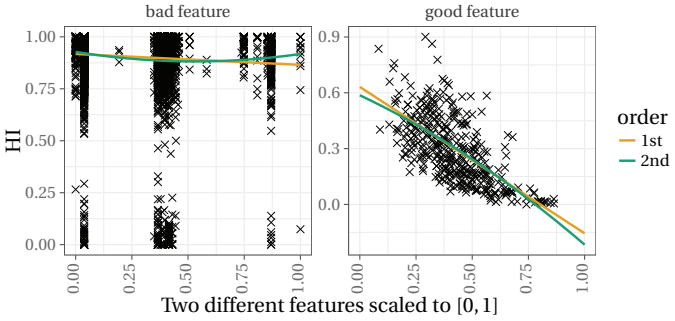
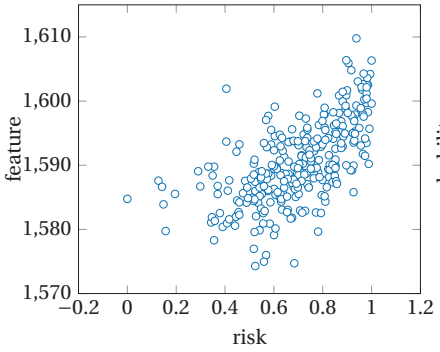


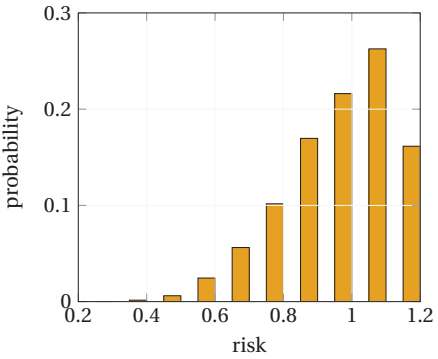
Figure 5.5: Example of two features holding different amounts of information.

5.5.3 Distribution-Based Similarity Estimation

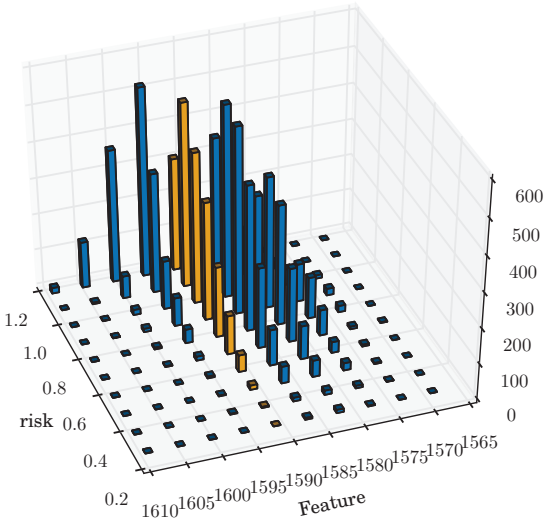
DBSE2 is a density-based estimation technique which is able to incorporate knowledge from a variable number of past samples from different operating modes *after* training was completed in addition to the most current sample. The basic idea is to generate a discretized



(a) Subsampled (1 : 16) scatterplot of feature X_3.



(b) Risk distribution for $X_3 \approx 1590$.



(c) Model to map discretized feature values to risk distributions.

Figure 5.6: Visualization of the key ideas of DBSE2.

probability distribution that maps each feature value to less or more likely RULs.

The first step in the *training phase* is to linearly map all RUL values to a risk indicator r ($RUL \mapsto r$) to unify all following computations. A scatterplot for feature X_3 of the NASA *Turbofan* data set and corresponding risk values is shown in Figure 5.6a. Figuratively speaking, a "model" in this case is a contourplot (or 2D histogram after discretization). The three dimensions are:

1. The centered and scaled feature value x.
2. The risk r.
3. The occurrence distribution $h(x,r)$, indicating how often feature value x and risk r were present in the training set.

Such a model is created for each feature. The continuous distribution $h(x,r)$ is then binned into a grid $h'(x,r)$ of size $n_{grid,feature} \times n_{grid,risk}$ (visually shown in for $n_{grid,feature} = n_{grid,risk} = 10$ in Figure 5.6c as a 3D histogram). This representation is scaled such that $\sum_j^{n_{grid,risk}} h'(x_i, r_j) = 1$ for all discretized ordinal feature levels $i = [1,2,...,n_{grid,feature}]$ (see Figure 5.6b).

Inference of a risk or RUL value for a given time series $\mathbf{t}_i = \{\mathbf{s}_{i,1}, \mathbf{s}_{i,2}, ..., \mathbf{s}_{i,L}\}$ of length L works as follows: For each time series element $\mathbf{s}_{i,l} = (x_{il1}, x_{il2}, ..., x_{ilP})$ (P features) within that time series, the corresponding model is polled. This yields a (discretized) risk probability distribution (as shown in Figure 5.6b) for every feature indexed by p of every given sample indexed by l part of $\mathbf{t}_i$ and each time series indexed by i. The respective pseudo code is given in Listing 5.1.

```
# get empty array for float tuples
risk_preds := Array((float, float))
t := getATimeseries()

# for each sample of the time series
for l:= 1 to L do
    # get the sample
    s := t[l]

    # for each feature
    for p:=1 to P do
        # get the feature value
        f_value := s[p]

        r_raw := getRawRiskValueFromHistogram(f_value, p)

        r_final := increaseRiskForPastSamples(r_raw, l)

        # append the risk and the
        # temporal proximity tuple to the array
        risk_preds.append((r_final, s))

prediction := predictAccordingToWTemp(risk_preds)
```

Listing 5.1: Pseudo Code to infer RUL values using the DBSE2 technique.

Every distribution of each feature of each sample of the time series yields a risk $r_{pred,raw}$ based on the highest probability. However, the "raw" risk needs to be adjusted to account for the passed time in case the sample under evaluation is not the most current of the respective time series. To combine past samples with the most current one, the risk of past samples needs to be raised to attribute for the passed time. Since this can be done for any number of hours (or timespan in general), it is not necessary that subsequent observations have the same time interval. E.g., the $h'(\mathbf{x}_{i(l-2)1}, r)$ distribution for a sample 2.3 hours in the past is shifted by adding $\Delta r = \frac{2.3}{\max(RUL)}$. The value is added to yield the final predicted risk: $r_{pred,final} = r_{pred,raw} + \Delta r$, which is appended to the array of inferred risk values.

The risk final prediction is – depending on the hyperparameters – either an average of the aggregated risk array holding a $r_{pred,final}$ for every feature of every sample, a weighted mean which linearly takes the temporal proximity to the most current sample into account, a weighted mean which takes the feature importance according to PBFG (Section 5.5.2) into account, or a combination of the latter two. The hyperparameters were heuristically tuned. These are:

- n_{hist}: The maximum number of historical samples to be considered (if less n_{hist} are

available, the available number of historical samples is used) for the prediction (evaluated values were $[1,4,8,16,32,64,128,256]$, depending of the data set specific maximum time series length),

- $n_{grid,feature}$: The number of bins to discretize the aforementioned distribution regarding the feature values (evaluated values were $[8,16,32,64,128,256,512]$),
- $n_{grid,risk}$: The same as above, just for the risk dimension ($[2,4,8,16,32,64,128,256]$),
- w_{temp}: Weight risk values in the risk array according to their temporal proximity to the most current sample (True or false), and
- w_{poly}: Weight features according to the PBFG technique (True or false).

If a data set contains different operation modes (e.g., Section 5.4 and Section 5.4), every operation mode is treated as a separate data set during training and prediction. The inferred risk values are aggregated the same way.

5.5.4 Bucketized RUL Regression

BRR combines three concepts: *First*, data is split into three parts based on the RUL. The split into three parts poses a tradeoff between accuracy and performance. This leads to more accurate results since every regressor can focus on the specific circumstances of a certain RUL range[1]. *Second*, since an RF does not take temporal dependencies into account, the features from the $n_{history}$ last samples of the same object are added as additional features. *Third*, valuable features are selected using the built-in trend analysis as described below.

Bucketizing of Training Samples

As displayed in Figure 5.7, BRR splits the data set into three subsets. First, all samples, with a RUL higher than $p_{crop} \cdot \max(RUL)$ are discarded and not used for training. This is due to the fact, that the "real" features (Figure 5.7, right) do not reflect any degradation in the early asset life. These indistinguishable samples provide no benefit for the classifier. Some wear has to occur, before the classifier is able to pick up the degradation.

[1] E.g. feature X_i might be important to estimate the RUL between 100 and 200, while feature Y might be important to distinguish RULs in the range between one and ten.

The remaining samples are split into two groups: critical ($EOL \leq RUL \leq RUL_{crit}$) and non-critical ($RUL_{crit} \leq RUL \leq RUL_{crop}$). The threshold is set by the a percentage p_{crit} with respect to the maximum RUL of the non-discarded samples (outside the cropped region marked gray). p_{crit} is set explicitly before training. During training, an RF is trained to to distinguish critical from non-critical samples (RF_{group}).

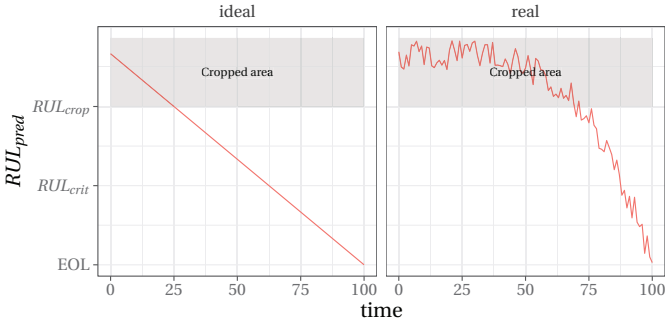


Figure 5.7: Different RUL areas.

Model

At total, three models are trained to estimate the RUL: The first, RF_{group}, distinguishes critical (close to the EOL) from non-critical samples. Afterwards, another two models (RF) are trained: One to classify critical samples ($MDL_{critical}$), and one for non-critical samples ($MDL_{uncritical}$). This yields a total number of three models. The RUL ranges of $MDL_{critical}$ and $MDL_{uncritical}$ do not overlap. To estimate the final RUL value, the predictions of $MDL_{uncritical}$ and $MDL_{critical}$ are combined in a weighted manner based on the criticality prediction of RF_{group}:

$$RUL_{pred} = RF_{group}(\mathbf{x}) \cdot MDL_{critical}(\mathbf{x}) + (1 - RF_{group}(\mathbf{x})) \cdot MDL_{uncritical}(\mathbf{x}), \tag{5.6}$$

with $\mathbf{x}$ being the features of sample under evaluation, $RF_{group}(\mathbf{x})$ referring the criticality prediction by the RF (which is naturally bounded by $[0,1]$, see Equation (5.1)), $MDL_{critical}(\mathbf{x})$ refers the RUL prediction of the RF specialized on critical samples, and $MDL_{uncritical}(\mathbf{x})$ the RUL prediction of the RF specialized on uncritical samples. This way, the RUL predictions of $MDL_{uncritical}$ and $MDL_{critical}$ are weighted according to the criticality prediction of RF_{group}.

X_1	X_2
1	3
2	4

(a) Original data set.

X_1	X_2	$X_{1,1}$	$X_{2,1}$	$X_{1,2}$	$X_{2,2}$
1	3	1	3	1	3
2	4	1	3	1	3

(b) Data set for $n_{history} = 2$ and "Copy".

X_1	X_2	$X_{1,1}$	$X_{2,1}$	$X_{1,2}$	$X_{2,2}$
1	3	0	0	0	0
2	4	1	3	0	0

(c) Data set for $n_{history} = 2$ and "Zero".

Table 5.2: Example for different $n_{history}$ modes.

Optimization

Preceding experiments have shown that the following hyperparametervalues yield promising results in terms of a small average and minimum s_{PHM} and RMSE. These will be evaluated in greater detail in the following.

- p_{crop}: Cropping of RUL, recommended values 0% – 50%.
- *drop*: If `True`, RULs above the threshold defined $RUL_{crop} \cdot p_{crop}$ will be dropped, if `False` RULs will be set to the threshold value.
- p_{crit}: Threshold for a sample to be considered critical, recommended values 0% – 50%.
- $n_{history}$: Adding of features from $n_{history}$ past samples from the same object, recommended values 0 – 5 (0 will not add any historical features).
- *mode*: Two different initialization modes are available to fill the feature vector $\mathbf{x}$ of the first sample of an object when the available history is shorter than $n_{history}$: "Copying" replicates the current feature values to the past ones, "zero" initializes the vector with zeros. An example for both modes is given in Table 5.2. As shown e.g. in Table 5.2c, if $n_{history}$ exceeds the number of available samples by more than 1, the values set initially in dependence on the "mode" can propagate to consecutive samples. The RUL of previous steps is not included as a lagged variable, since this value is unknown and would require $n_{history}$ inference computations.
- n_{pf}: Select the n_{pf} highest ranked features according to PBFG (Section 5.5.2), recommended values 5 – 15.

5.6 Evaluation

This section will evaluate the techniques introduced in Section 5.5 in greater detail. After tuning the hyperparameters for each novel technique, respectively, a comparison between all techniques will be laid out. The latter will compare all introduced techniques based on all data sets in terms of regression grade (RMSE and s_{PHM}) and run-time complexity for the training and testing stage.

All experiments in this section have been conducted on an HP™ Z-840 equipped with two Intel® Xeon® E5-2640 v3 2.60GHz CPUs and 96GB of RAM. A total number of 9503 models have been evaluated to substantiate the results of this chapter.

5.6.1 Hyperparameters of Distribution-based Similarity Estimation

Hyperparameter tuning of DBSE2 was solely performed on the *Turbofan* and *Weather* data sets. The reasons for this are three-fold: First, s_{PHM} scores on the automotive data sets exceeded the representable number range. This is due to the higher, maximum RUL values (the maximum observed RUL in the *Automotive 2* data set is ≈ 644 times larger than the maximum RUL of the *Turbofan* data set). Second, evaluating the entire hyperparameter grid was computationally infeasible on the automotive data set due to the higher data set size. Third, the actual RUL values of the *PHM08* test data set are not available.

The first evaluated hyperparameter is w_{poly}. If set to `True`, all risks are weighted according to the respective feature weight returned by the PBFG algorithm to generate the final, "averaged" mean risk. Thus, risk values from features with a high PBFG weight will have a higher influence on the final risk prediction. Figure 5.8 shows boxplots of the corresponding s_{PHM} and RMSE scores in dependence on whether PBFG-based feature weighting was used. While the median RMSE and s_{PHM} on the *Weather* data set were unaffected, both error scores were reduced by PBFG on the *Turbofan* data set. The median being at the top of the box for performing no PBFG on *Weather* data set in terms of RMSE might seem surprising. This can be explained by the fact that 67.86% of the evaluated RMSEs for this boxplot have the same (highest) value.

This may be explained by the fact that *Turbofan* contains demonstrably unhelpful features (which were manually sorted out by Wang et al. [136]), which might not be the case for the *Weather* data set. It is also to note that, unlike most machine learning algorithms (such

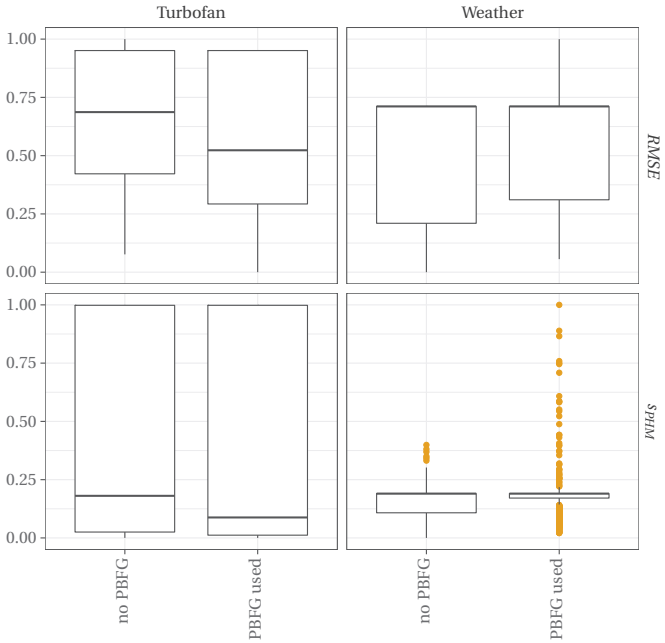


Figure 5.8: Effects of performing PBFG for DBSE2.

as the RF used in BRR), DBSE2 has no other inbuilt feature selection or weighting mechanism. Furthermore, since training and testing times were almost not affected by PBFG, it is in general recommended to use this technique.

Next, $n_{grid,feature}$ and $n_{grid,risk}$ are evaluated. Figure 5.9 shows the achieved RMSE and s_{PHM} scores for the *Weather* and *Turbofan* data set based on the used number of bins for discretizing the feature ($n_{grid,feature}$) and the risk ($n_{grid,risk}$). Refer to Figure 5.9 for the corresponding s_{PHM} scores. Again, the scores were scaled to a range between 0 and 1 for better visualization. While the score is color coded (black indicates a low/ good score), orange a bad score, the best score is marked by a bigger point. This represents the optimum.

On *Turbofan*, a higher number of bins for discretizing the feature is required (the optimum is reached with $n_{grid,feature} = 64$) than on the *Weather* data set, were the optimum is reached with $n_{grid,feature} = 8$. It is to note though, that the achieved scores were not strongly affected by the $n_{grid,feature}$. E.g., the mean of all (scaled to $[0,1]$) RMSEs of the *Weather* data set for $n_{grid,feature} = 64$ is 0.462 and thus only 1.99% larger than the best achieved value of

0.453 (for $n_{grid,feature} = 8$). Therefore, the recommendation is generally to try low values first (e.g. $n_{grid,feature} = \{32, 64\}$), as these will usually be sufficient – even if higher values do not have a significant negative influence. However it should be noted that the optimum value of $n_{grid,feature}$ is strongly dependent on the given scenario and its surrounding conditions such as the number of features or samples.

Regarding $n_{grid,risk}$, however, a completely different picture emerges: On both extensively evaluated data sets a surprisingly low value of $n_{grid,risk}$ yields the best results in terms of normalized RMSE and s_{PHM} score. While experiments on the *Weather* data set suggest $n_{grid,risk} = 4$, on the *Turbofan* data set even two bins are sufficient to yield the best results.

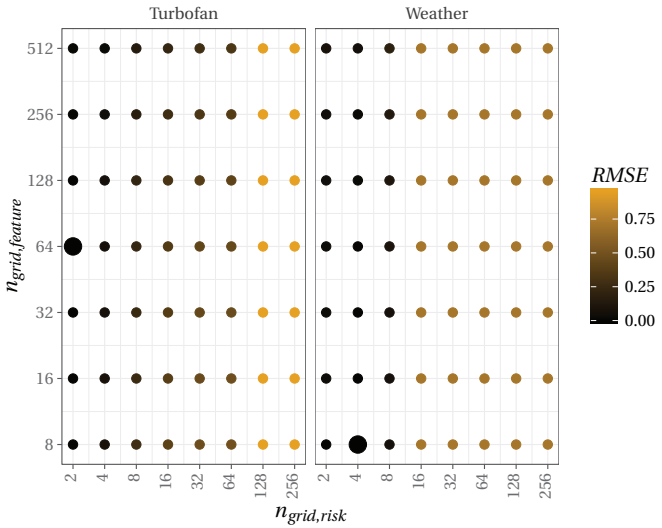
Considering historical samples of the same object ($n_{hist} > 0$) lowers both scores on both data sets. This is shown in Figure 5.10: Especially on the *Turbofan* data set, where not only the percentiles but also the median decreases with increasing values of n_{hist}. On the *Weather* data set, the median is unaffected due to the high number of experiments that yielded a similarly high error score in relation to the low number of experiments that yielded a lower error score. However, up to $n_{hist} = 32$ the boxplots indicate that more experiments yielded a lower error score. Based on the *Turbofan* data set, $n_{hist} = 32$ can be recommended as well: Both scores are plateauing for $8 \leq n_{hist} \leq 64$. Thus, $n_{hist} = 32$ is recommended as default value for other applications. Different applications, especially if the timespan between samples varries, may require different values. It is to note however, that n_{hist} should always be smaller than the maximum time series length of the data set.

Similarly to w_{poly}, where risk values are weighted based on the respective feature weight to generate the final prediction, w_{temp} can be used to weight risk values, too: In the latter case, weighting of risk values in the risk array is performed in accordance to their temporal proximity to the most current sample. However, as shown in Figure 5.11, temporal weighting does not lower the error scores based on the experiments on the *Turbofan* and *Weather* data sets.

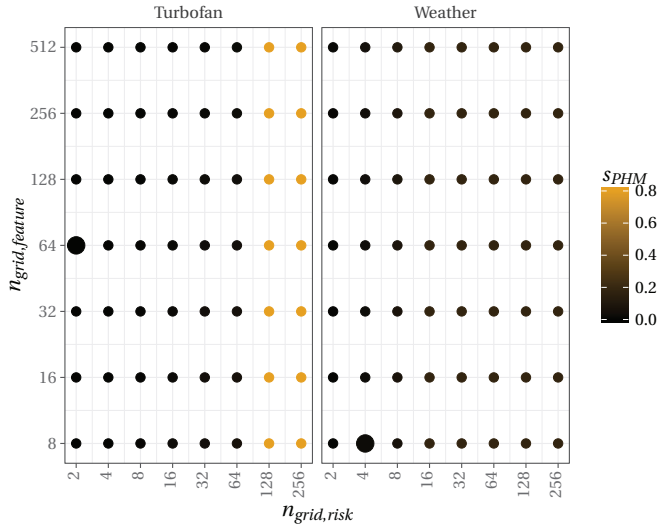
5.6.2 Hyperparameters of Bucketized RUL Regression

The detailed evaluation of BRR is solely based on the *Weather* and *Turbofan* data sets for the same reasons given in Section 5.6.1. However, the gained insights were successfully transferred to the other data sets, where they yielded promising results (see Table 5.3).

First, p_{crop} and *drop* are evaluated. Figure 5.12 shows boxplots based on the achieved,



(a) Evaluation for *RMSE*.



(b) Evaluation for s_{PHM}.

Figure 5.9: Evaluation of different n_{grid} combinations.

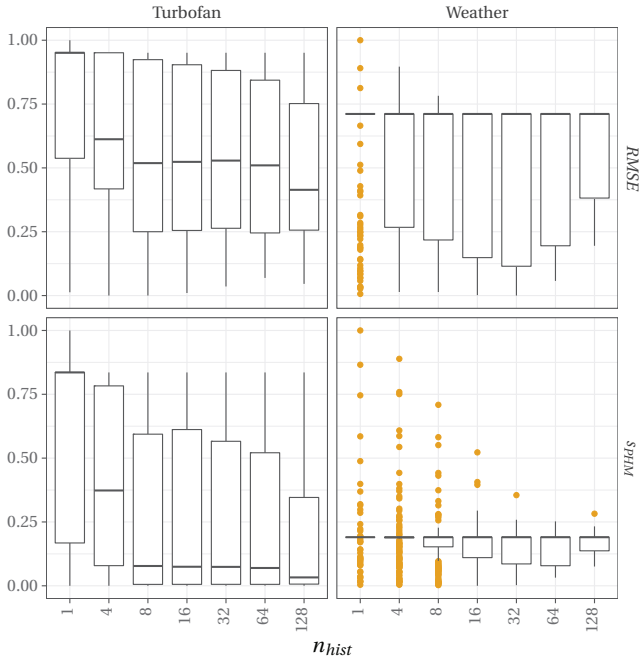


Figure 5.10: Influence of n_{hist}.

scaled RMSE and s_{PHM} scores. For values of $p_{crop} > 0$ an additional, the orange boxplot is plotted: This indicates that samples with a RUL above the RUL_{crop} (see Figure 5.7) are discarded and not used for training ($drop =$ `True`). As mentioned above, $drop =$ `False` indicates, that RUL values above RUL_{crop} were set to RUL_{crop}.

Two interesting facts can be observed: First, cropping in general can be considered a useful technique: The most obvious trend can be observed in terms of s_{PHM} on the *Weather* data set. Here, the error score was significantly reduced by this technique. The lowest scores were achieved by $p_{crop} = 0.5$. In terms of RMSE, lower and more consistent result can be achieved with $p_{crop} = 0.5$ as well (note the smaller box and the overall lower median). This consistency can be observed on the *Turbofan* data set as well in terms of *RMSE*. However, in terms of s_{PHM} the trend is much clearer (upper left in Figure 5.12): $p_{crop} = 0.5$ yields a much more consistently low error score with far less outliers. The benefical impact of cropping may be explained by the fact that every asset produces a lot of data before it eventually fails (if it fails at all) which inherently causes a high imbalance in most RUL estimation data sets. Crop-

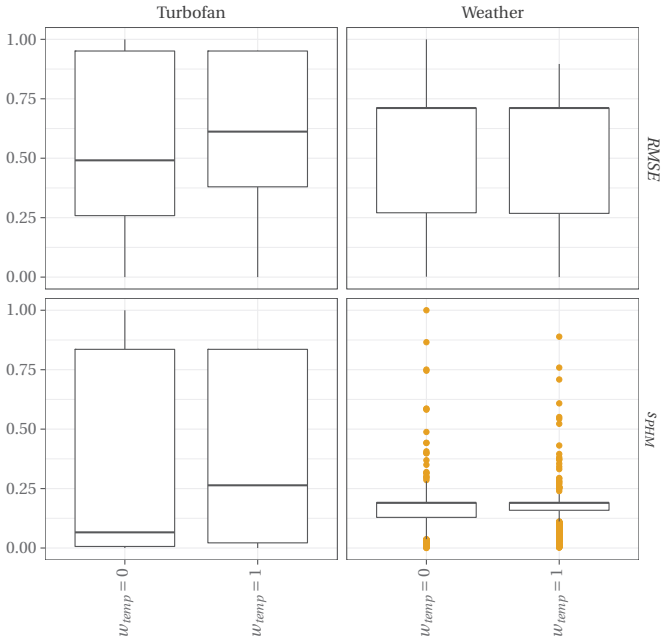


Figure 5.11: Influence of weighting risk values based on temporal proximity.

ping the early (healthy) life of an asset reduces this imbalance of healthy to failure-related samples to some extent.

The second interesting observation from Figure 5.12 is that dropping (*drop* =`True`) the cropped values *never* yields lower error scores: The orange boxplots always indicate higher error scores. While cropping is recommended in general, dropping should be avoided. Thus, $p_{crop} = 0.5$ and *drop* =`False` will be used in the following.

The next hyperparameter which will be evaluated is p_{crit}. As shown in Figure 5.13, no clear trend can be observed. Based on the s_{PHM} scores of the *Weather* data set, $p_{crit} = 0.3$ is yielding best results, however this contradicts the results based on RMSE: Here, $p_{crit} = 0.3$ is yielding the worst results while $p_{crit} = 0$ yields the best results. This result is aligned with the results on the *Turbofan* data set, where $p_{crit} = 0$ is also yielding the best results. Thus, setting $p_{crit} = 0$ is recommended in general. This means, that only a single RF is trained based on the non-cropped samples (ranging from $RUL = 0$ to RUL_{crop} in Figure 5.7).

The next examined hyperparameter is $n_{history}$ which will be evaluated in combination

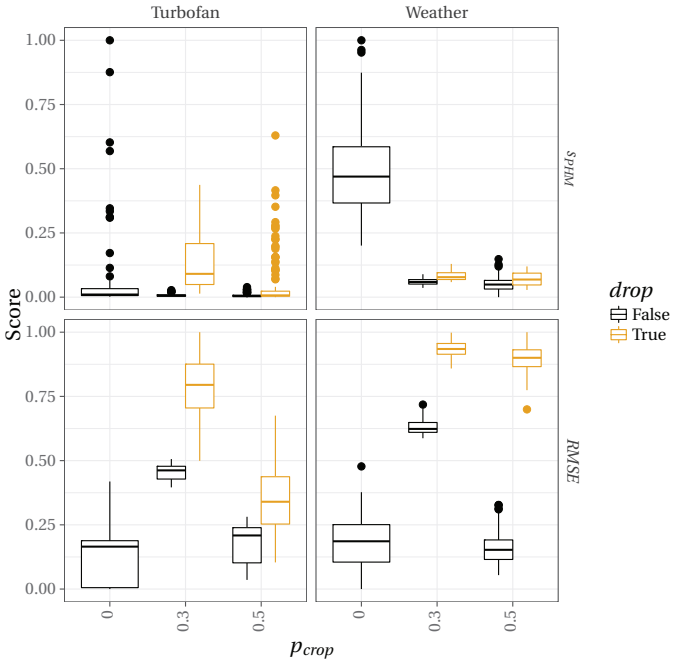


Figure 5.12: Effects of discarding samples based on their RUL.

with *mode* that determines how missing values are imputed which happens if $n_{history}$ exceeds the available number of available historical samples. For *mode* "Copy", the first feature values of the time series will be used again, for *mode* "Zero" zeros will be used (an example is given in Table 5.2).

As shown in Figure 5.14, adding no historical samples ($n_{history} = 0$) is outperformed in any case. However, too many historical samples ($n_{history} > 5$ for *Turbofan* and $n_{history} > 1$ for *Weather*) can increase the error scores. Thus, adding a small number of historical samples (results based on the *Weather* data set suggest $n_{history} = 1$, *Turbofan* results based on the s_{PHM} suggest $n_{history} = 2$) is recommended. Regarding the *mode*, no clear winner can be identified, although the results are consistent across both data sets: On the *Weather* data set, copying always yields the lower error score, on the *Turbofan* data set, adding zeros always results into lower error scores.

Since $n_{history}$ adds new features to the data set, it will be also evaluated in combination with n_{pf}: For $n_{pf} > 0$, the top n_{pf} features are selected based on the PBFG ranking. For

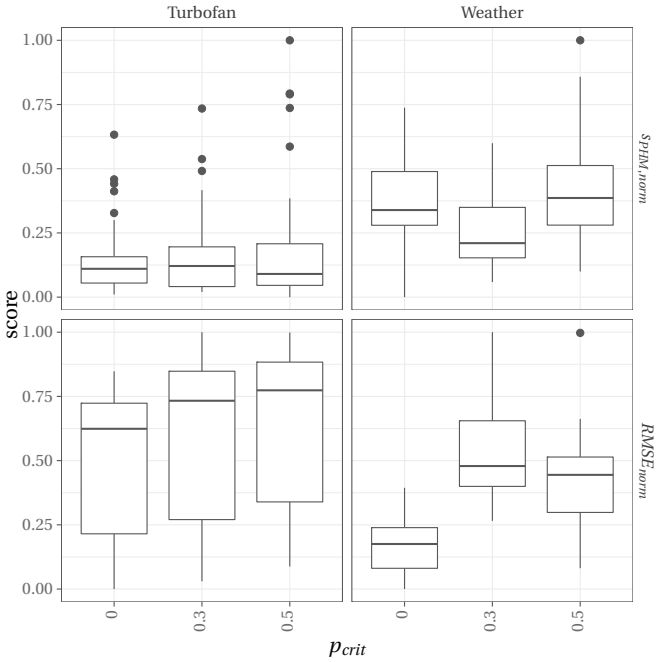


Figure 5.13: Effects of splitting the training according the RUL.

$n_{pf} = 0$, all features are used. On the *Turbofan* data set, selecting features according to Wang et al. [136] is plotted as $n_{pf} = -1$ in Figure 5.15. This however only works on the NASA *Turbofan* and *PHM08* data sets and was done manually (in contrast to PBFG which is calculated automatically).

Figure 5.15 shows the best achieved s_{PHM} scores for all evaluated $n_{history}$ and n_{pf} combinations. On the *Turbofan* data set, neither PBFG nor performing manual feature selection as proposed by Wang et al. [136] is able to lower the error score. A potential explanation for this might be that the RFs used by BRR are inherently able to identify and focus on relevant features. However, on the *Weather* data set, PBFG yields a lower error score ($n_{pf} = 10$) in comparison to when no feature selection takes place ($n_{pf} = 0$). This means, that the proposed polynomial feature selection is an adequate technique to select features for RUL prediction using RFs. However it should be noted that strongly differing results are close to each other in Figure 5.15. The expressiveness may thus be limited.

Additionally it should be noted, that performing PBFG-based feature selection (e.g.,

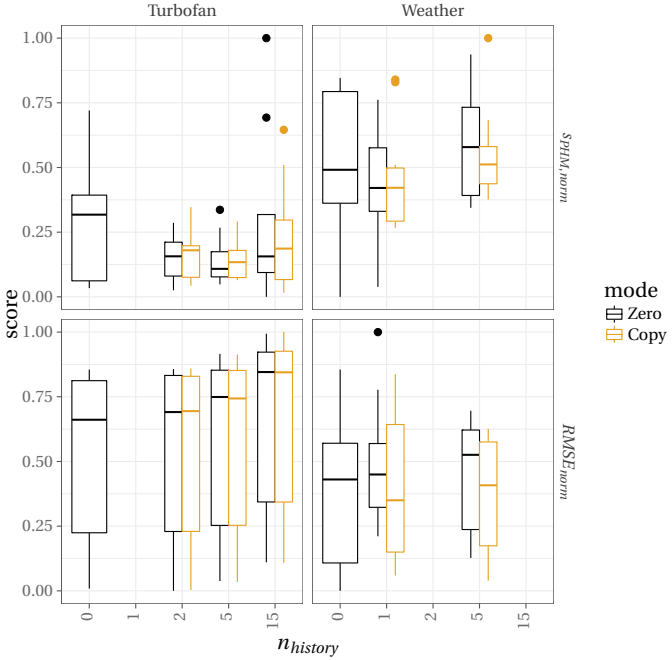


Figure 5.14: Effects of adding historical samples to the feature space on RMSE and s_{PHM}.

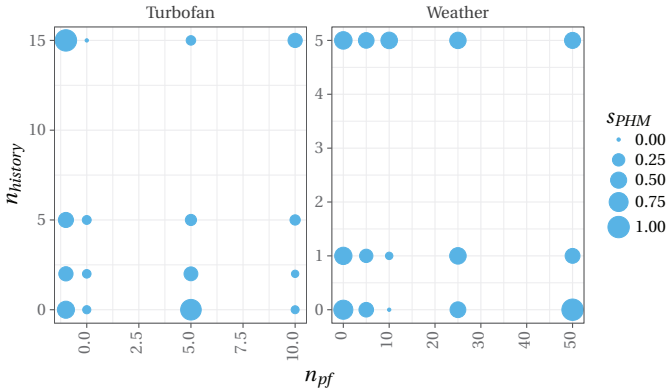


Figure 5.15: Effects of using polynomial feature selection on s_{PHM}.

Table 5.3: Performance comparison of all algorithms in terms of RMSE and s_{PHM} on all data sets.

	Automotive 1		Automotive 2		PHM08	Turbofan		Weather	
	s_{PHM}	*RMSE*	s_{PHM}	*RMSE*	s_{PHM}	s_{PHM}	*RMSE*	s_{PHM}	*RMSE*
BRR	Inf	1647.65	Inf	1876.45	3514.52*	6463.76	52.51	3.8	14.47
DBSE2	Inf	528.25	Inf	1898.24	35022.24*	26538.31	33.71	30.76	16.76
Naive	Inf	3355.21	Inf	2466.39	127290.08*	23605.99	54.77	97.45	15.25
Wang	Inf	846.78	na	na	1291.27*	27944.4	25.71	2.26	2.5

when using $n_{pf} = \in \{5, 10\}$) speeds up training by 2.21 times and testing by 1.36 times on average.

Also, adding feature derivatives as additional features was evaluated, without improving the performance.

5.6.3 Summary

Table 5.3 and Table 5.4 show the results on the test data set for each technique-data set pair (e.g. BRR-*Automotive 1*). RMSE for the *PHM08* dataset is not available since this value was not returned by the evaluation website. Since the hyperparameters of the Wang approach were explicitly set using the guidelines provided by Wang et al. [136], no hyperparameter tuning was necessary and only a single model was trained for each data set. The "best" model for all other combinations has been selected based on the lowest s_{PHM} score during training. If s_{PHM} exceeded the representable number range, the RMSE achieved during training was used. The scores for the *PHM08* data set (marked by an asterisk) have been obtained by submitting the predicted RULs of the test data set to the online-test-tool[2].

Table 5.3 shows the achieved s_{PHM} and RMSE scores. An "na" is indicating missing results. This is due to memory constraints (*Automotive 2*, Wang). "Inf" is indicating that the number exceeded the representable range of numbers. This only affects the s_{PHM} score. For the *PHM08* data set, no RMSE is given, since the website used for evaluation did only return the s_{PHM}.

The naive RF approach marks a solid baseline that is outperformed in terms of s_{PHM} and RMSE in most cases. Exceptions are the s_{PHM} score on the *Turbofan* data set and the RMSE on the *Weather* data set: In both cases, DBSE2 was outperformed slightly. DBSE2 models for

[2] Available under `https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/`, last accessed 07/11/2018.

Table 5.4: Training and testing time of the best performing model for all technique-data set pairs.

	Automotive 1		Automotive 2		PHM08		Turbofan		Weather	
	$t_{train}[s]$	$t_{test}[s]$	$t_{train}[s]$	$t_{test}[s]$	$t_{train}[s]$	$t_{test}[s]$	$t_{train}[s]$	$t_{test}[s]$	$t_{train}[s]$	$t_{test}[s]$
BRR	32.96	0.16	158.83	0.14	428.33	2.94	181.05	1.19	36.79	0.25
DBSE2	4.4	5.26	382.06	6.02	7.99	15.47	6.76	12.39	2	0.97
Naive	145.19	0.2	4335.48	0.14	124.05	1.11	120.39	1.01	35.5	0.24
Wang	1.2	1200.03	na	na	5.18	539.36	4.67	592.84	0.95	1.12

the *Turbofan* data set that scored $s_{PHM} = 110416.057$ exist, however, these yielded a higher train error and were thus not selected according to the above mentioned strategy.

BRR yields the best results on the *Automotive 2* and *Turbofan* (in terms of s_{PHM}) data sets and always outperforms the naive approach. Especially the low error score on largest evaluated data set (*Automotive 2*) is noteworthy.

DBSE2 yields very promising results on the automotive data sets: It yielded the lowest error score on the *Automotive 1* data set, a very close to the lowest score on *Automotive 2*.

Wang's approach resulted in a `MemoryError` on *Automotive 2* data set due to the many nested `for` loops, e.g., required to shift each test time series over each cycle in the trained offline model pool. Table 5.3 and 5.4 are therefore void in the corresponding cells. As expected, Wang outperforms all other algorithms on the *PHM08* data set. By adopting the hyperparameters as described in Section 5.5.1, this approach also yielded the best results on the *Weather* data set. On the *Turbofan* data set Wang approach yielded the best RMSE score. The approach proposed by Wang et al. [136] turned out to be a powerful and surprisingly versatile technique. Other researchers are encouraged to try out the open source implementation by Schlegel et al. [106].

All training and testing times given in seconds are shown in Table 5.4 to allow for a relative comparison: BRR yields the slowest training times in general. Only the naive approach takes longer to train on the Automotive data sets. However, testing times are significantly lower by multiple orders of magnitude in comparison to DBSE2 and especially Wang across all data sets. In terms of testing time, the naive approach always ranks similar. This is clearly due to the fact, that both approaches rely on a single RF (naive) or three RFs (BRR).

Wang, on the other hand, is *extremely* fast to train. This is due to the way the approach works: During training, no iterative optimization problems need to be solved. Instead, the training stage only involves fitting polynomials to the data points for each time series of each

object. However, testing is *extremely* slow due too the many nested for loops. It is important to note, that the Wang approach takes *longer to test* than any other approach takes for *training and testing* combined. This might render high frequency RUL monitoring on embedded hardware unfeasible.

DBSE2 shares the property of training extremely fast since training essentially consists only of creating a 3D-risk-feature histogram for each feature, respectively. The reason that training takes longer in comparison to Wang is that training also involves "testing": Due to the tunable hyperparameters offered by DBSE2, a testing score is required (which involves prediction). However, this is currently done single-threaded. E.g., the model used for *Turbofan* data set took 0.390s to create the 3D histogram and 6.371s to predict the RUL for all training samples (which is equal to 94.23% of the total training time). Major speedups can be expected when performing the prediction multi-threaded.

5.7 Conclusion and Outlook

This chapter presented one entirely new approach which has been implemented from scratch (DBSE2), a novel RF-based approach (BRR), and an open source implementation by Schlegel et al. [106] based on Wang et al. [136]. All approaches were evalatuated in comparision to a naive baseline approach consisting of a single RF on five different publicly available and proprietary data sets to ensure generalizability of the evaluated algorithms. In addition, all techniques were open sourced [106].

BRR leads to promising results and yielded consistently low error scores across all data sets. This indicates a broad applicability. While the training time is (except for the *Automotive 2* data set) the slowest, testing time is the fastest and on par with the naive aproach. Especially Wang but also DBSE2 are outperformed in terms of prediction time by multiple orders of magnitude. This enables new areas of application, such as continuous monitoring in the vehicle if the model is trained offboard beforehand. Future work should evaluate advanced bucketing techniques that take the class balance (the number of elements per RUL range) into consideration, and the evaluation of other regression models such as Bayesian regression to replace the RF. Also, a version of DBSE2 where no discretization (factorization) of the features is required and instead a distribution is fitted seems promising.

DBSE2 is a promising similarity-based approach which yields *extremely* fast training

times, and it is very lightweight in terms of code size and memory footprint during application. The inference step includes only matrix multiplications and additions which enables this algorithm to be deployed even in lowest cost Internet of Things (IoT) devices. Additionally, by reducing the number of bins used for discretizing features and risk levels, the memory footprint of the model can be further reduced since less feature-risk pairs need to be stored. This, however, trades off regression accuracy. Future research may evaluate different forms of RUL cropping (e.g., discarding samples with a high RUL for training). Also, instead of binned 2D histograms, parametric density models may yield good results. Lastly, the RUL estimation is currently performed single threaded. Parallelizing might result into major performance increases where multiple time series need to be evaluated (e.g. during training). Also, features that are already discretized before entering the model may be exluded from the built-in discretization algorithm.

A different issue which affects all RUL estimation techniques is that usually not all observed time series within a data set eventually encounter an error. This raises the question, what the true RUL labels for these non-faulty time series are, since the "ground truth" is unknown. In this chapter, the RUL of the oldest (first) sample $\mathbf{s}_{nf,1}$ of a non-faulty time series $\mathbf{t}_{nf}$ was manually set to 1.5 times the maximum RUL encountered in the whole data set yielding $RUL_{nf,1} = 1.5 \cdot \max(RUL_{dataset})$. All subsequent samples are relabelled accordingly: Let $\Delta T(\mathbf{s}_1, \mathbf{s}_2)$ be a function that returns the time passed between samples $\mathbf{s}_1$ and $\mathbf{s}_2$. Now, after the RUL of the first sample $\mathbf{s}_{nf,1}$ was set manually, the RUL of the subsequent sample $\mathbf{s}_{nf,2}$ is defined as follows: $RUL_{nf,2} = RUL_{nf,1} - \Delta T(\mathbf{s}_{nf,1}, \mathbf{s}_{nf,2})$. E.g., the RUL of sample $\mathbf{s}_{t2}$ is equal to the RUL of the previous sample ($\mathbf{s}_{t1}$) subtracted the timespan which passed between the two samples. Better strategies may exist and need to be identified.

Trends such as IoT offer the chance to use RUL-based predictive maintenance for completely new areas that go far beyond automotive applications. The methods – now publicly available – presented in this chapter are leightweight, generalizable machine learning based solutions to pave the way for high product quality and reliability in upcoming technologies and systems.

Chapter 6

Apache Spark and Application

Without explicitly disclosing, the *Automotive 1* and *2* data sets used in the previous chapter were transformed and preprocessed by Apache Spark. Especially *Automotive 2*, which is essentially a subset of the *Automotive 2L* data set (which is introduced later in this chapter), was not processable in-memory using conventional single-node frameworks such as `pandas` or `dplyr`. However, being able to perform calculations in-memory is an extremely desirable state, as this speeds up calculations considerably. Incremental online learning can only be used to a limited extent, since it is generally desirable to deemphasize older samples, which are based, for example, on an outdated software version, over time or not to use them at all.

In contrast to single-node setups, a cluster offers the possibility to scale up the available memory (Random Access Memory (RAM)) by adding more relatively cheap nodes built with commodity hardware. The by far most popular, de-facto standard in-memory cluster computing framework today is Apache Spark ("Spark"). Other, more streaming focused in-memory cluster computing frameworks include Apache Flink [130], Apache Heron [131], Onyx [133], and Apache Apex ™[129].

Since the existing data sets are overstraining single-node frameworks today and the data sets will become even larger in the future, Apache Spark will be evaluated in this chapter. For that to be possible, results and insights from the previous chapters are blended into a single machine learning pipeline, implemented entirely using `Scala` and Apache Spark, with the goal of being as *generic* as possible (and not tailored towards a specific application). This pipeline is then evaluated regarding the following questions: Which obstacles occur when training models on large amounts of data using Spark and how can these be avoided? Also, how the most promising feature selection methods and techniques to cope with imbalance

perform in a decentralized cluster setup on large data sets will be evaluated. Also, the scalability of the implemented pipeline and Spark will be evaluated. In order to build the bridge to the application, this chapter will also present a prototypical user interface and provide insights that may be taken into account in order to use the predictions of the pipeline as profitably as possible.

Section 6.1 will provide fundamental information regarding the design of Apache Spark, which is crucial for understanding the following sections. After defining the questions to be researched (Section 6.2), a complete pipeline is proposed (Section 6.3). The proposed pipeline starts with the extraction of key-value pair formatted data sets out of a relational database. Afterwards, the data sets are transformed (Section 6.3.1) into a columnar layout suitable for machine learning while paying attention to the specifics of the various feature types used. The machine learning models (Section 6.3.2) are then created autonomously, and the results served to an end user in an optically appealing way (Section 6.3.4).

The comprehensive evaluation starts with discussing pitfalls and caveats that emerged during the implementation of the aforementioned pipeline, and tries to provide helpful hints how to avoid them for fellow researchers whenever possible. A more in depth evaluation laid out in Section 6.4 evaluates the modeling grade (primarily in terms of auPRC) and whether the high expectations regarding the computing power and scalability of Apache Spark were fulfilled. Section 6.5 provides mental impulses regarding economic aspects, before deploying the machine learning pipeline discussed in the following to production. Section 6.6 will conclude by summarizing the key findings.

6.1 State of the Spark

This section aims to briefly introduce the concepts used in the following. Spark offers a rich set of classes and design patterns. It is to note though that Spark is still in an early development phase, and Application Programmable Interfaces (APIs) are likely to change in future versions. Spark was implemented in `Scala` and provides APIs in `Python`, `Java`, `R`, and `Scala`. Debugging Spark applications is simplified by using the native `Scala` API, which was used for this work.

Spark is divided into multiple libraries which can be independently combined based on the requirements. Spark “Core” forms the basis, the “MLlib” offers classes to implement

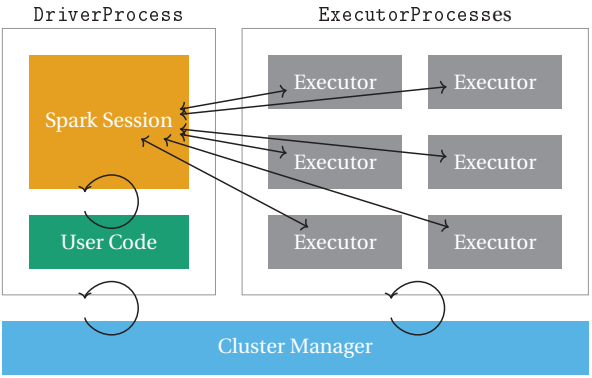


Figure 6.1: The Spark cluster architecture.

complete machine learning pipelines, “SQL” offers advanced data manipulation based on a syntax similar to Structured Query Language (SQL). These were the libraries used for this chapter. Other available libraries are “GraphX” (enabling distributed graph processing) and “Streaming” (enabling work flows for continuous data streams).

Figure 6.1 shows the main components of the Spark architecture based on Zaharia and Chambers [143]. Spark applications consist of one `DriverProcess` and a set of `ExecutorProcesses`. The `DriverProcess` is launched on a master node in the cluster. Its tasks include: responding to an users inputs, and managing the cluster in terms of scheduling and distributing work to the `ExecutorProcesses`. The `DriverProcess` forms the heart of a Spark application and maintains all relevant state information during the complete lifetime of an application. It manifests as a `SparkSession` to the user. This represents the entry point to perform all actions on the cluster. `ExecutorProcesses`, on the other hand, perform the actual computations and report state information as well as the final result back to the `DriverProcess`.

Spark has APIs to many other tools available in the Apache big data software stack. An extensive survey can be found in Kamburugamuve et al. [66]. One of the most common use cases is to use Spark in combination with Yet Another Resource Negotiator (YARN) [135] to allocate resources such as cores and RAM. However, this is optional, since Spark can also be launched using Spark’s standalone cluster manager (most likely requesting almost all resources, see Section 6.3.3 for further thoughts on this matter). Also, Spark has rich Hadoop Distributed Filesystem (HDFS) [115] support which ensures data locality.

Many of the concepts described in the following aim to perform successful parallel computations in a cluster setting on commodity hardware, even if nodes fail (this and the following sections are based on [144]). Another core feature of Spark is to be able to distribute data across the cluster. This way, even data sets that are too large to be stored on a single machine can be persisted.

Data sets are typically divided into partitions, which are distributed across the executors. Partitions are essentially chunks of a larger data set that were split priorly by the `DriveProcess`. Operations are performed with respect to the partitions' location on the corresponding executor (and node). Processing the data in-memory is one of the key optimizations for the speedup of Apache Spark for small, iterative workloads such as machine learning [145].

The programming model includes the following concepts: *Immutable* data sets (Section 6.1.1), *lazy* evaluation (Section 6.1.2), and *ephemeral* intermediate results (Section 6.1.4).

6.1.1 Immutable Data Sets

Spark offers different data types to store and interact with data. The Resilient Distributed Data Set (RDD) is the fundamental, typed, and parallel collection, the higher abstractions rely on. The latter are `DataFrames`, which are dynamically typed, carry a schema and can also be used for nested data (e.g., originating from a JSON file). They can be optimized for computations. `Datasets` share all properties of a `DataFrame` but are statically typed, thus offering compile-time type safety.

These core classes for holding data *cannot* be altered after creation. Sparks Programming Model refers to this as *immutability*.

6.1.2 Actions and Lazy Transformations

Spark offers two types of operations to be performed on `DataFrames`: Transformations and actions. They differ with regards to where they are executed and whether or not they trigger an execution.

Transformations only require work on the executors. Narrow transformations such as `.filter()`, `.map()`, and `.zip()` only require data which is locally available. Wide trans-

formations such as `.groupBy()` or `.repartition()` are likely to require data which is distributed across several executors and/or redistribute the data. This *shuffling* can be very expensive, because transferring data over the network may be required.

Actions perform work on the executors and return the (often aggregated) results to the driver. Thus, the result has to fit into the RAM available to the driver process.

Spark executes all operations in a *lazy* manner. This means, that all transformations are collected in a Directed Acyclic Graph (DAG) until data is requested by the driver. The latter corresponds to actions, such as `.count()`. This enables Spark to optimize the execution plan prior to the actual computation.

6.1.3 Estimators and Transformers

Similar to `sklearn` [91], Spark has adopted the pipeline concept, which allows the user to chain multiple pre-modeling and a modeling stages into a single workflow. An important distinction between `PipelineStages` is whether it is an `Estimator` or a `Transformer`.

`Transformers` process the passed data set in a distinct and priorly set way. `Estimators`, on the other hand, infer knowledge (and thus always offer a `.fit()` method) from the data set and return a `Transformer` afterwards, also referred as `PipelineModel`.

6.1.4 Ephemeral Intermediate Results

Intermediate results are not persisted after transformations unless the user explicitly requests Spark to do so (the Spark Programming Model refers this as *ephemeral*). However, persisting intermediate results can yield major performance increases, if the result (`DataFrame`, `Dataset`, `RDD`) of an expensive computation is again required after completion of the current task. Yet, this causes increased storage cost. Spark keeps lineage of all previous operations which led to a certain partition, to satisfy the fail safe requirement. Thus, being able to recompute the partition after node failure.

6.2 Research Demand

The automotive data sets used for the evaluation of the proposed algorithms in the previous chapters, already push libraries which run on a single-node such as `pandas` (`Python`) and `dplyr` (`R`) to their limits. In case of the “Automotive 2” data set from Section 5.4, the data set

needed to be reduced prior to processing. Also, the transformation from the original key-value pair format returned by the database into the columnar layout (required for machine learning) was only practicably by Apache Spark. This was not possible on any other single-node framework.

Thus, a multi-node framework which is able to work with large scale data sets that are expected to grow further in the future, needs to be identified and evaluated. The framework is subject to the priorly elaborated requirements, such as the generic applicability, and the suitability for imbalanced and heterogeneous data sets. In addition, the evaluation also assesses the scalability of the framework.

Also, potential pitfalls and corresponding countermeasures need to be identified.

6.3 Apache Spark Pipeline

The complete data-processing and machine learning pipeline is depicted in Figure 6.2. The following notations are used: Blue marks all elements that do not rely on any Spark library, orange is used whenever this processing step or storage is enabled by Spark. Also, thick black arrows indicate large scale data transfer, whereas thin gray arrows indicate lightweight communication such as the mean μ and standard deviation σ for the feature scaler or the actual "model" holding the tuned parameters after training.

The pipeline starts by extracting available information from a relational database. Information currently includes, among others (as denoted in Section 3.1.1), DTCs, MVs, parts, etc. It is to note, that this list can be extended without requiring any manual modification thanks to the overall *generic* design of the pipeline. The information is stored "as is" in a *key-value* buffer. This step is necessary to perform daily deltaloads (that only query the data that was added since the last query) instead of complete dumps. As mentioned earlier, this format is unsuitable for machine learning tasks. Thus, it is transformed into a columnar layout. This is the first step to utilize Spark. And, depending on the size of the data set, this is a step, where all other data manipulation frameworks such as `pandas` (`Python`) or `dplyr` (`R`) fail according to own experiments.

This columnar layout (stored as a parquet file, which is a common columnar storage format in the Hadoop ecosystem) is imported by Spark as a `DataFrame` (see Section 6.1.1). Stratified splitting is performed to ensure that the training and test set share approximately the

same class balance ratio (this chapter exclusively focusses on classification). Afterwards, following the training path, techniques for imbalanced learning are applied as evaluated in Chapter 4 and described in Section 6.3.1 regarding the specifics of Spark. Afterwards, strings are encoded numerically (e.g. using One Hot Encoding (OHE)), and discrete and continuous features are normalized. The identified values for the mean and the standard deviation are stored to be used later on for testing and inference. After applying feature selection (Section 6.3.2) the actual model training takes place. The output is, aside from the tuned model parameters, meta information that, e.g. allowing to assess the model quality.

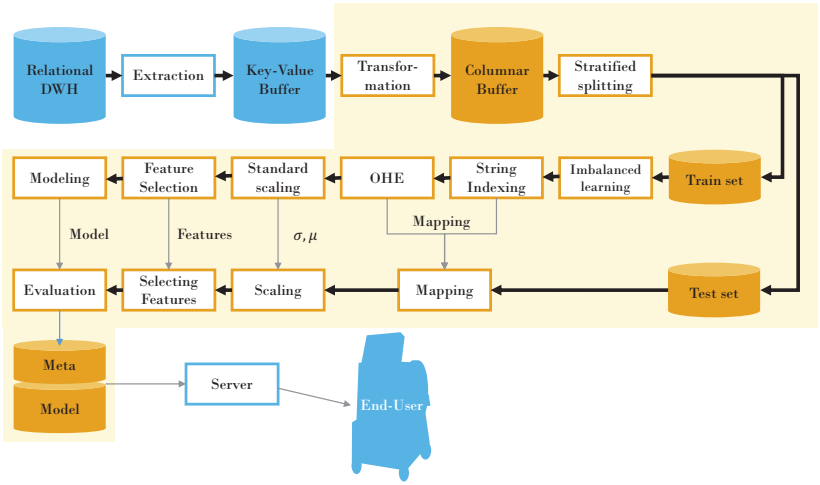


Figure 6.2: Overview over the pipeline.

6.3.1 Preprocessing

Although briefly noted throughout the previous chapters, preprocessing is explained in more in detail here. The fundamental task to tackle is a binary one-versus-rest classification. Since the ingested `DataFrame` holds all available features and targets, the following logic is applied: The `TARGET` column is set in a "1-vs-all" manner: It is 1 if and only if the modeled target was observed at this sample (e.g. "the part was switched") and 0 else. All other features of the same type are removed (e.g. if the target is a DTC, all other DTCs are dropped), as well as all other potential targets such as parts, actions, and DCs. Also, "Metacolumns", holding

information such as the readout ID are kept for later usage (such as presenting information to the end-user, see Section 6.3.4) but not used for training.

Another goal of the preprocessing step is to convert the `DataFrame` with one column of each feature into a format that is required by `MLlib`. `MLlib` requires all features to be contained in a single `VectorColumn`. Each cell of this column holds a vector of discrete, continuous, and Boolean features. The `TARGET` column is required to be a discrete (for classification) type column as well.

As shown in Figure 6.2, the `DataFrame` is now split into a training and a test set. In this chapter, a fixed value of `trainPercentage` = 0.7 was used. Since the Automotive data sets is highly imbalanced, this split is performed stratified. This ensures, that the balance between minority and majority classes is approximately the same in the training and testing set. Not performing a stratified split yields serious problems this poses the risk that not a single minority sample makes its way into the testing set. Although this is less likely, a training set without a single positive sample can cause the training to fail entirely.

Imbalanced Learning

Based on the insights gained in Chapter 4, the most promising approach (SMOTE) as well as Random Under-Sampling (RUS) were used.

RUS was performed in any case to speed up computation. The variable `samplingRatio` was used to set the ratio of majority class samples in relation to minority class samples (e.g., `samplingRatio` = 5 refers a training set, where five times as many majority class samples are present compared to the number of minority class samples).

String Indexing

String indexing is necessary to make string encoded information such as the color “black” (that might affect the thermal behavior of a car due to its higher absorption) accessible to the model. To do so, the `PipelineStage` called `StringIndexer` was used, yielding a nominal column where every unique string was replaced by a number. The Spark `DataFrame` internally marks the feature as a nominal feature. This way, it is ensured that the feature is not treated as a ordinal, discrete, or continuous feature. Since the `StringIndexer` is an `Estimator`, and therefore needs to be trained using the `fit()` method on the data set, special care is needed: If a string occurs in the testing set which was not present in the training

set, a dummy number is assigned.

Special Treatment of Important Features

The Spark API allows to define User defined functions (UDFs). These are handy to perform a certain task for all values of a column or feature respectively. UDFs were defined to process two string typed columns, that were important for the diagnostic use case and would not have been represented properly by the other techniques discussed in this section. The first one is the dealership number, which is unique to every dealer worldwide. It is an nominally encoded integer (a dealer that is "one unit away" is not necessarily located in close distance). The interesting, geographical information (enabling inferences regarding to the climate, gasoline quality, etc.) is hidden by default. Therefore, a UDF was defined to transform the dealer number to longitudinal and latitudinal coordinates (Section 3.4.1). The other variable where manual effort was invested is the version number. This string consists of two parts: The version itself, incrementing over time, and a maturity grade metric. Since a new version might fix a known issue on one hand, but might introduce a new issues due to the low maturity, these two informations were extracted into two separate, continuous and discrete features.

"Special treatment" seems to contradict the generic claim of this chapter and this work as a whole. Due to the outstanding importance, especially of the used software version, and the fact that both of the specially treated features are one of the few features available across the whole fleet, this decision is inevitable.

One Hot Encoding

Nominal features (such as produced by a `StringIndexer`) can be misleading to the model, since they suggest a linear relation where none exists. To tackle this, a `OneHotEncoder` was used. The idea is to transform a nominal feature where, e.g. "black" is represented by a 0 and "white" is represented by a 1 into two binary columns: A "is black" column, which is 1 if and only if the nominal column was 0 and a "is white" column which is 1 if and only if the nominal column was 1. After one hot encoding, the original, nominal column is dropped.

Standard Scaling

For reasons discussed in Section 3.1.3 (improving convergence and allowing weight-based feature importance assessment based on L_1 LR), scaling is performed on all continuous features if an LR model was used. The `StandardScaler` `PipelineStage` was used to create zero mean $\mu = 0$ and standard deviation $\sigma = 1$ features. RF-based models do not require this step, which is why this `PipelineStage` is skipped in the latter case.

Assembling the Vector

As noted earlier, `MLlib` requires the features to be encoded in a `VectorColumn`. Fortunately, the Spark API offers a `PipelineStage` that makes this as easy as one line of code: The `VectorAssembler` transforms all `inputCols` to a `VectorColumn`. Aside from the actual feature values, metadata is stored along with it. This is important to be able to assess feature importances after training. Also, the `TARGET` column is casted to `double`. Now, the data set is ready for training.

6.3.2 Modeling

After the preprocessing completed, the actual modeling takes place. First, relevant features are selected. Second, the model hyperparameter space is evaluated using CV.

Feature Selection

Spark `MLlib` natively offers only χ^2-test-based feature selection, which was identified in Chapter 4 as the most promising technique. Thus, other promising techniques from Chapter 3 were implemented as Spark `PipelineStages` by Kaminski and Schlegel [67] based on the theoretical foundations given in Section 3.1.

These are a `CorrelationSelector`, a `GiniSelector`, an `InfoGainSelector`, an RF-based `ImportanceSelector` and an L_1 regularized `LRSelector`.

Since the Gini coefficient as introduced in Section 3.1.2 and used throughout Chapter 3 was the worst performing filter measure in terms of "contribution" (see Table 3.7), the `GiniSelector` (referred as "Gini") in the remainder of this chapter will evaluate the "Gini gain"[1] as proposed by Kononenko and Matjaž [72] and Čehovin and Bosnić [26] based on the

[1] Originally, the measure was called "Gini index" by Čehovin and Bosnić. However, this name conflicts with the Gini index as proposed by Breiman [20].

Gini index proposed by Breiman [20]. For a feature X_p the Gini gain is defined as follows [26]:

$$Gini_{gain} = \left(1 - \sum_{x \in \mathbb{D}_{X_p}} P(X_p = x) \sum_{y \in \mathbb{D}_Y} P(Y = y | X_p = x)^2\right) - \left(1 - \sum_{y \in \mathbb{D}_Y} P(Y = y)^2\right). \quad (6.1)$$

$P(X_p = x)$ refers the probability of feature X_p taking value x, $P(Y = y|X_p = x)$ refers the probability of the sample belonging to class y given that the feature X_p has value x, and $P(Y = y)$ refers the probability of a sample belonging to class y when being randomly drawn from the data set. The first part of Equation (6.1) (left of the second minus sign) calculates multiple Gini indices for every distinct (binned) sub-data set where feature X_p has only a single value and averages the Gini indices in a weighted manner according the probability of feature X_p taking the respective value. The second part calculates the Gini index for the original dataset. Finally, a "gain" is calculated by subtracting the second from the first part.

The χ^2 test required extra work, since it only works on discrete random variables. `MLlibs QuantileDiscretizer PipelineStage` is a built-in method to transform continuous features (columns) into ordinal features (columns). This, however does not work for `VectorColumns`, which is why this class had to be extended. Each feature was binned into 8 discrete levels, as this provided a good trade-off between classification grade and computational complexity.

Based on the ranking from all aforementioned measures, the top `percentile%` features were selected for further processing.

Cross-Validation

The built-in `CrossValidator PipelineStage` was used to tune the models hyperparameters. The best hyperparameter combination was selected based on the best auPRC during training (averaged across all folds). It is to note though, that only model hyperparameters were tuned using CV. Other, hyperparameters such as the training-test-split, used sampling ratios, or settings of different upstream `PipelineStages` were selected based on the examinations laid out in the previous chapters. The tuning grid was created using the built-in helper object `HyperParameterGrid`. Tuned hyperparameters for the RF are:

- `numTrees`: The number of trees in the forest.
- `maxDepth`: The maximum depth of each tree.

- `numBins`: The number of bins to discretize the feature space.
- `impurity`: The split criterion to perform the impurity based branching.

For the L_1 regularized LR, the following hyperparameters were optimized (the `elasticNetParam` was set to L_1):

- `regParam`: The regularization parameter α according to Equation (2.7), with $\alpha = \frac{1}{C}$ (see Equation (2.12)).
- `maxIter` or `tol`: Whichever applies first, defines when the algorithm considers itself converged. `tol` refers the minimum loss reduction that is required to happen on every iteration, `maxIter` is naive upper bound for the number of iterations that usually indicates that the model failed to converge.

6.3.3 Cluster Setup

At the time of the experiments, Apache Spark was still a relatively new framework. Thus, no up-to-date hosted version was already available. Therefore, a prototypical Spark 2.1 cluster including an HDFS was set up using two workstations `WS1` and `WS2`. `WS1` had 32 CPU cores clocked at 2*GHz* and 92*GB* of memory. `WS2` is the same workstation that was used for the experiments laid out in the previous chapters (32 CPU cores at 2*GHz*, 96*GB* RAM). The workstations were connected using a point-to-point Gigabit Ethernet connection.

The resources of a Spark cluster are shared between a single `Driver` process and multiple `Executor` processes. While the `Executors` perform the actual work (e.g. transforming key-value into columnar layout) and are distributed across potentially multiple nodes, the `DriverProcess`is located on a single node. It is responsible for to an users program, maintaining state information, distributing the work the the `ExecutorProcesses`, etc.

Further information on how to setup a Spark cluster and launch a application are given in Appendix A.

6.3.4 Graphical User Interface and Backend

The Graphical User Interface (GUI) aims to serve three different use cases:

1. Simplifying the development, e.g., by easing the debugging process.

2. Assisting workshop staff in the future.

3. Providing additional information to asses model quality and important features to BMW engineers in the future.

It was implemented based on the Model-View-Controller pattern: The front end (view) was implemented using AngularJS [50] and Boostrap [132]. AngularJS is a `JavaScript`-based framework for single page web applications. Bootstrap is a toolkit that offers a rich set of predefined controls, layouts, and other GUI components. The program logic (controller) was implemented using the Play framework [78]. This was the obvious choice, since it is the most popular web-framework that offers a native `Scala` API which eases integration with the `Scala` implementation of the proposed Spark pipeline.

An example of the user interface is shown in Figure 6.3. In the top right corner, a search input is available. This can be used to input a Vehicle Identification Numbers (VINs), or specific readout IDs. In both cases, the Backend is queried for all readouts of the corresponding car. Multiple readouts can be selected at the same time. This way, changes are traceable over time. In Figure 6.3, three readouts from the four available are selected (on the left of the depicted car). For each readout, all available models are evaluated. For each model, a score is calculated (details on how this score is calculated are given in Section 6.5). Tapping on the model brings up a screen that shows additional information such as the auPRC or the important features used (not displayed).

An issue, also discussed in Section 6.4.1, is the long inference time. The upside is, that the prediction of a score approximately takes the same time, independent on the number of passed samples – be it 10, 100, or 1000. However, the time for loading and applying a model is $\approx 10s$. This makes it unfeasible to calculate a high number of model (each representing a distinct part, action, etc.) rankings upon request by the GUI. Pre-calculating and caching might solve this issue. However, the current implementation of Spark cannot be recommended for workloads where a high number of models is applied to a small number of samples.

6.4 Evaluation

This section briefly evaluates the proposed pipeline. The focus is explicitly on the suitability of Spark itself (Section 6.4.1), the computational complexity, and scalability (Section 6.4.3).

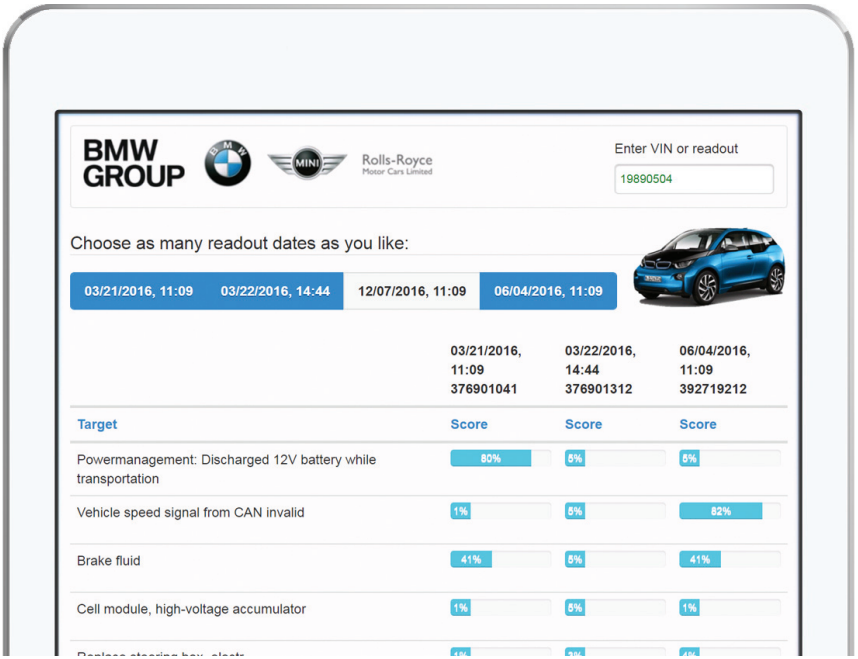


Figure 6.3: Prototypical implementation of the user interface.

Techniques, already discussed in greater detail (Chapter 3: feature selection, Chapter 4: tackling imbalance), are not re-evaluated here. The minimum number of positive samples for a target to be modeled was 20 throughout this chapter.

6.4.1 Limitations of Apache Spark

Through the extensive examination of Apache Spark in a real world scenario, the following interesting insights surfaced.

Persisting Intermediate Results

Transformations in Spark are *ephemeral*, which denotes the circumstance that intermediate results are discarded upon completion of the task. An "intermediate result" is, e.g., a `DataFrame` that already includes all features and targets in a columnar layout after all preprocessing steps, as described in Section 6.3.1. A "task" consists e.g., of all steps necessary to build a model for a certain part. Discarding the aforementioned `DataFrame` after the task

has been completed would require that exactly the same (or at least a similar) `DataFrame` is recomputed for the following modeling target.

To avoid this, the Spark API offers two methods to persist (keep) intermediate results: `DataFrame.cache()` and `DataFrame.persist()`. While `cache()` uses the default storage level, this can be explicitly set with `persist()`. At total, 12 different storage levels are available, differing in the location (RAM versus disk), compression, redundancy level, etc. Selecting the right storage level is crucial to ensure fast computation. `cache()` should be avoided, since it typically uses a slow combination of RAM and disc. By default, intermediate results should be persisted to RAM only. If this is no option, the following options can be considered (in this order): The usage of compression, applying sampling to reduce the data set, or organizing a more capable cluster. Storing on disc should be avoided in any case: Dean [16] examined that disc latency is up to 1500 times slower than RAM (even with Solid State Drive (SSD)).

Coping with High Dimensionality

Spark 2.1 is surprisingly unsuitable to work with high-dimensional data sets with more than approximately 3000 columns *independent* on the hardware. Due to the lazy nature of Spark (see Section 6.1.2), an execution plan is created until results are requested. Among other things, this execution plan includes all features' names before and after each transformation. The storage complexity of the execution plan thus increases by $O(n_{transformations} \cdot n_{features})$.

Therefore, multiple, consecutive `DataFrame.withColumn()` method calls, which are usually used to create new columns, should be avoided. The reason is, that every call increases $n_{transformations}$ by one. Instead, bundling multiple column creations into a single transformation using `DataFrame.select()` is advised.

The execution plan is optimized before any computation is executed. Optimization, e.g., by moving transformations which reduce the amount of data to the front, is in general a good idea. However, the execution plan optimizer creates many constant variables during optimization. The Java Virtual Machine (JVM), which is used by Spark, restricts constant variables to be no larger than $2^{16} - 1$ `Bytes` (or in this case pure American Standard Code for Information Interchange (ASCII) `chars`). This limit is quickly exceeded when working with high dimensional `DataFrames`. This issue, however, is known [40] and has been fixed in Spark 2.3. Which again underlines the importance of having an up-to-date Spark cluster to

the disposal.

A workaround was to split the pipeline into smaller steps. This, however, increased the runtime. Transforming features into a different feature space (e.g. using PCA) would have been another possibility, but was not feasible due to the requirement of an interpretable feature space.

The Bottlenecks of Optimization

As discussed earlier, Spark evaluates all transformations in a lazy manner and optimizes the DAG before execution. As beneficial this might be for the actual computation on one hand, as time consuming the process of optimization can be on the other hand: This process is unfortunately single-threaded. Especially a large amount of transformations (≈ 1000) performed on many features (≈ 2000) can cause this optimization to take multiple hours.

Concurrent Model Building

One might assume that Spark enables highly parallel model building given the plentiful resources of cores and RAM available. However, this is not exactly true. While Spark distributes transformations, and the training of a single model across the cluster, is not possible to train different models at the same time out of the box. To do so, instantiating multiple master nodes would be required, which is not aligned with the point of having a big cluster made accessible by a single entry point that distributes workload automatically. Thus, training models on a single and potent workstation is still faster in most cases, since the overhead for synchronization and task distribution is basically zero. Also, if multiple threads are launched in the `DriverProcess`, these threads are competing for the ressources. In this scenario, timeouts occur regularly causing tasks to finish only rarely.

Inference

The downsides of training models with Spark was already discussed. Prediction based on the trained models using Spark is also cumbersome, since a fully featured `SparkContext` is required to do so. Having a possibility to load and use serialized models that were trained by Spark would have made the whole process more practicable. Spark is able to export models to Predictive Model Markup Language (PMML) files (which is basically a file holding, e.g., all inferred model parameters), but as of today there is no execution or modeling framework

which is capable of importing PMML files. A lightweight version of Spark, which does not require a running cluster infrastructure, would be another option to tackle this, but is not available as of today. Packages such as `mlleap` [139] claim to ease deployment of a Spark pipeline, but require the re-implementation of custom `PipelineStages`, which would have been very cumbersome due to the high proportion of custom code.

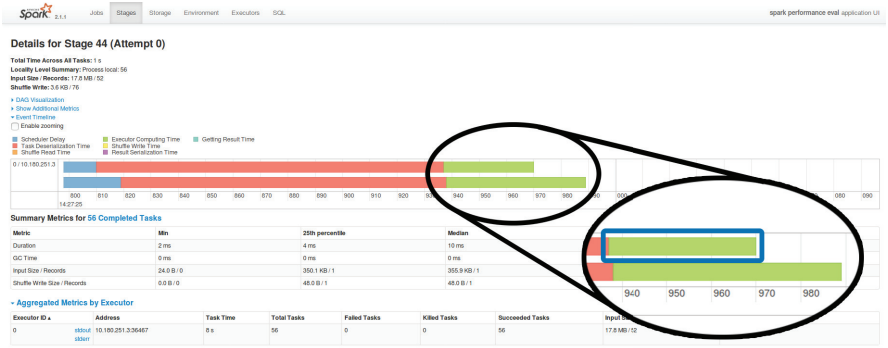
The Right Number of Partitions

Setting the right number of partitions is crucial. The number of partitions can be set explicitly by `DataFrame.repartition()`, which is absolutely necessary for Spark to parallelize well. Unfortunately, this is not done automatically (e.g., based on a heuristic). Typically, when a Comma Separated File (CSV) is loaded from disk – which happens regularly in the proposed pipeline (see Section 6.3) – the number of partitions defaults to a single partition, which essentially disables parallelization. The number of partitions needs to be increased.

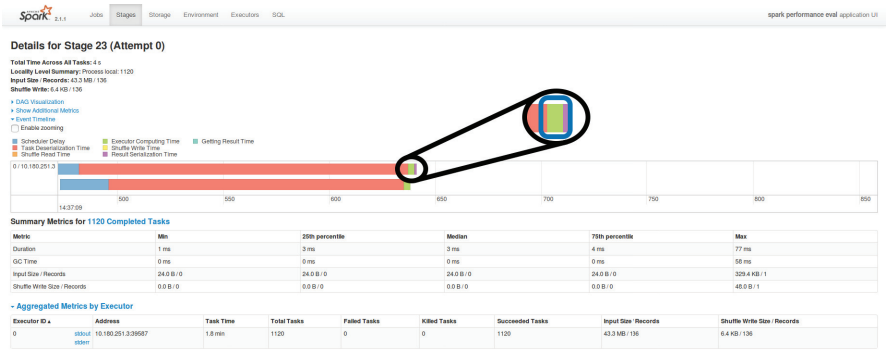
However, some stages are easy to compute. In this case, reducing the number of partitions of the `DataFrame` passed from the previous stage can increase the performance. The same task is depicted in Figure 6.4a and Figure 6.4b, respectively, for different numbers of partitions. Figure 6.4a shows the computation times for the different computations of a single stage. The bar consists of two different parts: The actual computation (green, marked by a blue box), and the overhead (all other fractions), including the scheduler delay, task de-serialization time, etc. In Figure 6.4a, the `DataFrame` is partitioned into 1120 partitions. This yields massive amounts of overhead: Only $\approx 1\%$ of the total time are used for the actual computation yielding a total computation time of $1.8 min$. The amount of overhead can be reduced, by reducing the number of partitions to 56. The share of actual computation is this way increased to 10%, yielding a total computation time of $8 s$.

Rapid Development

Due to the fast pace of Spark development (a new version adding new features is released every six months [134]), big cooperate surroundings tend to be several versions behind. This can be particularly obstructive when the new version introduces essential features. An example for this is model serialization which has been introduced with Spark v.2.0 [118]. If model persistence is a requirement, and the companies infrastructure lacks this update, the developer is forced to instantiate and manage an own cluster.



(a) SparkUI with 56 partitions.



(b) SparkUI for the same task with 1120 partitions.

Figure 6.4: Performance comparison for different numbers of partitions.

6.4.2 Data Sets

Similar to the corresponding *Automotive* data sets introduced in Section 5.4, the data sets used for evaluating the techniques proposed in this chapter are similar, yet larger (note the *L* suffix), thanks to greater computing power offered by Spark. The data sets are shown in Table 6.1. Note that a battery related DTC in *Automotive 2L* was actually used as the (only) target. That is, all other DTCs have been dropped.

Automotive 1L

Automotive 1L was collected during workshop sessions from 30000 hybrid vehicles and consists of non-personal data only. It was collected between the end of 2013 and mid of 2017 and offers a variety of potential targets. Each car was sampled four times on average. It consists

Table 6.1: Details on large *Automotive* data sets.

Data set	Samples	Targets			Features						Number of
		TA	SP	DC	DTC	RO	CP	EE	EC	MV	Features
Automotive 1L	121900	529	1109	462	380	13	20	202	146	243	3107
Automotive 2L	5130616	0	0	0	1	8	20	297	25	1428	1781
Credit Card	284807	-	-	-	-	-	-	-	-	-	30
NIPS 1,2	100000	-	-	-	-	-	-	-	-	-	1000
NIPS 3,4	100000	-	-	-	-	-	-	-	-	-	2000

of 1004 features and 2100 potential targets. In the *Automotive 1L* data set, TA, SP, and DC were used as targets. Only ≈ 21.0% of the potential target DCs, and 20.6% of the SPs and TAs exceed the above mentioned requirement of at least 20 or more positive samples. A number of 20 positive samples yields a balance ratio of 0.000164 (roughly 1 : 6100).

Automotive 2L

Automotive 2L was collected from 95000 hybrid vehicles and consists of non-personal data only during the same period. Each car was sampled 54 times on average in this data set, with an average distance of 11.05 days between the samples. This data set is interesting since it already hints the growing amount of available data and therefore underlines the necessity to identify a scalable solution for data-driven workshop diagnostics. The balance ratio of this data set is 0.0000663 (roughly 1 : 15000). This data set is used to evaluate if Spark is scalable to deal with even large-scale data sets. In this data set, the single DTC was used as target (class), all other feature groups were used as features.

Credit Card

In the *Credit Card* data set [34] each sample is an anonymous credit card transaction from European cardholders performed in September 2013. The 30 features include time of the transaction and the transferred amount. The remaining features have been transformed using PCA for privacy reasons. The target is to predict whether the transaction was fraudulent (1) or not (0). This data set was selected for several reasons: First, using a non-automotive data set ensures general applicability. Second, the high number of samples and the imbalance ratio of 0.00173 (roughly 1 : 577) are in the same region of *Automotive 1L*, ensuring transferability of the gained insights. Third, this data set enables the identification of promising hyperparameters in a more timely manner due to the lower number of dimensions.

Synthetic Data Set (NIPS)

The evaluate the feature selection algorithms, multiple synthetic data sets were used. This allows to set the number of relevant features explicitly. To do so, the `make_classification()` method provided by `sklearn` was used to generate a CSV file that was then imported by Spark. The used generation algorithm was proposed by Guyon [53] to create data sets for the Conference on Neural Information Processing Systems (NIPS) held in 2003. Four different data sets *NIPS* 1-4 were created with varying amounts of overall features including a certain number of informative features.

The latter are the only ones to carry helpful information to predict the class. This data set, unlike any other data set used in this work, was perfectly balanced. Also, each class consisted of 10 clusters, which is another parameter (`n_clusters_per_class`) of the described method `make_classification()` to create a more realistic and challenging data set.

Table 6.2: Details on the *NIPS* data set.

Name	samples	features	informative
NIPS1	100000	1000	10
NIPS2	100000	1000	100
NIPS3	100000	2000	10
NIPS4	100000	2000	100

6.4.3 Experiments

For all experiments, a 70% to 30% test-train split was used. Hyperparameters were tuned using a 3 fold CV on training data. The two major pipeline hyperparameters, influencing the scalability most (which is the major concern of this chapter), are

- the used model type, being either LR or RF, and
- the sampling ratio which defines the ratio between the number of negative (majority class) samples and positive (minority class) samples in the data set. A sample ratio of 5 indicates that the majority class samples in the training set are sampled down such that there are “5” times as many majority class samples as minority class samples in the training data set.

Other tunable hyperparameters are not optimized in the following but rely on the insights gained in Chapter 3 and Chapter 4. Instead, the most promising techniques will be

evaluated regarding their cost (in terms of computational complexity) and beneficial influences on the classification performance (in terms of auPRC) in a cluster computing setup. To reduce the hyperparameter space before the evaluation of feature selection and imbalance techniques, optimum values for model type and sampling ratio are identified based on the *Credit Card* data set.

Figure 6.5 shows the achieved auPRC in dependence on the used model type and the used sampling ratio. Multiple, interesting observations can be derived: First, the LR tends to yield a higher testing auPRC for higher sampling ratios. However, this effect fades away as the sampling ratio increases. Second, the RF yields higher auPRC performance, independent of the sampling ratio. This might be explained with the linear nature of the LR, which is not able to reflect the actual (nonlinear) decision boundary. Third, a sampling ratio above 5 only yields a drastic increase of the variance of the achieved test auPRC when an RF was used and should thus be avoided.

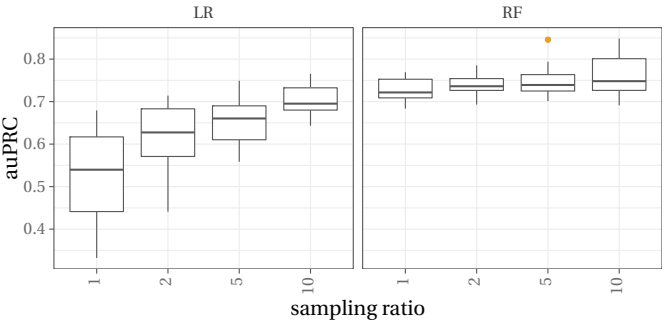


Figure 6.5: Evaluation of different sampling ratios based on auPRC on the *Credit Card* data set.

The training times depicted in Figure 6.6 show a very clear trend for the LR classifier: The larger the used sampling ratio, the lower the training time. This is counter-intuitive, and may be explained with the way an LR is trained (optimized). A higher sampling ratio results into more data available for training, which in case of the LR clearly yields better classifiers (Figure 6.5). This may cause the iterative training algorithm to stop earlier, since the minimum classification grade improvement required to start the next iteration is undershot earlier.

The training times of the RF are only slightly increasing with higher sampling ratios and in general lower (faster) in comparison to the LR. This contradicts the insights gained in

Section 4.4: According to Figure 4.17, training of an LR was 5 to 15 times faster in comparison to training an RF. Even after taking the size of the tuning grid into consideration (LR: 12, RF: 6) this can only be explained with the better parallelizability of an RF. This may be explained conceptually: An RF involves training multiple trees. Each subtree or branch can be trained independently on a separate isolated thread or even worker. Thus, less communication overhead is necessary in comparison to LR, where mini-batch gradient descent or limited-memory Broyden-Fletcher-Goldfarb-Shannon [87] require more communication between threads or worker to share, e.g., weight updates of the LR.

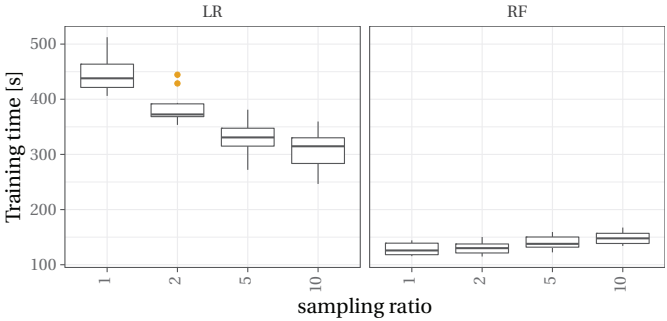


Figure 6.6: Evaluation of different sampling ratios regarding the training time on the *Credit Card* data set.

Therefore, to evaluate the feature selection and imbalance techniques in the following, a sample ratio of 5 to reduce the size of the training data set will be used in combination with an RF. The testing data set will not be subsampled.

Feature Selection

As argued above, feature selection techniques will be mainly evaluated on the *NIPS* data sets. Wrapper measures as introduced in Section 3.1.3 were not considered, they are too time consuming in the given cluster computing scenario. For comparison, the results visualized in Figure 6.7 are enhanced by the *Automotive 1L* data set (which uses a different scaling of the ordinate). Due to the limited computing power available, 40 representative modeling targets from the *Automotive 1L* were selected[2].

[2]Aside from the above mentioned requirement of 20 positive samples, the strategy was as follows: First, all remaining potential targets were divided in four quartiles in dependence on the number of available samples. Then, from each quartile, ten samples were randomly chosen.

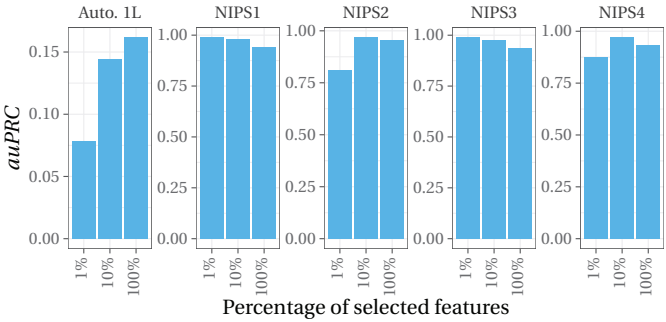


Figure 6.7: Influence of feature selection on the auPRC of an RF.

If only the 1% highest ranked features are selected, the classification grade (auPRC) cleary suffers in multiple cases (*Automotive 1L*, *NIPS2*, *NIPS4*). This can be explained by the fact that many of actual informative features do not make it into the training set, thus wasting information. The other extreme, selecting all features (100%) and not performing any feature selection at all, can also harm the classification grade. This can be observed on the *NIPS1*, *NIPS3*, and *NIPS4* data sets. This is an important thing to note which underpins the results from Chapter 3. Neither selecting *all* features, nor only 1% can be advised. Thus, selecting 10% marks the best trade-off in terms of auPRC since this value yields consistent high auPRC scores across all evaluated data sets. However, this can only be applied to other scenarios to a limited extent. Instead, it is recommended to follow the procedure described in Chapter 3.

Next, the influence on the training duration is investigated. Figure 6.8 shows the training durations for both classifiers, different selection percentages, the computation time of the various feature selection techniques, and (if available) the individual computation times of the selection techniques. Please note the different ordinate scales: The LR is faster on the NIPS data sets in terms of training duration. However, on the *Automotive 1L* data set, the training durations are roughly equal. This is an interesting observation: While the LR provided by the `MLlib` is unaffected the number of features, `MLlib`s implementation of the RF requires more time for training as the number of features increases. Combining LR with feature selection techniques makes no sense from the training time perspective: Mainly due to the correlation based (Section 3.1.2) filter, the overall time it takes to create a model is always lower, if no feature selection takes place. The time it takes for the feature selection techniques to compute does not outweigh the savings in terms of model training time.

However, feature selection speeds up RF training: Computing feature rankings and training the model on the 1% or 10% subset of the features, is always faster on the *NIPS* data sets. Thus, combining an RF with feature selection can influence the model creation positively.

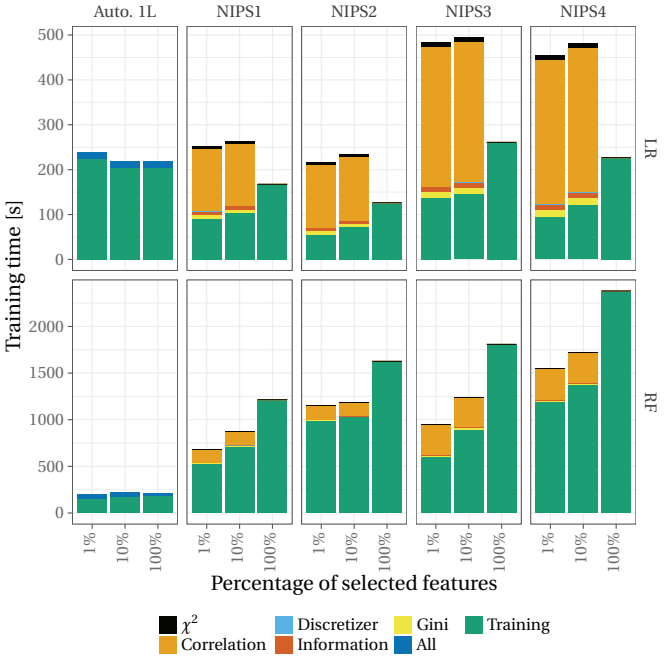


Figure 6.8: Influence of feature selection on the training time.

Figure 6.8 also shows, that the correlation takes significantly longer to compute than all other filter measures. The respective proportions, averaged over all *NIPS* data sets are visualized in Figure 6.9. Calculating the correlation measure takes 152*s* on average. This equals to 89% of the 171*s* total average time to compute all filter measures. For comparison, the training takes 149s (991s) on average for the LR (RF). Unlike "Information" (MI) and the Gini Measure, which were calculated using discretized features, this was not the case for correlation. Thus, every value of every feature-target combination had to be compared, causing a lot of communication overhead and slowing down the computation.

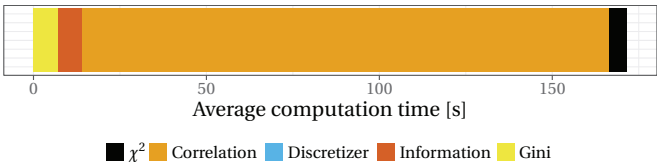


Figure 6.9: Timings of various filter measures in Spark.

Imbalance

This section briefly evaluates SMOTE [29] in a big data setup. SMOTE is currently the by far most popular (in terms of citations) and most promising (according to Section 4.1.1) technique to tackle imbalanced data sets.

Figure 6.10 shows the percentage of models on the ordinate that performed above a given threshold on the abscissa. Both, auPRC and auROC values are plotted for comparison. Also, the random strategy (where only random subsampling takes place) is compared to a combination of upsampling by SMOTE and random subsampling. SMOTE consistently yields a slightly higher percentage of models that perform above the respective threshold. This holds true for *any* given threshold and both measures, although the influence is more visible and more meaningful based on the auPRC measure.

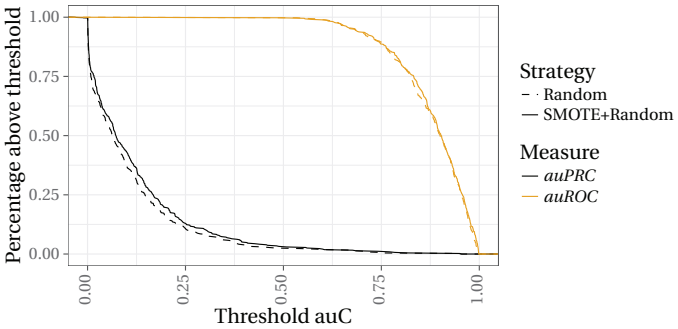


Figure 6.10: Evaluation of *Automotive 1L* data set.

Although measurable, the classification performance improvements caused by SMOTE are not in any relation to the drastically increased processing times. On average, models trained without SMOTE based on *Automotive 1L* took 210 *s*. If SMOTE is applied, "SMOTing" and training of a RF yielded a combined training time of 511 *s*, which is more than doubled.

This effect is further worsened by the fact that that "SMOTing" itself raises the sampling time by 42s on average in comparison to just performing undersampling (14s).

A Brief Note on Scalability

This section briefly elaborates, whether Spark is suitable to process the by far biggest automotive diagnostic data set used in this work (*Automotive 2L*). The aforementioned computation times (see, e.g., Figure 6.6, Figure 6.8, or Figure 6.9) suggest that tuning hyperparameters on a data set of this size is not feasible. Although slow, processing and model creation were possible and yielded the following durations:

Table 6.3: Durations on *Automotive 2L* data set.

Training strategy	Sampling time [s]	Training time [s]
Random	703.4	1664.8
SMOTE+Random	2062.9	3757.4

As Table 6.3 shows, model building is possible, despite being slow. Artificially upsampling the minority class to five times the samples using SMOTE raises the sampling time by about 1359s. In both cases (Random and SMOTE+Random), the majority class is subsampled as described above (the class balance is 1 : 5 in both cases). Upsampling the minority class also increases the time it takes to create the RF model to model the target DTC of *Automotive 2* by more than 100%.

6.5 Brief Economic Analysis

All measures introduced in Section 2.3 aim to assess the classifiers' ability to distinguish between classes, but do not consider economic aspects. However, this is an important step before deploying a machine learning system [31]. This section aims to provide fundamental thoughts that should be considered when investigating this matter further (which is outside the scope of this work).

In general, accurately measuring the *cost* assigned to prediction is very hard in the given scenario. Customer dissatisfaction caused by wrong or too expensive repairs is hard to measure, laws differ between countries which can cause sometimes the company and sometimes the customer to be liable for the same issue, etc. Also, the proposed techniques aim at performing multiple binary classifications for all potential parts and actions. While a single error

(predicting a slightly lower score to the part that should be switched) does not do any harm when being evaluated in isolation, but may cause a different outcome if combined with other errors (predicting a slightly higher score on the wrong part).

Based on Table 2.1, predictions can be separated into the following categories:

- FP: Predicting a defect to be present although it is not.
- TP: Predicting a defect to be present which is indeed the case.
- FN: Predicting a defect not to be present although it is.
- TN: Predicting a defect not to be present which is indeed the case.

These categories can be further refined based on whether the defect was noticeable by the customer or latent (not noticeable), which is shown in Table 6.4, where + is referring a positive cost, − is referring a negative cost (cost savings), and · is referring no cost:

Type	cost	
	noticeable	latent
FP	++	+
TN	−	−
FN	+	·
TP	−	−

Table 6.4: Cost assessment of different prediction outcomes.

Predicting a FP when a defect is noticeable is most costly outcome: Both, switching the wrong part and leaving a dissatisfied customer needs to be avoided. If it is a latent defect, the cost reduces to just switching the wrong part. Predicting TP and TN is desirable, reducing the overall cost. A FN yields costs for potentially dissatisfied customers if the defect is noticeable and no costs for a latent error.

This can be formalized as follows: FPs are the most expensive errors. Thus, using the auPRC to assess the classification performance as laid out in Section 2.3 seems suitable because it penalizes a high number of FPs (via the precision). To identify the economically most viable repair, the following formula is proposed:

$$score = pred_{repair} \cdot \left(1 - \frac{c_{repair}}{\sum_i^R (c_i)}\right) \cdot \frac{auPRC - auPRC_{min}}{1 - auPRC_{min}}, \tag{6.2}$$

with $pred_{repair}$ being the prediction for the currently evaluated repair, c_{repair} the cost that is caused by performing the repair, $\sum_i^R(c_i)$ the summed cost for all possible repairs, followed by the auPRC scaled to a [0, 1] interval to allow comparison of different data sets with difference balance ratios. Equation (6.2) formalized the following intuitions to identify the economically best repair (highest *score*):

1. A high $pred_{repair}$: This indicates that the corresponding action or part is likely to resolve the issue.

2. A low cost of the repair c_{repair}: The lower the cost, the better. If, e.g., two actions are equally likely to resolve the issue ($pred_{repair}$), the action with lower associated cost should be preferred.

3. A reliable model indicated by *auPRC*: The higher the auPRC during testing, the more trustworthy the predictions of the model are. However, since the auPRC is dependent on the balance ratio $\aleph = \frac{N_{\mathscr{I}}}{N}$ of the data set ($N_{\mathscr{I}}$ is the number of minority class samples, see Chapter 4), special care is needed.

To make the auPRC ratings of multiple models comparable, the auPRC must be normalized since the auPRC depends on the class balance ratio (which varies across data sets). To do so, the minimum possible auPRC is required for each data set, respectively. The minimum possible auPRC is defined using the balance ratio $\aleph = \frac{N_{\mathscr{I}}}{N}$ ($N_{\mathscr{I}}$ is the number of minority class samples, see Chapter 4) [18]:

$$auPRC_{min} = 1 + \frac{(1-\aleph)(\ln(1-\aleph))}{\aleph} \tag{6.3}$$

This can be explained as follows (for more details, please refer Boyd et al. [18]): Recall and precision both depend on the number of TPs (known from the confusion matrix). Thus, not every point in the precision-recall space is reachable based on a valid confusion matrix (e.g., all values need to be positive). The following inequality holds for the dependence of precision and recall [18]:

$$precision \leq \frac{\aleph \cdot recall}{1-\aleph+\aleph \cdot recall}. \tag{6.4}$$

The above inequality implies that, for a given balance ratio $\aleph$, every model must produce a PRC that lies *above* the minimum possible PRC [18].

6.6 Conclusion

This chapter evaluated the suitability of the most popular in-memory cluster computing framework available today regarding its suitability to drive a complete large scale machine learning pipeline. The pipeline was designed to satisfy the *generic* requirement. The pipeline included all steps, from the extraction of the data set from a relational database, to model building, and finally presenting the gained insights to an end user.

This prototypical, but comprehensive implementation yielded several interesting impressions. On the one hand, Spark has made it possible to work with large scale data sets that simply would not have been processable with conventional single-node frameworks. This applies in particular to the transformation of large data sets from key-value into the columnar layout required for machine learning. However, the the downsides should not be underestimated.

First and foremost, Spark's architecture is not suitable to evaluate a single sample using a high magnitude of models. Currently, each sequential classification of a single sample takes $\approx 10s$. Parallelizing the classification is currently not possible using the native Spark APIs and would require to instantiate multiple master nodes. This, however, contradicts the idea of having a single entry point to utilize the computing power of a whole cluster. Another option to tackle this issue is to create one multi-class model instead of multiple two-class models. However, this poses new challenges in terms of the (economic) evaluation of classification quality for future research.

Also, the extensive manual tuning required to ensure the computations to finish in a timely manner should be noted. Decisions regarding the optimal number of partitions, when and how to persist, etc. should be well considered. To ease the development, future research should examine how to perform more of the just mentioned decisions automatically.

In general, it can not be recommended to use Spark for extracting, transformating, or modeling unless computations fail or are impossible due to memory constraints on single-node frameworks. Doing transformation, feature selection and downsampling on the cluster, and machine learning on a workstation should be considered as a viable option.

In a nutshell: Spark is trading off performance for scalability when compared to single-node frameworks.

Chapter 7

Summary and Recommendations for Future Work

This chapter aims to summarize what was done and to point out the most important results. Also, the findings will be discussed and recommendations for further research derived.

7.1 Summary, Conclusions, and Discussion

Chapter 2, or more specifically Section 2.3, compared two promising measures to assess the classification performance in imbalanced scenarios in great detail: The auROC and the auPRC. The auPRC turned out be more meaningful and sensitive according to own tests on artificial data, and is also considered to be the better measure in literature [101, 57, 36]: A classifier, that dominates in terms of auPRC will always dominate in terms of auROC. This does not hold vice versa. The auPRC yielded promising results in the following chapters, even rendering techniques to tackle imbalance are superfluous, that would have been required if other measures were used. Therefore, objective 1 (the definition of a meaningful error score for imbalanced scenarios) can be considered done. However, the auPRC is dependent on the balancedness of the data set. Thus, if the performance on multiple data sets of different balancedness shall be compared, special care is required: In this case, the auPRC either needs to be normalized as described in Section 6.5 or an otherwise inferior (according to Section 2.3) metric such as the auROC must be used.

In Chapter 3, a feature selection pipeline with multiple layers of differing run-time complexity was proposed. A wide range of classifier independent filter measures, and classifier

dependent wrapper measures were evaluated in respect to the achieved auPRC and the run-time complexity. The χ^2 filter is best suited to identify important features in a reasonable time. The LR wrapper also delivered promising results. However, it should be noted that certain combinations do not reduce the overall training time: If, for example, LR is used, a single χ^2 filter requires more time to select the features than the model training itself. However, this ratio can be tilted if the model training is repeated multiple times after a single feature selection, or if slower models, such as an RF, are used. The pipeline achieved perfect classification or comparable results without manual adjustment on publicly available data sets, so objective 2 (the combination and adaptation of techniques that are able to select relevant features from a high dimensional feature space) can be considered completed. It has to be noted that this chapter focused on feature subset selection only, to ensure that final models are interpretable by experts. Techniques that transform the features into a different feature space such as PCA were not evaluated but may yield promising results.

In total, five of the most popular techniques to tackle imbalanced data sets and a novel approach were evaluated in Chapter 4. All existing techniques focus on upsampling the minority class, or removing noisy samples. In contrast, the proposed approach aims to reduce noise close to the decision boundary by scaling samples to their corresponding class center, which is multiple orders of magnitude faster. The evaluation was laid out in terms of modeling grade (primarily auPRC) and run-time complexity on nine publicly available and one proprietary automotive data set. According to the experiments, SMOTE is yielding good results in terms of auROC and a reasonably low computation time. SMOTE, and the novel approach were the only approaches, where the beneficial influence on the auROC was provable and the computation time reasonable. This satisfies objective 3 (the development and comparison of techniques to tackle the high imbalance of the automotive and other, publicly available data sets). However, if model hyperparameters are tuned with respect to the auPRC (as discussed in Section 2.3) and the final modeling grade is also assessed in auPRC, no technique can be proven to be effective based on the experiments laid out in Section 4.4. This suggests that selecting the optimal measure may be the better option in general.

For Chapter 5, two novel RUL estimation techniques were designed, implemented and evaluated in comparison to a naive, RF based approach and an approach proposed by Wang et al. [136]. The approach by Wang scored first place in a competition on one of the most popular RUL data sets and thus marks the baseline. The novel techniques are based on cas-

caded RFs and on density based estimation. Especially the density based approach was *extremely* promising and has not been published yet: It yielded the lowest error scores on both diagnostic data sets[1] and the by far fastest times if solely training is considered. Also, the RF based and Wangs approach dominated on some data sets. Unfortunately, the current implementation of Wangs approach is very slow. The fact that the code used in this work was open sourced allows other researchers to optimize it. According to the experiments, the density based approach in particular copes very well with short and varying length time series. In addition, the insights obtained in this chapter make it possible to select the appropriate approach depending on the application. All in all, objective 4 (the development of techniques to accurately estimate the RUL based on short, variable length time series) can thus be considered satisfied.

Chapter 6 strongly builds on the knowledge gained so far. The most promising techniques for feature selection and tackling imbalance were implemented using `Scala` to create an end-to-end machine learning pipeline based on Apache Spark. Despite being an in-memory, cluster computing framework, data set transformation and model training are much slower in comparison to single node frameworks[2]. In addition, many caveats surfaced that reduce the usefulness further. It has to be noted though, that the largest data set used in this work was only transformable from its original key-value pair into the columnar layout required for machine learning using Apache Spark. Due to the slower computation and the many caveats, Spark can only be recommended if the data set can not be processed by conventional means. Even then, after the transformation, further processing on a single PC with a conventional single-node framework should be considered. As concluded in Chapter 6: "Spark is trading off performance for scalability when compared to single-node frameworks." (objective 5, the identification and adaptation of a multi node in-memory computing frame work to deal with large data volumes).

All techniques discussed above were evaluated on a variety of data sets to satisfy objective 6 (solutions shall not be restricted to a single scenario) and compared regarding their classification performance *and* computational complexity as demanded by objective 7.

[1] On other data sets comparing scores were achieved.
[2] This is subject to the condition that the data fits into the RAM of a single computer.

7.2 Recommendations for Further Work

Detailed recommendations regarding future research directions for the topics addressed in the previous chapters are already given in the respective concluding sections respectively. This section will thus focus on the proposed solutions that are more broadly applicable.

Currently, labels for training the models are created by using existing workshop processes as ground truth. However, this can be misleading, since, e.g., the exchange of a wrong part cannot be excluded. The customer concerned will return promptly and ask for the problem to be rectified, which means that the right action dominates the data in the long term. This process, however, is not optimal, as it generates additional profits for the workshop and unnecessary costs for the manufacturer (provided the part is under warranty). Thus, a new form of incentivisation is required, that not only encourages workshop staff to perform the most cost-effective countermeasure but also reward them if helpful feedback such as "this was the wrong repair" is provided to increase the overall label and thus data quality.

Apart from Chapter 5, this work has dealt with the isolated classification of workshop operations. In the future, however, the trend towards more data will also find its way into the automotive industry. Due to the availability of data with a higher sampling rate, i.e. a smaller time interval between observation points, time series techniques (as already indicated in Chapter 5) will become applicable. The consideration of the temporal progressions of individual vehicles has the potential to add great value and should therefore be investigated. Chapter 5 can provide clues. Also, applying Deep Learning may become feasible.

In the context of Deep Learning – where in general more training samples are required due to the higher number of tunable model parameters in comparison to conventional learning – a different approach how to utilize the available data may be worth to be investigated: Currently, e.g., each part or maintenance action is modeled based on a subset of data which originates only from cars of the same model with the same engine. This, however, causes two major caveats: First, summed over the whole fleet, more models need to be built (since the same part is modeled for each car–engine combination individually). Second, since the data set is split into multiple sub-data sets, less data is available for each model training process. It is therefore recommended to examine which models can be trained across multiple vehicles and engines. This increases the amount of training data and reduces the number of models.

Section 6.5 already laid out a brief economic analysis. This, however, should be evaluated

in much greater detail before deploying an autonomous, machine learning based diagnostic system into productive use. The cost of different classification outcomes, customer satisfaction, etc. is hard to assess and should be further investigated and quantified.

Appendix A

Spark Cluster Setup and Application Launch

Applications are started on the cluster using the `spark-submit` Command Line Interface (CLI). Aside from the `--master` argument (specifying the IP of the clusters masternode), and executable (`.jar`) to start, additional parameters are required to define the resources that will be used for the job. The parameters (and the used values) are:

- `driver-cores`: Number of cores reserved for the `Driver` process (5).
- `executor-cores`: Number of cores for each `Executor` process (5).
- `num-executors`: The total number of `Executor` processes (11)[1].
- `spark.driver.memory`: RAM reserved for the `Driver` process in GB (15).
- `executor-memory`: RAM assigned to each `Executor` process in GB (14).
- `spark.driver.maxResultSize`: Since actions fail, if `spark.driver.maxResultSize` is exceeded by the result size, this number was increased. It has to be smaller than `spark.driver.memory`, though (10).

This way, based on the $92GB + 96GB = 188GB$ available RAM, $14 \cdot 11GB + 15GB = 169GB$ (89.9%) were used. Also, given $11 \cdot 5 + 5 = 60$ (93.8%) of the 64 cores were used. The above mentioned values were determined based on the following guidelines:

[1] For experiments on a single machine, only `num-executors` was reduced to 6.

- If `executor-cores` is too high, HDFS throughput is reduced because the HDFS has trouble with too many concurrent threads. HDFS achieves full write throughput with 5 tasks per executor [51].
- 2 cores and 5GB are reserved on each machine for the operating systems, HDFS and the various daemons to run smoothly.
- Spark lives within the JVM. If the amount of memory for each `Executor` process is too big, Garbage Collection (GC) takes up a considerable amount of time. Using the settings described above, GC always took less than 10% of the total execution time.

List of Figures

2.1 Example of a DT. 12
2.2 Model that identifies the 100 real positives samples without error. Imbalance 1 : 1000. 19
2.3 Poor model. Imbalance 1 : 1000. 19
2.4 Worse than poor model. Imbalance 1 : 100. 20
2.5 Catastrophic model. Imbalance 1 : 10. 20

3.1 The χ^2 distribution. 31
3.2 Overview of the feature selection pipeline. 38
3.3 Example why standard outlier filters fail, μ is marked as solid, $3 \cdot \sigma$ as dashed line. 40
3.4 Average model performance depending on target type. 44
3.5 Number of features used in final model above threshold. 51
3.6 Computation time: k-NN, RF, LR, and all filter measures. 53

4.1 Comparison of the original data set before and after applying SMOTE. 67
4.2 Comparison of the original data set before and after applying ADASYN. 69
4.3 Comparison of the original data set before and after the removal of Tomek links. 70
4.4 Comparison of the original data set and the consistent subset according to CNN. 71
4.5 Comparison of the original data set and the consistent subset according to OSS. 72
4.6 Demonstration of CSS with artificial data. 74
4.7 CSS on Vowel data set. 75
4.8 Comparison of the original data set and scaled data set. 76
4.9 Comparison of different CSS modes based on auPRC calculated during CV. . . 80
4.10 Comparison of different CSS modes based on auPRC averaged across all classifiers. 81
4.11 Clustering of data sets. 82

4.12 Comparison of a robust and non–robust PT-classifier combination. 83
4.13 Test versus train auPRC of SMOTE and CNN. 84
4.14 Achieved auPRC based on the data set cluster, PT, and classifier. 85
4.15 Distributions of the tested error scores. 88
4.16 Computation time of different PTs. 90
4.17 Training time of various classifiers dependent on the preprocessing technique. 91

5.1 Example for a model which is derived from a time series. 103
5.2 Visualization of mutliple models. 103
5.3 Example for a model using Wangs technique before the history is shifted. 104
5.4 Example for a model using Wangs technique after shifting the history by the optimum number of hours. 104
5.5 Example of two features holding different amounts of information. 106
5.6 Visualization of the key ideas of DBSE2. 107
5.7 Different RUL areas. 111
5.8 Effects of performing PBFG for DBSE2. 114
5.9 Evaluation of different n_{grid} combinations. 116
5.10 Influence of n_{hist}. 117
5.11 Influence of weighting risk values based on temporal proximity. 118
5.12 Effects of discarding samples based on their RUL. 119
5.13 Effects of splitting the training according the RUL. 120
5.14 Effects of adding historical samples to the feature space on RMSE and s_{PHM}. . . 121
5.15 Effects of using polynomial feature selection on s_{PHM}. 121

6.1 The Spark cluster architecture. 128
6.2 Overview over the pipeline. 132
6.3 Prototypical implementation of the user interface. 139
6.4 Performance comparison for different numbers of partitions. 143
6.5 Evaluation of different sampling ratios based on auPRC on the *Credit Card* data set. 146
6.6 Evaluation of different sampling ratios regarding the training time on the *Credit Card* data set. 147
6.7 Influence of feature selection on the auPRC of an RF. 148

6.8 Influence of feature selection on the training time. 149
6.9 Timings of various filter measures in Spark. 150
6.10 Evaluation of *Automotive 1L* data set. 150

List of Tables

1.1 Example data set for automotive diagnostics. 4

2.1 Confusion matrix. 17

3.1 Examples for different feature values and their corresponding Gini coefficients. 27
3.2 Sample data. 30
3.3 Contingency table. 30
3.4 Expected frequencies. 30
3.5 Chi-square points. 30
3.6 Feature group importance. 46
3.7 Filter measure performance. 48
3.8 Influence of the wrapper algorithm on the pipeline performance. 50
3.9 Influence of n_{FAF} on pipeline performance. 51
3.10 Influence of n_{FAW} on model performance and training time. 52
3.11 Influences of *NPR* on the pipeline performance and training time. 52
3.12 Overview over the layer runtimes in seconds depending on the number of processed features. 55
3.13 Performance of the pipeline on the *golub* data set. 56
3.14 Performance of the pipeline on the *secom* data set. 56
3.15 Overview of pipeline performance and runtime using an RF wrapper and auROC. 58
3.16 Overview of pipeline performance and runtime using an RF wrapper and auPRC. 58
3.17 Overview of pipeline performance and runtime using an RF wrapper and F_1. . 58
3.18 Overview of pipeline performance and runtime using an LR wrapper and auROC. 58
3.19 Overview of pipeline performance and runtime using an LR wrapper and auPRC. 59
3.20 Overview of pipeline performance and runtime using an LR wrapper and F_1. . 59

4.1 Overview of the evaluated data sets. 77
4.2 RMSD of training and test auPRC. 84
4.3 RMSD of training and test auROC. 86
4.4 Statistical comparison of the cluster mean F_1 from PTs compared to the naive approach based on the p–value. 88
4.5 Statistical comparison of the cluster mean auROC from PTs compared to the naive approach based on the p–value. 89
4.6 Statistical comparison of the cluster mean auPRC from PTs compared to the naive approach based on the p–value. 89

5.1 Overview on evaluation data sets. 99
5.2 Example for different $n_{history}$ modes. 112
5.3 Performance comparison of all algorithms in terms of RMSE and s_{PHM} on all data sets. 122
5.4 Training and testing time of the best performing model for all technique-data set pairs. 123

6.1 Details on large *Automotive* data sets. 144
6.2 Details on the *NIPS* data set. 145
6.3 Durations on *Automotive 2L* data set. 151
6.4 Cost assessment of different prediction outcomes. 152

Acronyms

ADASYN ADaptive Synthetic Sampling Approach. 69–71, 79, 81, 87, 88, 90, 92–96, 168

ANN Artificial Neural Network. 23, 66, 100

API Application Programmable Interface. 131, 132, 138, 139, 142, 144, 160

ARMA Autoregressive Moving Average Model. 100

ASCII American Standard Code for Information Interchange. 145

auPRC area under Precision Recall Curve. 18, 20, 43, 45, 52–55, 58, 60–63, 82–93, 95, 96, 131, 141, 143, 151–155, 158, 159, 161, 162, 168–172

auROC area under Receiver Operating Characteristics Curve. 18–20, 45, 52, 54, 60, 61, 83, 87, 88, 91–93, 95, 96, 155, 161, 162, 171, 172

bagging bootstrap aggregating. 12

BRR Bucketized RUL regression with trend-based feature selection. 105, 113, 114, 117, 121, 124, 126–128

CLI Command Line Interface. 166

CNN Condensed Nearest Neighbor. 72–74, 87, 88, 90, 92–96, 168, 169

CP Car Parameter. 4, 38, 39, 47, 48, 149

CSS Class Sensitive Scaling. 65, 74–77, 83–85, 87, 88, 90–92, 94–96, 168, 169

CSV Comma Separated File. 147, 150

CV Cross-Validation. 17, 75, 80, 82–84, 86, 91, 139, 141, 150, 168

DAG Directed Acyclic Graph. 134, 145

DBSE2 Distribution-based Similarity Estimation v.2.. 105, 110–112, 116–118, 126–128, 169

DC Diagnostic Code. 37, 45, 149

DC Diagnostic Code. 5, 137

DT Decision Tree. 11–13, 168

DTC Diagnostic Trouble Code. 4, 24, 37, 38, 47, 48, 135, 137, 148, 149, 157

EC Environmental condition. 4, 38, 47, 149

ECU Electrical Control Unit. 4, 24, 37, 40

EE Extra Equipment. 4, 38, 39, 47, 149

EOL End Of Life. 102, 114

F_1 $F_{\beta=1}$. 17, 20, 45, 52–55, 58, 60–62, 83, 91–93, 171, 172

FG feature group. 46–48

FN False Negative. 18, 157, 158

FP False Positive. 18, 55, 157, 158

GC Garbage Collection. 167

GLM Generalized Linear Model. 14

GUI Graphical User Interface. 142, 143

HDFS Hadoop Distributed Filesystem. 133, 142, 167

HI Health Indicator. 98, 99, 101, 105–109

IBU International Bitter Unit. 12

IoT Internet of Things. 129

IPA India Pale Ale. 12

JVM Java Virtual Machine. 145, 167

k-NN k-nearest neighbors. 16, 17, 34, 36, 43, 44, 52, 55, 56, 58–60, 64, 65, 74, 82, 83, 88, 90, 91, 94, 101, 168

KDB Knowledge Database. 3

LR Logistic Regression. 14, 15, 17, 23, 34–36, 42–44, 46, 47, 49, 52, 53, 55–62, 79, 82, 83, 90, 100, 105, 106, 139, 141, 151, 152, 154, 162, 168, 171, 172

MDL Minimum Description Length. 27

MI Mutual Information. 24, 25, 27, 31, 51, 56, 154

MLE maximum likelihood estimation. 14

MV Measurement Value. 5, 38, 46–48, 135, 149

NID noisy, imbalanced data set. 65–67, 76, 78

NTF No Trouble Found. 3

OCC One-Class Classifier. 65, 66, 82, 90, 91, 94, 96

OEM Original Equipment Manufacturer. 3, 4, 6

OHE One Hot Encoding. 136

OSS One-Sided Selection. 73, 75, 79, 87, 88, 90, 93, 95, 96, 168

PBFG Polynomial-Based Feature Grading. 109, 113, 116–118, 124, 169

PCA Principal Component Analysis. 23, 76, 145, 150, 162

PHM prognostics and health management. 99, 100, 103, 105

PMML Predictive Model Markup Language. 146

PRC Precision Recall Curve. 18–20, 159

PT preprocessing technique. 65, 78, 79, 83, 86–96, 169, 172

RAM Random Access Memory. 130, 132, 134, 144–146, 163, 166

RDD Resilient Distributed Data Set. 133

RF Random Forest. iii, 11–13, 23, 34, 35, 43, 44, 52, 55, 56, 58–63, 79, 82, 83, 90, 96, 102, 105, 114, 115, 117, 123, 124, 126–128, 139–141, 151–154, 156, 157, 162, 163, 168, 170, 171

RMSD Root-Mean-Square Difference. 86–88, 172

RMSE Root Mean Squared Error. 103, 105, 107, 109, 115–118, 121–123, 125–127, 169, 172

RNN Recurrent Neural Network. 100

RO Readout data. 4, 38, 39, 47, 48, 149

ROC Receiver Operating Characteristic. 18–20

RUL Remaining Useful Lifetime. iii, 6–9, 97–110, 112–117, 121, 122, 124, 126, 128, 129, 163, 169

RUS Random Under-Sampling. 137

SMOTE Synthetic Minority Oversampling TEchnique. 68–70, 78, 83, 87, 90–96, 137, 154–156, 162, 168, 169

SP Switched Part. 5, 45, 149

SQL Structured Query Language. 132

SSD Solid State Drive. 145

SVM Support Vector Machine. 66, 82, 100

SVR Support Vector Regression. 101

TA Taken Action. 5, 45, 149

TN True Negative. 18, 55, 157, 158

TP True Positive. 18, 157–159

UDF User defined function. 138

VIN Vehicle Identification Number. 142

WLR Weighted Logistic Regression. 82, 90, 91, 96

WRF Weighted Random Forest. 82, 90, 96

YARN Yet Another Resource Negotiator. 132

Bibliography

[1] R. Ahmed, M. E. Sayed, S. A. Gadsden, J. Tjong, and S. Habibi. "Automotive Internal-Combustion-Engine Fault Detection and Classification Using Artificial Neural Network Techniques". In: *IEEE Transactions on Vehicular Technology* 64.1 (2015), pp. 21–33.

[2] M. Aminanto, R. Choi, H. C. Tanuwidjaja, P. D. Yoo, and K. Kim. "Deep Abstraction and Weighted Feature Selection for Wi-Fi Impersonation Detection". In: *IEEE Transactions on Information Forensics and Security* 13.3 (2018), pp. 621–636.

[3] F. Arif, N. Suryana, and B. Hussin. "A Data Mining Approach for Developing Quality Prediction Model in Multi-Stage Manufacturing". In: *International Journal of Computer Applications* 69.22 (2013), pp. 35–40.

[4] A. Azarian and A. Siadat. "A global modular framework for automotive diagnosis". In: *Advanced Engineering Informatics* 26 (2012), pp. 131–144.

[5] S. Barua, M. M. Islam, X. Yao, and K. Murase. "MWMOTE–majority weighted minority oversampling technique for imbalanced data set learning". In: *IEEE Transactions on Knowledge and Data Engineering* 26.2 (2014), pp. 405–425.

[6] G. E. A. P. A. Batista, A. L. C. Bazzan, and M. C. Monard. *Balancing Training Data for Automated Annotation of Keywords: a Case Study*. 2003. URL: http://www.inf.ufrgs.br/maslab/pergamus/pubs/balancing-training-data-for.pdf (visited on 07/16/2018).

[7] BearingPoint GmbH. *Global Automotive Warranty Survey Report*. URL: https://www.bearingpoint.com/files/AutoWarrantyReport_final_web.pdf (visited on 01/01/2009).

[8] C. M. Bishop. *Pattern recognition and machine learning*. Information science and statistics. New York: Springer, 2006. ISBN: 978-0387310732.

[9] BMW A.G. *BMW Geschäftsbericht 2011*. URL: https://www.bmwgroup.com/content/dam/bmw-group-websites/bmwgroup_com/ir/downloads/de/2011/bericht2011.pdf (visited on 07/16/2018).

[10] BMW A.G. *BMW Geschäftsbericht 2012*. URL: https://www.bmwgroup.com/content/dam/bmw-group-websites/bmwgroup_com/ir/downloads/de/2012/bericht2012.pdf (visited on 07/16/2018).

[11] BMW A.G. *BMW Geschäftsbericht 2013*. URL: https://www.bmwgroup.com/content/dam/bmw-group-websites/bmwgroup_com/ir/downloads/de/2013/geschaeftsbericht2013.pdf (visited on 07/16/2018).

[12] BMW A.G. *BMW Geschäftsbericht 2014*. URL: https://www.bmwgroup.com/content/dam/bmw-group-websites/bmwgroup_com/ir/downloads/de/2014/12507_GB_2014_de_Finanzbericht_Online.pdf (visited on 07/16/2018).

[13] BMW A.G. *BMW Geschäftsbericht 2015*. URL: https://www.bmwgroup.com/content/dam/bmw-group-websites/bmwgroup_com/ir/downloads/de/2015/12784_GB_2015_dt_Finanzbericht_Online.pdf (visited on 07/16/2018).

[14] BMW A.G. *BMW Geschäftsbericht 2016*. URL: https://www.bmwgroup.com/content/dam/bmw-group-websites/bmwgroup_com/ir/downloads/de/2017/GB/13044_BMW_GB16_de_Finanzbericht.pdf (visited on 07/16/2018).

[15] V. Bolón-Canedo, N. Sánchez-Maronno, and A. Alonso-Betanzos. "A review of feature selection methods on synthetic data". In: *Knowledge and Information Systems* 34.3 (2005), pp. 483–519.

[16] J. Bonér. *Latency Comparison Numbers*. 2016. URL: https://gist.github.com/jboner/2841832 (visited on 07/16/2018).

[17] S. Boslaugh. *Statistics in a Nutshell, 2nd Edition*. O'Reilly Media, Incorporated, 2012. ISBN: 9781449361129. URL: https://books.google.de/books?id=s1llAQAACAAJ.

[18] K. Boyd, V. S. Costa, J. Davis, and C. D. Page. "Unachievable region in precision-recall space and its effect on empirical evaluation". In: *Proceedings of the International Conference on Machine Learning* (2012), p. 349.

[19] L. Breiman. "Random forests". In: *Machine Learning* 45.1 (2001), pp. 5–32.

[20] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Vol. 19. CRC Press, 1984. ISBN: 978-0412048418.

[21] L. Breiman. "Bagging predictors". In: *Machine Learning* 24.2 (1996), pp. 123–140.

[22] L. Breiman. "Technical Note: Some properties of splitting criteria". In: *Machine Learning* 24.1 (1996), pp. 41–47.

[23] S. Breker, A. Claudi, and B. Sick. "Capacity of Low-Voltage Grids for Distributed Generation: Classification by Means of Stochastic Simulations". In: *IEEE Transactions on Power Systems* 30.2 (2015), pp. 689–700.

[24] P. Bruce and A. Bruce. *Practical Statistics for Data Scientists: 50 Essential Concepts*. Boston: O'Reilly Media, Inc, 2017. ISBN: 978-1491952962.

[25] W. Caesarendra, A. Widodo, and B.-S. Yang. "Application of relevance vector machine and logistic regression for machine degradation assessment". In: *Mechanical Systems and Signal Processing* 24.4 (2010), pp. 1161–1171.

[26] L. Čehovin and Z. Bosnić. "Empirical evaluation of feature selection methods in classification". In: *Intelligent Data Analysis* 14.3 (2010), pp. 265–281.

[27] B. Chandra and R. K. Sharma. "Exploring autoencoders for unsupervised feature selection". In: *International Joint Conference on Neural Networks* (2015), pp. 1–6.

[28] N. V. Chawla, N. Japkowicz, and A. Kotcz. "Editorial: special issue on learning from imbalanced data sets". In: *ACM Sigkdd Explorations Newsletter* 6.1 (2004), pp. 1–6.

[29] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. "SMOTE: Synthetic minority over-sampling technique". In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357.

[30] N. V. Chawla, A. Lazarevic, L. O. Hall, and K. W. Bowyer. "SMOTEBoost: Improving prediction of the minority class in boosting". In: *Lecture Notes in Computer Science* (2003), pp. 107–119.

[31] J. Cleve and U. Lämmel. *Data Mining*. Berlin, Boston: De Gruyter Oldenbourg, 2016. ISBN: 978-3110456752.

[32] P. Cortez and A. J. R. Morais. "A data mining approach to predict forest fires using meteorological data". In: *Proceedings of the Portuguese Conference on Artificial Intelligence* 13 (2007), pp. 512–523.

[33] H. Cramer. *Mathematical methods of statistics*. Princeton University Press, 1946. ISBN: 978-0691005478.

[34] A. Dal Pozzolo, O. Caelen, R. A. Johnson, and G. Bontempi. "Calibrating probability with undersampling for unbalanced classification". In: *IEEE Symposium Series on Computational Intelligence* (2015), pp. 159–166.

[35] C. Damgaard and J. Weiner. "Describing inequality in plant size or fecundity". In: *Ecology* 81.4 (2000), pp. 1139–1142.

[36] J. Davis and M. Goadrich. "The relationship between Precision-Recall and ROC curves". In: *Proceedings of the 23rd international conference on Machine learning* (2006), pp. 233–240.

[37] J. Deutsch and D. He. "Using Deep Learning-Based Approach to Predict Remaining Useful Life of Rotating Components". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 48.1 (2017), pp. 1–10.

[38] P. M. Dixon, J. Weiner, T. Mitchell-Olds, and R. Woodley. "Bootstrapping the Gini coefficient of inequality". In: *Ecology* 68.5 (1987), pp. 1548–1551.

[39] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. 2nd ed. New York: Wiley, 2001. ISBN: 978-0471056690.

[40] A. Eskilson. *Apache Spark JIRA 18016*. 2016. URL: `https://issues.apache.org/jira/browse/SPARK-18016` (visited on 07/16/2018).

[41] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. "LIBLINEAR: A Library for Large Linear Classification". In: *Journal of Machine Learning Research* 9 (2008), pp. 1871–1874.

[42] U. Fayyad and K. Irani. "Multi-interval discretization of continuous-valued attributes for classification learning". In: *Thirteenth International Joint Conference on Artificial Intelligence* (1993), pp. 1022–1027.

[43] C. Ferri, J. Hernández-Orallo, and R. Modroiu. "An experimental comparison of performance measures for classification". In: *Pattern Recognition Letters* 30.1 (2009), pp. 27–38.

[44] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*. Vol. 1. Series in Statistics. New York, USA: Springer, 2001. ISBN: 978-0387848570.

[45] E. Frisk, M. Krysander, and E. Larsson. “Data-Driven Lead-Acid Battery Prognostics Using Random Survival Forests”. In: *Annual Conference Of The Prognostics and Health Management Society* (2014), pp. 92–101.

[46] E. Fuchs, T. Gruber, H. Pree, and B. Sick. “Temporal data mining using shape space representations of time series”. In: *Neurocomputing* 74.1 (2010), pp. 379–393.

[47] K. Goebel, B. Saha, and A. Saxena. “A comparison of three data-driven techniques for prognostics”. In: *62nd Meeting of the Society for Machinery Failure Prevention Technology* (2008), pp. 119–131.

[48] T. Golub, D. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J. Mesirov, H. Coller, M. Loh, J. Downing, and M. Caligiuri. “Molecular classification of cancer: class discovery and class prediction by gene expression monitoring”. In: *Science* 286.5439 (1999), pp. 531–537.

[49] R. Gouriveau, K. Medjaher, and N. Zerhouni. *From prognostics and health systems management to predictive maintenance 1: Monitoring and prognostics*. Reliability of multiphysical systems set. London: ISTE, Wiley, 2016. ISBN: 978-1848219373.

[50] B. Green and S. Seshadri. *AngularJS*. 1st ed. Sebastopol: O’Reilly & Associates, 2013. ISBN: 978-1449344856.

[51] M. Grover and T. Malaska. *Top 5 Mistakes when writing Spark applications*. URL: https://databricks.com/session/top-5-mistakes-when-writing-spark-applications (visited on 07/16/2018).

[52] H. Guo and H. L. Viktor. “Learning from imbalanced data sets with boosting and data generation: the databoost-im approach”. In: *ACM Sigkdd Explorations Newsletter* 6.1 (2004), pp. 30–39.

[53] I. Guyon. *Design of experiments of the NIPS 2003 variable selection benchmark*. 2003. URL: http://clopinet.com/isabelle/Projects/NIPS2003/Slides/NIPS2003-Datasets.pdf (visited on 07/16/2018).

[54] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. “Gene Selection for Cancer Classification using Support Vector Machines”. In: *IEEE Machine Learning* 46.1 (2002), pp. 389–422.

[55] H. Han, W.-Y. Wang, and B.-H. Mao. "Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning". In: *International Conference on Intelligent Computing* (2005), pp. 878–887.

[56] P. Hart. "The condensed nearest neighbor rule (Corresp.)" In: *IEEE transactions on information theory* 14.3 (1968), pp. 515–516.

[57] H. He and E. A. Garcia. "Learning from imbalanced data". In: *IEEE Transactions on Knowledge and Data Engineering* 21.9 (2009), pp. 1263–1284.

[58] H. He, Y. Bai, E. A. Garcia, and S. Li. "ADASYN: Adaptive synthetic sampling approach for imbalanced learning". In: *IEEE International Joint Conference on Neural Networks* (2008), pp. 1322–1328.

[59] F. O. Heimes. "Recurrent neural networks for remaining useful life estimation". In: *IEEE International Conference on Prognostics and Health Management* (2008), pp. 1–6.

[60] A. Heng, S. Zhang, A. C. Tan, and J. Mathew. "Rotating machinery prognostics: State of the art, challenges and opportunities". In: *Mechanical Systems and Signal Processing* 23.3 (2009), pp. 724–739.

[61] O. Hryniewicz and J. Karpinski. "Prediction of reliability: the pitfalls of using Pearson's correlation". In: *Eksploatacja i Niezawodność* 16.3 (2014), pp. 472–483.

[62] C. Huertas and R. Juárez-Ramírez. "Filter feature selection performance comparison in high-dimensional data: A theoretical and empirical analysis of most popular algorithms". In: *IEEE 17th International Conference on Information Fusion* (2014), pp. 1–8.

[63] N. Japkowicz, C. Myers, and M. Gluck. "A novelty detection approach to classification". In: *Proceedings of the international joint conference on Artificial intelligence* 1.14 (1995), pp. 518–523.

[64] N. Japkowicz and S. Stephen. "The class imbalance problem: A systematic study". In: *Intelligent Data Analysis* 6.5 (2002), pp. 429–449.

[65] M. V. Joshi, R. C. Agarwal, and V. Kumar. "Predicting rare classes: Can boosting make any weak learner strong?" In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining* (2002), pp. 297–306.

[66] S. Kamburugamuve, G. Fox, D. Leake, and J. Qiu. *Survey of apache big data stack*. 2013. URL: http://grids.ucs.indiana.edu/ptliupages/publications/survey_apache_big_data_stack.pdf (visited on 07/16/2018).

[67] M. Kaminski and B. Schlegel. *Feature Selection for Apache Spark*. 2017. URL: https://github.com/MarcKaminski/spark-FeatureSelection (visited on 06/17/2018).

[68] S. S. Khan and M. G. Madden. "A survey of recent trends in one class classification". In: *Irish Conference on Artificial Intelligence and Cognitive Science* (2009), pp. 188–197.

[69] N.-H. Kim, D. An, and J.-H. Choi. *Prognostics and Health Management of Engineering Systems*. Basel: Springer International Publishing, 2017. ISBN: 978-3319447407.

[70] K. Kira and L. A. Rendell. "The feature selection problem: Traditional methods and a new algorithm". In: *Machine Learning Proceedings* 2 (1992), pp. 129–134.

[71] I. Kononenko. "Estimating Attributes: Analysis and Extensions of RELIEF". In: *Lecture Notes in Computer Science* 784 (2005), pp. 171–182. (Visited on 07/06/2016).

[72] I. Kononenko and M. Kukar. *Machine learning and data mining: introduction to principles and algorithms*. Chichester: Horwood Publishing, 2007. ISBN: 978-1904275213.

[73] M. Kubat, R. C. Holte, and S. Matwin. "Machine learning for the detection of oil spills in satellite radar images". In: *Machine Learning* 30.2-3 (1998), pp. 195–215.

[74] M. Kubat and S. Matwin. "Addressing the curse of imbalanced training sets: one-sided selection". In: *International Conference on Machine Learning* 97 (1997), pp. 179–186.

[75] G. Lemaitre, F. Nogueira, and C. K. Aridas. "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning". In: *Journal of Machine Learning Research* 18.17 (2017), pp. 1–5.

[76] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata. "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median". In: *Journal of Experimental Social Psychology* 49.4 (2013), pp. 764–766.

[77] M. Lichman. *UCI Machine Learning Repository*. 2013. URL: https://archive.ics.uci.edu/ml/index.php (visited on 06/17/2018).

[78] Lightbend. *Play Framework Documentation*. 2018. URL: https://www.playframework.com/documentation/2.6.x/Home (visited on 07/16/2017).

[79] H. Liu and H. Motoda. *Computational methods of feature selection*. Boca Raton: CRC Press, 2007. ISBN: 978-1584888789.

[80] J. Liu, A. Saxena, K. Goebel, B. Saha, and W. Wang. *An Adaptive Recurrent Neural Network for Remaining Useful Life Prediction of Lithium-ion Batteries*. 2010. URL: http://www.dtic.mil/dtic/tr/fulltext/u2/a562707.pdf (visited on 07/16/2018).

[81] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008. ISBN: 978-0521865715.

[82] S. Marsland. *Machine learning: An algorithmic perspective*. Chapman & Hall/CRC machine learning & pattern recognition series. Boca Raton: CRC Press, 2009. ISBN: 978-1420067187.

[83] C. E. Metz. "Basic Principles of ROC Analysis". In: *Seminars in Nuclear Medicine* 8.4 (1978), pp. 283–298.

[84] I. Mitov, K. Ivanova, K. Markov, V. Velychko, P. Stanchev, and K. Vanhoof. "Comparison of discretization methods for preprocessing data for pyramidal growing network classification method". In: *New trends in intelligent technologies, sofia* (2009), pp. 31–39.

[85] A. Mosallam, K. Medjaher, and N. Zerhouni. "Data-driven prognostic method based on Bayesian approaches for direct remaining useful life prediction". In: *Journal of Intelligent Manufacturing* 27.5 (2016), pp. 1037–1048.

[86] T. C. Müller, O. Krieger, A. Breuer, K. Lange, and T. Form. "A Heuristic Approach for Offboard-Diagnostics in Advanced Automotive Systems". In: *SAE International Journal of Passenger Cars - Electronic and Electrical Systems* 2 (2009), pp. 344–351.

[87] K. P. Murphy. *Machine learning: a probabilistic perspective*. Cambridge: The MIT Press, 2012. ISBN: 978-0262018029.

[88] A. Ng. "Feature selection, L 1 vs. L 2 regularization, and rotational invariance". In: *Twenty-first international conference on Machine learning* (2004), p. 78.

[89] A. Nuhic, T. Terzimehic, T. Soczka-Guth, M. Buchholz, and K. Dietmayer. “Health diagnosis and remaining useful life prognostics of lithium-ion batteries using data-driven methods”. In: *Journal of Power Sources* 239 (2013), pp. 680–688.

[90] R. Pearson, G. Goney, and J. Shwaber. “Imbalanced clustering for microarray time-series”. In: *International Conference on Machine Learning* 3 (2003), pp. 1–8.

[91] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[92] K. M. Prabhu. *Window functions and their applications in signal processing*. Boca Raton: CRC Press, 2013. ISBN: 978-1138076136.

[93] R. Prytz, S. Nowaczyk, T. Rögnvaldsson, and S. Byttner. “Predicting the need for vehicle compressor repairs using maintenance records and logged vehicle data”. In: *Engineering Applications of Artificial Intelligence* 41 (2015), pp. 139–150.

[94] B. Raskutti and A. Kowalczyk. “Extreme re-balancing for SVMs: a case study”. In: *ACM Sigkdd Explorations Newsletter* 6.1 (2004), pp. 60–69.

[95] R. Razavi-Far, M. Farajzadeh-Zanjani, S. Chakrabarti, and M. Saif. “Data-driven prognostic techniques for estimation of the remaining useful life of lithium-ion batteries”. In: *IEEE International Conference on Prognostics and Health Management* (2016), pp. 1–8.

[96] K. Rieck. *Machine learning for application-layer intrusion detection*. 2009. URL: https://www.depositonce.tu-berlin.de/bitstream/11303/2496/2/Dokument_38.pdf (visited on 07/16/2018).

[97] J. Rissanen. “Modeling by shortest data description”. In: *Automatica* 14.5 (1978), pp. 465–471.

[98] M. Robnik-Šikonja and I. Kononenko. “An adaptation of Relief for attribute estimation in regression”. In: *Machine Learning: Proceedings of the Fourteenth International Conference* (1997), pp. 296–304.

[99] P. Romanski and L. Kotthoff. *Package FSelector*. 2016. URL: https://cran.r-project.org/web/packages/FSelector/FSelector.pdf (visited on 06/17/2018).

[100] Y. Saeys, I. Inza, and P. Larranaga. “A review of feature selection techniques in bioinformatics”. In: *Bioinformatics* 23.19 (2007), pp. 2507–2517.

[101] T. Saito and M. Rehmsmeier. “The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets”. In: *Public Library of Science ONE* 10.3 (2015), pp. 1–21.

[102] A. Saxena and K. Goebel. *Phm08 challenge data set*. 2008. URL: https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/ (visited on 07/16/2018).

[103] A. Saxena and K. Goebel. *Turbofan engine degradation simulation data set*. 2008. URL: https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/ (visited on 07/16/2018).

[104] A. Saxena, K. Goebel, D. Simon, and N. Eklund. “Damage propagation modeling for aircraft engine run-to-failure simulation”. In: *IEEE International Conference on Prognostics and Health Management* (2008), pp. 1–9.

[105] B. Schlegel and M. Kaminski. *Next Generation Workshop Car Diagnostics at BMW Powered by Apache Spark*. 2017. URL: https://www.youtube.com/watch?v=aVK-5QFmZDo& (visited on 06/17/2018).

[106] B. Schlegel, P. Wolf, and A. Mrowca. *RUL Estimation Code*. 2017. URL: https://github.com/BernhardSchlegel/rul-estimation (visited on 07/16/2018).

[107] B. Schlegel, A. Mrowca, P. Wolf, B. Sick, and S. Steinhorst. “Generalizing application agnostic remaining useful life estimation using data-driven open source algorithms”. In: *IEEE 3rd International Conference on Big Data Analysis* (2018), pp. 102–111.

[108] B. Schlegel and B. Sick. “Dealing with Class Imbalance the Scalable Way: Evaluation of Various Techniques Based on Classification Grade and Computational Complexity”. In: *IEEE International Conference on Data Mining Workshops* (2017), pp. 69–78.

[109] B. Schlegel and B. Sick. “Design and optimization of an autonomous feature selection pipeline for high dimensional, heterogeneous feature spaces”. In: *IEEE Symposium Series on Computational Intelligence* (2016), pp. 1–9.

[110] B. Schölkopf, R. C. Williamson, A. J. Smola, and Others. “Support vector method for novelty detection”. In: *Advances in neural information processing systems* 12 (1999), pp. 582–588.

[111] M. Schwabacher and K. Goebel. *A Survey of Artificial Intelligence for Prognostics*. URL: http://www.aaai.org/Papers/Symposia/Fall/2007/FS-07-02/FS07-02-016.pdf (visited on 07/06/2018).

[112] C. Seiffert, T. M. Khoshgoftaar, J. van Hulse, and A. Napolitano. “Resampling or reweighting: A comparison of boosting implementations”. In: *IEEE International Conference on Tools with Artificial Intelligence* 20 (2008), pp. 445–451.

[113] A. Sen. *On economic inequality*. Oxford University Press, 1973. ISBN: 978-0198281931.

[114] C. E. Shannon. “A Mathematical Theory of Communication”. In: *Bell System Technical Journal* 27 (1948), pp. 623–656.

[115] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. “The hadoop distributed file system”. In: *IEEE 26th symposium on Mass storage systems and technologies* (2010), pp. 1–10.

[116] X.-S. Si, W. Wang, C.-H. Hu, and D.-H. Zhou. “Remaining useful life estimation – A review on the statistical data driven approaches”. In: *European Journal of Operational Research* 213.1 (2011), pp. 1–14.

[117] J. P. Siebert. *Vehicle recognition using rule based methods*. Ed. by Turing Institute. 1987.

[118] Spark Developers. *Spark Release 2.0.0: Change log*. 2016. URL: https://spark.apache.org/releases/spark-release-2-0-0.html (visited on 06/17/2018).

[119] P. Struss. “Model-based problem solving”. In: *Foundations of Artificial Intelligence* 3 (2008), pp. 395–465.

[120] D. M. J. Tax and R. P. W. Duin. “Support vector data description”. In: *Machine Learning* 54.1 (2004), pp. 45–66.

[121] The German Association of the Automotive Industry. *Largest automobile markets worldwide between January and December 2016, based on new car registrations (in 1,000s)*. 2017. URL: https://www.statista.com/statistics/269872/largest-

automobile-markets-worldwide-based-on-new-car-registrations/ (visited on 07/16/2018).

[122] J.-H. Thomas and B. Dubuisson. "A Diagnostic Method using Wavelets Networks: Appication to Engine Knock Detection". In: *IEEE International Conference on Systems, Man, and Cybernetics* 1 (1996), pp. 244–249.

[123] Z. Tian. "An artificial neural network method for remaining useful life prediction of equipment subject to condition monitoring". In: *Journal of Intelligent Manufacturing* (2012), pp. 227–237.

[124] I. Tomek. "Two modifications of CNN". In: *IEEE Trans. Systems, Man and Cybernetics* 6 (1976), pp. 769–772.

[125] P. D. Turney. "Robust classification with context-sensitive features". In: *arXiv preprint cs/0212041* (2002), pp. 268–176.

[126] G. Vachtsevanos and P. Wang. "Fault prognosis using dynamic wavelet neural networks". In: *IEEE Systems Readiness Technology Conference AUTOTESTCON Proceedings* (2001), pp. 857–870.

[127] G. Vachtsevanos. *Intelligent fault diagnosis and prognosis for engineering systems*. Hoboken, Weinheim: Wiley, 2006. ISBN: 978-0471729990.

[128] T. van Tran, H. Thom Pham, B.-S. Yang, and T. Tien Nguyen. "Machine performance degradation assessment and remaining useful life prediction using proportional hazard model and support vector machine". In: *Mechanical Systems and Signal Processing* 32 (2012), pp. 320–330.

[129] Various. *Apache Apex*. 2018. URL: https://github.com/apache/apex-core (visited on 06/17/2018).

[130] Various. *Apache Flink*. 2018. URL: https://github.com/apache/flink (visited on 06/17/2018).

[131] Various. *Apache Heron*. 2018. URL: https://github.com/apache/incubator-heron (visited on 06/17/2018).

[132] Various. *Bootstrap Documentation*. 2018. URL: https://getbootstrap.com/docs/4.0/getting-started/introduction/ (visited on 07/16/2018).

[133] Various. *Onyx*. 2018. URL: https://github.com/onyx-platform/onyx (visited on 06/17/2018).

[134] Various. *Spark Versioning Policy*. 2018. URL: https://spark.apache.org/versioning-policy.html (visited on 06/17/2018).

[135] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, and Others. "Apache hadoop yarn: Yet another resource negotiator". In: *Proceedings of the annual Symposium on Cloud Computing* 4 (2014), p. 5.

[136] T. Wang, J. Yu, D. Siegel, and J. Lee. "A similarity-based prognostics approach for remaining useful life estimation of engineered systems". In: *IEEE International Conference on Prognostics and Health Management* (2008), pp. 1–6.

[137] S. Watanabe. "Feature compression". In: *Advances in information systems science* (1970), pp. 63–111.

[138] G. M. Weiss. "Mining with rarity: a unifying framework". In: *ACM Sigkdd Explorations Newsletter* 6.1 (2004), pp. 7–19.

[139] H. Wilkins, A. Sarb, and M. Semeniuj. *mleap*. 2018. URL: https://github.com/combust/mleap (visited on 07/16/2018).

[140] K. S. Woods, C. C. Doss, K. W. Bowyer, J. L. Solka, C. E. Priebe, and W. P. Kegelmeyer Jr. "Comparative evaluation of pattern recognition techniques for detection of microcalcifications in mammography". In: *International Journal of Pattern Recognition and Artificial Intelligence* 7.06 (1993), pp. 1417–1436.

[141] K. Xu, M. Xie, L. C. Tang, and S. L. Ho. "Application of neural networks in forecasting engine systems reliability". In: *Applied Soft Computing* 2.4 (2003), pp. 255–268.

[142] J. Yan, M. Koc, and J. Lee. "A prognostic algorithm for machine performance assessment and its application". In: *Production Planning & Control* 15.8 (2004), pp. 796–801.

[143] M. Zaharia and B. Chambers. *Spark: The Definitive Guide*. Sebastopol: O'Reilly Media, Inc, 2017. ISBN: 978-1491912201.

[144] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. "Apache Spark: A Unified Engine for Big Data Processing". In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[145] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster Computing with Working Sets". In: *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), p. 10.

[146] F. Zamora-Martínez, P. Romeu, P. Botella-Rocamora, and J. Pardo. "On-line learning of indoor temperature forecasting models towards energy efficiency". In: *Energy and Buildings* 83 (2014), pp. 162–172.

[147] Z. Zheng, X. Wu, and R. Srihari. "Feature selection for text categorization on imbalanced data". In: *ACM Sigkdd Explorations Newsletter* 6.1 (2004), pp. 80–89.

[148] M. Zieba, S. K. Tomczak, and J. M. Tomczak. "Ensemble boosted trees with synthetic features generation in application to bankruptcy prediction". In: *Expert Systems with Applications* 58 (2016), pp. 93–101.

ISBN 978-3-7376-0738-4