UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering

Featurebasierte Fehlervorhersage mittels Methoden des Machine Learnings

Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Informatik

vorgelegt von

Stefan Hermann Strüder

Erstgutachter: Prof. Dr. Jan Jürjens
Institut für Softwaretechnik

Zweitgutachter: Dr. Daniel Strüber
Chalmers University of Technology - Göteborg, Schweden (bis 02.2020)
Radboud-Universität - Nijmegen, Niederlande

Koblenz, im April 2020

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Kurzfassung

Softwarefehler sind ein großes Ärgernis in der Softwareentwicklung und können nicht nur zu Rufschädigungen, sondern auch zu erheblichen finanziellen Schäden für Unternehmen führen. Aus diesem Grund wurden im vergangenen Jahrzehnt zahlreiche Techniken zur Erkennung und Vorhersage von Fehlern entwickelt, welche zum großen Teil auf Methoden des Machine Learnings basieren. Diese Techniken zielen üblicherweise auf die Vorhersage von Fehlern auf Dateiebene ab. Seit einigen Jahren steigt jedoch die Popularität von featurebasierter Softwareentwicklung: ein Paradigma welches auf Funktionsinkremente eines Softwaresystems (Features) setzt und somit für eine breite Variabilität des Softwareproduktes sorgt. Eine gängige Implementationstechnik für Features basiert auf Annotationen mit Präprozessoranweisungen, wie `#IFDEF` und `#IFNDEF`, deren Code sich über mehrere Dateien des Quellcodes der Software verteilt („code scattering“). Ein Fehler in solchem Featurecode kann aufgrund dessen weitreichende Folgen für die Funktionalität der gesamten Software haben. Weist ein Teil des Featurecodes Fehler auf, so wird die gesamte Funktion des Features fehlerhaft und führt unter Umständen zum Ausfall der gesamten Funktionalität der Software (Features sind „cross-cutting“, d.h. dateiübergreifend). An dieses Problem knüpft diese Arbeit an. Es wird eine Vorhersagetechnik für fehlerhafte Features entwickelt, welche auf Methoden des Machine Learnings basiert. Die Auswertung von acht Klassifikatoren, welche jeweils auf einem individuellen Klassifikationsalgorithmus basieren, zeigt, dass mithilfe des für diese Arbeit erstellten featurebasierten Datensets, eine Genauigkeit von bis zu 84% für die Vorhersage von fehlerhaften oder fehlerfreien Features erreicht werden konnte. Es wird zudem gezeigt, wie der Aspekt der Featureorientierung im Rahmen der Erstellung des Datensets eingebunden wurde und welche Resultate im Vergleich zur herkömmlichen dateibasierten Methodik erzielt werden konnten. Dieser Vergleich zeigte, dass der zusätzliche Einbezug des Featureaspekts in die dateibasierte Fehlervorhersage keinen signifikanten Einfluss auf die Vorhersageergebnisse hat.

Abstract

Software errors are a major nuisance in software development and can lead not only to damage to reputation but also to considerable financial losses for companies. For this reason, numerous techniques for detecting and predicting defects have been developed over the past decade, which are largely based on machine learning methods. These techniques are usually aimed at predicting defects at the file level. However, in recent years the popularity of feature-based software development has been increasing: a paradigm that relies on functional increments of a software system (features) and thus ensures a wide variability of the software product. A common implementation technique for features is based on annotations with preprocessor instructions, such as `#IFDEF` and `#IFNDEF`, whose code is spread over several files of the software source code („code scattering“). A bug in such feature code can have far-reaching consequences for the functionality of the entire software. If a part of the feature code contains defects, the entire function of the feature becomes faulty and may lead to the failure of the entire functionality of the software (features are „cross-cutting“). This problem is the subject of this thesis. A prediction technique for defective features is developed, which is based on machine learning methods. The evaluation of eight classifiers, each based on an individual classification algorithm, shows that the feature-based dataset created for this thesis allows an accuracy of up to 84% for the prediction of defective or defect-free features. It is also shown how the feature oriented aspect was integrated into the creation of the dataset and what results were achieved compared to the traditional file-based methodology. This comparison showed

that the additional inclusion of the feature aspect in the file-based defect prediction does not have a significant impact on the prediction results.

Anmerkung

Diese Masterarbeit entstand in Teilen in Zusammenarbeit mit der Forschungsgruppe der Division of Software Engineering unter der Leitung von Thorsten Berger am Department of Computer Science and Engeneering der Chalmers Universität of Technology in Göteborg, Schweden.

Mein besonderer Dank gilt Thorsten Berger für die Ermöglichung und Finanzierung dieser Zusammenarbeit. Ebenfalls gilt mein Dank dem gesamten Team der Forschungsgruppe für die Unterstützung bei Problemen und Fragen zu meiner Arbeit. Ein weiterer Dank gilt Daniel Strüber für seine Initiative zur Ermöglichung der Zusammenarbeit und für seine Funktion als herausragender Betreuer dieser Masterarbeit.

Comment

This master thesis was partly written in cooperation with the research group of the Division of Software Engineering headed by Thorsten Berger at the Department of Computer Science and Engineering of Chalmers University of Technology in Gothenburg, Sweden.

My special thanks goes to Thorsten Berger for facilitating and financing this cooperation. I would also like to thank the entire team of the research group for their support in case of problems and questions concerning my work. Further thanks are due to Daniel Strüber for his initiative to facilitate the cooperation and for his function as an outstanding supervisor of this master thesis.

„Der wird kein Koch, sondern was Ordentliches!“
Für dich.
In Erinnerung.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung und Motivation

Softwarefehler stellen einen erheblichen Auslöser für finanzielle Schäden und Rufschädigungen von Unternehmen dar. Solche Fehler reichen von kleineren „Bugs“ bis hin zu schwerwiegenden Sicherheitslücken. Aus diesem Grund herrscht ein großes Interesse daran, einen Entwickler zu warnen, wenn er aktualisierten Softwarecode veröffentlicht, der möglicherweise einen oder mehrere Fehler beinhaltet.

Zu diesem Zweck haben Forscher und Softwareentwickler im vergangenen Jahrzehnt verschiedene Techniken zur Fehlererkennung und Fehlervorhersage entwickelt, die zu einem Großteil auf Methoden und Techniken des *Machine Learnings* basieren [6]. Diese verwenden in der Regel historische Daten von fehlerhaften und fehlerfreien Änderungen an Softwaresystemen in Kombination mit einer sorgfältig zusammengestellten Menge von *Attributen* (in der Regel Features genannt[1]), um einen gegebenen Klassifikator zu trainieren [3, 11]. Dieser kann dann anschließend verwendet werden, um eine akkurate Vorhersage zu erhalten, ob eine neu erfolgte Änderung an einer Software fehlerhaft oder frei von Fehlern ist.

Die Auswahl an Algorithmen für die Klassifikation ist groß. Studien zeigen, dass aus dem Pool von verfügbaren Algorithmen sowohl Entscheidungsbaum-basierte (zum Beispiel J48, CART oder Random Forest) als auch Bayessche Algorithmen (zum Beispiel Naïve Bayes (NB), Bernoulli-NB oder multinomieller NB) die meistgenutzten sind [32]. Alternative Algorithmen sind beispielsweise Regression, k-Nearest-Neighbors oder künstliche neuronale Netze [6]. Anzumerken ist allerdings, dass es keinen Konsens über den besten verfügbaren Algorithmus gibt, da jeder Algorithmus unterschiedliche Stärken und Schwächen für bestimmte Anwendungsfälle aufweist.

Das Ziel dieser Arbeit ist die Entwicklung einer solchen Vorhersagetechnik für Softwarefehler basierend auf Software-Features. Diese Features beschreiben Inkremente der Funktionalität eines Softwaresystems. Die auf diese Weise entwickelten Softwaresysteme heißen Software-Produktlinien und bestehen aus einer Menge von ähnlichen Softwareprodukten. Sie zeichnen sich dadurch aus, dass sie eine gemeinsame Menge von Features sowie eine gemeinsame Codebasis besitzen [36]. Durch das Vorhandensein verschiedener Features entlang der Softwareprodukte, kann eine breite Variabilität innerhalb einer Produktlinie erreicht werden. Eine detaillierte Einführung in den Themenkomplex von Software-Produktlinien und featurebasierter Softwareentwicklung kann in Abschnitt 2.1 gefunden werden. Bei der Entwicklung der Vorhersagetechnik wird die Implementation von Features mittels Präprozessor-Anweisungen,

[1] Um einem missverständlichen und doppeldeutigen Gebrauch des Feature-Begriffes vorzubeugen, wird für die hier verwendete Beschreibung der Charakteristika von Daten auch im weiteren Verlauf dieser Ausarbeitung der Begriff „Attribute“ verwendet.

wie `#IFDEF` und `#IFNDEF` (auch Präprozessor-Direktiven genannt) betrachtet. Dieser bisher in wissenschaftlichen Arbeiten nur einmal im Rahmen einer Fallstudie betrachtete Ansatz [24] ist aufgrund mehrerer Gründe chancenreich:

1. Wenn ein bestimmtes Feature in der Vergangenheit mehr oder weniger fehleranfällig war, so ist eine Änderung, die das Feature aktualisiert, wahrscheinlich ebenfalls mehr oder weniger fehleranfällig.
2. Features, die mehr oder weniger fehleranfällig scheinen, könnten besondere Eigenschaften haben, die im Rahmen der Fehlervorhersage verwendet werden können.
3. Code, der viel featuresspezifischen Code enthält (insbesondere die sogenannten Feature-Interaktionen), ist möglicherweise fehleranfälliger als sonstiger Code.

Ein initiales und einfaches Beispiel für einen Softwarefehler innerhalb eines Features ist in Listing 1.1 dargestellt. Es ist zu erkennen, dass innerhalb des Codes des Features `print_time` die `printf`-Anweisung einen Schreibfehler enthält, der dazu führt, dass der Featurecode nicht ausgeführt werden kann. Ferner kann fehlerhafter Featurecode dazu führen, dass die gesamte Funktionalität des Features und gegebenenfalls der gesamten Software beeinträchtigt oder verhindert wird. Ein reales Beispiel für einen Fehler aus dem Sourcecode des Softwareprojekts Vim ist in Abbildung 3.4 dargestellt.

```
1 int test() {
2     #IFDEF print_time
3     prntf("Current time: %s", time(&now));
4     #ENDIF
5
6     printf("Hello World!");
7     return 0;
8   }
```

Listing 1.1: Exemplarische Darstellung eines fehlerhaften Features

Das zuvor genannte Ziel der Arbeit setzt sich aus mehreren Teilzielen zusammen. Dazu zählen die Erstellung eines Datensets unter Einbezug des Feature-Aspekts (Kapitel 3). Dieses Datenset dient wiederum zum Training einer repräsentativen Auswahl an Klassifikatoren (Kapitel 4) mit anschließender vergleichender Evaluation (Kapitel 5) dieser. Zusätzlich wird das featurebasierte Datenset mit einem klassischen dateibasierten Datenset (Abschnitt 5.3) verglichen, dessen Entwicklung aus einer wissenschaftlichen Arbeit entnommen wurde [18]. Ein genauer Überblick über die Forschungsziele befindet sich im nächsten Abschnitt.

Sollte sich im Rahmen der Evaluation einer dieser Klassifikatoren als besonders effektiv erweisen, so würde diese Arbeit den Stand der Technik hinsichtlich der Fehlererkennung in Features vorantreiben und Organisationen erlauben, bessere Einblicke in die Fehleranfälligkeit von Änderungen in ihrer durch Variabilität geprägten Codebasis zu erhalten.

1.1 Forschungsziele und Forschungsfragen

Wie bereits in der Einleitung beschrieben, ist das übergeordnete Ziel dieser Arbeit die Entwicklung einer Vorhersagetechnik für Fehler in featurebasierter Software unter Zuhilfenahme von Methoden des Machine Learnings. Dazu ist vorgesehen, das Augenmerk hinsichtlich der Datengrundlage auf Commits des Versionierungssystems Git zu richten. Ein Commit beschreibt dabei die Freischaltung von Veränderungen an einer oder mehreren Dateien. Als Datenbasis für das Trainieren der Klassifikatoren dienen dann Commits, für die auf Grundlage

eines automatisierten Verfahrens aus der Literatur eine Klassifikation in „defekt“ und „fehlerfrei“verfügbar ist. Dies ermöglicht es, für ausstehende oder zukünftige Softwareversionen akkurate Vorhersagen zu treffen, ob diese Fehler beinhalten. So kann das Risiko der Konsequenzen von Softwarefehlern gesenkt werden.

Der Prozess der Entwicklung der Vorhersagetechnik ist in drei zu erreichende Forschungsziele eingeteilt. Jedem Forschungsziel werden Forschungsfragen zugeordnet, deren Aufklärung einen zusätzlichen Teil zur Erfüllung der Ziele beiträgt. Im Folgenden werden die Forschungsziele (RO – „research objective“) mit ihren zugehörigen Forschungsfragen (RQ – „research question“) vorgestellt. Die Beantwortung der Forschungsfragen erfolgt im weiteren Verlauf der Arbeit. Erkennbar sind die Antworten auf die Fragen an ihrer Einrahmung.

RO1: Erstellung eines Datensets zum Trainieren von relevanten Machine-Learning-Klassifikatoren

- ⇒ RQ1a: Welche Daten kommen für die Erstellung des Datensets in Frage?
- ⇒ RQ1b: Wie weit müssen die Daten vorverarbeitet werden, um sie für das Training nutzbar zu machen?

RO2: Identifikation und Training einer Auswahl von relevanten Machine-Learning-Klassifikatoren basierend auf dem Datenset

- ⇒ RQ2: Welche Machine-Learning-Klassifikatoren kommen für die gegebene Aufgabe in Frage?

RO3: Evaluation und Gegenüberstellung der Klassifikatoren sowie Vergleich zu modernen Vorhersagetechniken, die keine Features nutzen

- ⇒ RQ3a: Welche miteinander vergleichbaren Merkmale besitzen die Klassifikatoren?
- ⇒ RQ3b: Welche Metriken können für den Vergleich verwendet werden?
- ⇒ RQ3c: Wie lassen sich die Klassifikatoren mit weiteren Vorhersagetechniken, die keine Features nutzen, vergleichen?
- ⇒ RQ3d: Wie beeinflusst die Verwendung von featurebasierten Metriken die Fehlervorhersage?

Zusätzlich zu den drei genannten Forschungszielen umfasst die Bearbeitung der Masterarbeit eine Vor- und Nachbereitung, sodass sich insgesamt fünf Arbeitsphasen ergeben:

- Vorbereitung
- Abschluss des ersten Forschungsziels (*Erstellung des Datensets*)
- Abschluss des zweiten Forschungsziels (*Training von Machine-Learning-Klassifikatoren*)
- Abschluss des dritten Forschungsziels (*Evaluation und Vergleich*)
- Nachbereitung

Diese Arbeitsphasen werden im kommenden Abschnitt anhand des verwendeten Forschungsdesigns näher erläutert.

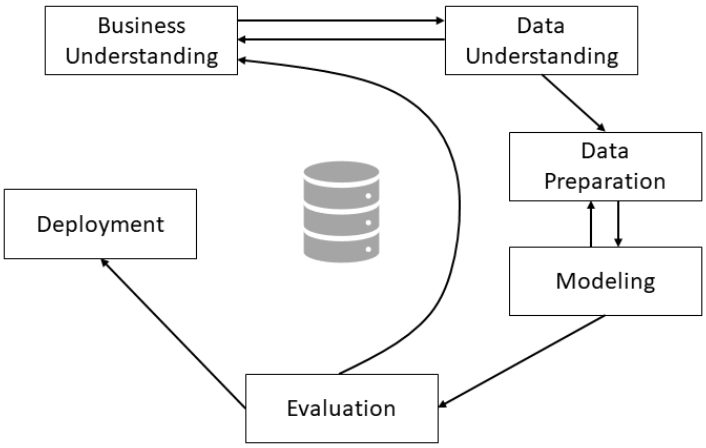


Abbildung 1.1: CRISP-DM Prozessmodell nach [7]

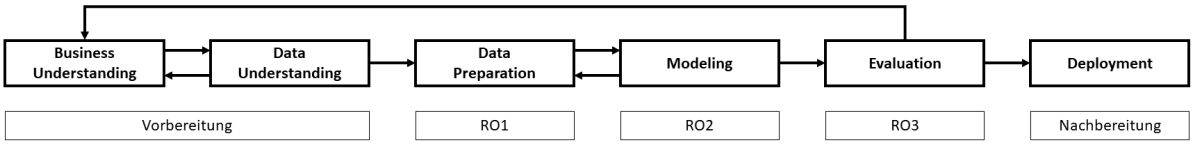


Abbildung 1.2: Phasen des CRISP-DM Prozessmodells nach [7] mit Zuordnung der Arbeitsphasen

1.2 Forschungsdesign

Die für diese Arbeit gewählte Methodik basiert auf dem Prozessmodell „Cross-Industry Standard Process for Data Mining“, kurz CRISP-DM, nach Chapman et al. [7]. Es wird als Vorlage für die Arbeitsphasen zur Erreichung der Forschungsziele dieser Arbeit verwendet. Da sich der überwiegende praktische Teil dieser Arbeit auf Programmierung im Bereich des Machine Learning konzentriert, bildet das CRISP-DM Prozessmodell ein passendes vordefiniertes Vorgehen. Eine grafische Aufarbeitung des Prozessmodells mit seinen sechs zugehörigen Phasen sowie den Verbindungen zwischen ihnen ist in Abbildung 1.1 dargestellt.

Das CRISP-DM-Prozessmodell wurde, wie der ausgeschriebene Name bereits andeutet, ursprünglich für die Erarbeitung von Data-Mining-Projekten entwickelt, eignet sich jedoch auch zur Verwendung im Rahmen eines Machine Learning Projektes, da sich die in beiden Bereichen verwendeten Methoden und Prozesse zu einem erheblichen Teil überlagern. Ein Überblick über die sechs Phasen des Prozessmodells ist in Abbildung 1.2 dargestellt. Zusätzlich umfasst diese Abbildung die Zuordnung der Arbeitsphasen, die im vorherigen Abschnitt definiert wurden. Eine Erläuterung der Phasen des Prozessmodells erfolgt im Anschluss.

Die ersten beiden Phasen *Business Understanding* und *Data Understanding* widmen sich der Vorbereitung der Arbeit. Die initiale Phase umfasst dabei die allgemeine Einarbeitung in das zugrundeliegende Thema und der Formulierung der Forschungsziele. Die darauffolgende Phase *Data Understanding* dient der Suche und Einsicht von für den weiteren Verlauf der Phasen relevanten Daten und, falls vorhanden, vorgefertigten Datensets (*Anmerkung:* Es konnten keine passenden Datensets gefunden werden). Da Commits als Datenbasis zum Training der Klassifikatoren betrachtet werden, wird der überwiegende Teil der Suche nach Daten in Software-Repositories stattfinden, welche dem Versionierungssystem Git zugrunde liegen. Für die weiteren Phasen ist es von besonderer Bedeutung, den Aufbau der Daten sorgfältig zu untersuchen. Die dritte Phase *Data Preparation* kümmert sich um die Erstellung des featurebasierten Datensets und den dort hinführenden Prozessen. Diese Phase ist deckungsgleich mit

den Anforderungen des ersten Forschungsziels. Zur Anwendung kommt das im vorherigen Schritt erstellte Datenset in der Phase *Modeling*. In dieser werden die auf Machine-Learning-Algorithmen basierenden Klassifikatoren mithilfe des Datensets trainiert und anschließend getestet, um Anpassungen an den Algorithmen hinsichtlich einer höheren Genauigkeit der Vorhersagen vornehmen zu können. Diese Phase spiegelt somit die Erfüllung des zweiten Forschungsziels wider. Die fünfte Phase umfasst die *Evaluation* der Resultate des zuvor erfolgten Schrittes und deckt somit die Erfüllung des dritten Forschungsziels ab. Die Nachbereitung der Arbeit wird durch die Phase *Deployment* abgedeckt. Diese umfasst die Erstellung bzw. Finalisierung der schriftlichen Ausarbeitung sowie das Erstellen der Abschlusspräsentation und der anschließenden Vorführung dieser im Rahmen des Kolloquiums. Ferner sind in Abbildung 1.2 Rückpfeile zwischen einzelnen Phasen angegeben. So können beispielsweise Erkenntnisse im Rahmen der Phase Data Understanding zu offenen Fragestellungen führen, die die Phase des Business Understanding betreffen. Gleiches gilt für die Phase Modeling mit einem Rückpfeil zur Phase Data Preparation. Weiterhin können Erkenntnisse, die im Rahmen der Evaluationsphase gewonnen werden können, zuvor unbekanntes Wissen aus der ersten Phase erläutern.

1.3 Aufbau der Arbeit

Diese Ausarbeitung ist in sechs Kapitel unterteilt. Kapitel 1, welches mit diesem Abschnitt abgeschlossen wird, diente zur Einführung in das Thema der Masterarbeit. Ebenso stellte es die theoretischen Rahmenbedingungen der Arbeit vor. Kapitel 2 dient zur Vermittlung von Basiswissen zu den grundlegenden Themenkomplexen dieser Ausarbeitung. Dazu wird zunächst die featurebasierte Softwareentwicklung vorgestellt, ehe dann die Machine-Learning-Klassifikation sowie die darauf aufbauende Fehlervorhersage erläutert werden. Kapitel 4 und Kapitel 5 widmen sich der Auseinandersetzung des praktischen Teils dieser Masterarbeit in Form der Erstellung des featurebasierten Datensets sowie des Trainings der Machine-Learning-Klassifikatoren. Die Gegenüberstellung und Evaluation dieser Klassifikatoren erfolgt in Kapitel 5 inklusive eines Vergleiches zu einer nicht-featurebasierten klassischen Methode zur Fehlererkennung. Eine abschließende Zusammenfassung sowie ein Ausblick auf weiterführende Projekte, die auf diese Masterarbeit aufbauen können, erfolgen in Kapitel 6.

Zusätzlich wird die Ausarbeitung von zahlreichen Abbildungen und Tabellen zur verständlichen Verdeutlichung von Zusammenhängen ergänzt.

Kapitel 2

Hintergrund

Dieses Kapitel dient zur Einführung in die dieser Arbeit zugrundeliegenden Themen und hat das Ziel, Basiswissen für den weiteren Verlauf der Ausarbeitung aufzubauen. Dazu wird zunächst die featurebasierte Softwareentwicklung erläutert, ehe dann der Themenbereich des Machine Learnings vorgestellt wird. Dazu werden die Klassifikation und die Fehlervorhersage mittels Machine Learning erläutert. Unterstützt werden die Abschnitte von Grafiken zum besseren Verständnis der Zusammenhänge.

2.1 Featurebasierte Softwareentwicklung

Das zentrale Konzept hinter der featurebasierten Softwareentwicklung stellen sogenannte Software-Produktlinien dar. Wie bereits in der Einleitung erwähnt wurde, beschreiben Software-Produktlinien eine Menge von ähnlichen Softwareprodukten, welche eine gemeinsame Menge von Features sowie eine gemeinsame Codebasis besitzen und sich durch die Auswahl der verwendeten Features unterscheiden, sodass eine breite Variabilität innerhalb einer Produktlinie entstehen kann [4, 36].

Der zentrale Prozess der Generierung einer Software-Produktlinie ist in Abbildung 2.1 dargestellt. Aufgeteilt wird dieser Prozess in das Domain Engineering und das Application Engineering. Im Rahmen des Domain Engineerings wird ein sogenanntes Variabilitätsmodell (Variability Model) erstellt, welches die wählbaren Features und Constraints für mögliche Selektionen beschreibt [4]. Gängige Implementationstechniken für Features reichen von einfachen Lösungen durch Annotationen basierend auf Laufzeitparametern oder Präprozessor-Anweisungen bis hin zu verfeinerten Lösungen basierend auf erweiterten Programmiermethoden, wie zum Beispiel Aspektorientierung. In einigen dieser Implementierungstechniken wird jedes Feature als wiederverwendbares Domain Artifact modelliert und gekapselt, welches im Prozess des Application Engineerings in Form einer Konfiguration zusammen mit weiteren Features, im Hinblick auf die gewünschte Funktionalität der Software, ausgewählt werden kann. Ein Software Generator erzeugt dann die gewünschten Softwareprodukte basierend auf den bereits zuvor genannten Implementationstechniken für Features.

Die in dieser Arbeit betrachtete Implementierungstechnik von Features basiert auf Anweisungen beziehungsweise Bedingungsdirektiven des C-Präprozessors. Die für diese Arbeit relevanten Direktiven lauten `#IFDEF` und `#IFNDEF`. Einfache Beispieleinsätze für beide Direktiven sind in Listing 2.1 und Listing 2.2 zu sehen. Sie wurden jeweils aus der wissenschaftlichen Literatur entnommen [17, 23]. Die Direktive `#IFDEF` leitet in Listing 2.1 den Code des Features

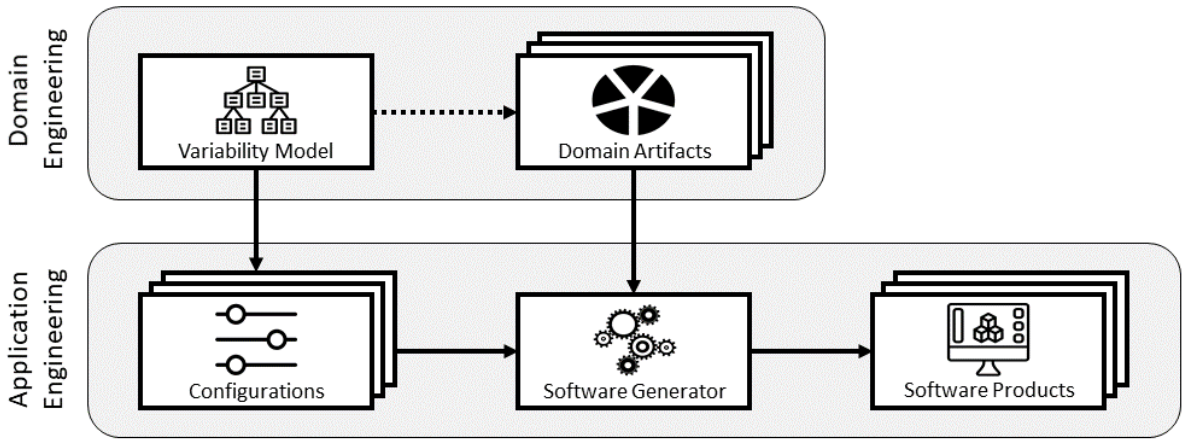


Abbildung 2.1: Generierung von Software-Produktlinien nach [36]

__unix__ ein, welcher mit der Anweisung #ENDIF endet. Der in den Zeilen 2 bis 6 angegebene Codeteil wird genau dann nur ausgeführt, wenn das Feature __unix__ im Rahmen der Konfiguration des Softwareproduktes definiert beziehungsweise aktiviert ist [35]. In diesem Fall wird die Bedingung der Direktive erfolgreich erfüllt [35]. Sie schlägt fehl, wenn das Feature nicht definiert beziehungsweise aktiviert ist [35]. Die Direktive #IFNDEF wird für Code verwendet, der ausgeführt werden soll, wenn ein Feature nicht definiert ist. Im Falle des Beispiels in Listing 2.2 wird der in Zeile 3 angedeutete Code nur ausgeführt, wenn NO_XMALLOC nicht aktiviert wurde. Es besteht zudem die Möglichkeit, Features bzw. ihren Code zu verschachteln. Ein Beispiel dafür ist in Listing 2.3 angegeben. Es ist zu erkennen, dass sich der Code von FEAT_MZSCHEME innerhalb der bedingten Gruppe von USE_XSMP befindet. Der in Zeile 5 angedeutete Code kann somit nur ausgeführt werden, wenn USE_XSMP aktiviert ist. Im Fall von Verschachtelung beendet ein #ENDIF immer das nächstgelegene #IFDEF oder #IFNDEF [35]. Es besteht zudem die Möglichkeit, Direktiven mittels „und“ (&&, and) oder „oder“ (||, or) zu erweiterten Bedingungen zu verknüpfen, die zudem Negation in Form des !-Operators (anstelle von #IFNDEF) enthalten können [35, 25]. Dargestellt ist dies in Listing 2.4.

```
1 #IFDEF __unix__
2   #include "directorySelection.h"
3   #include "directoryNames.h"
4   void getDirectoryName(char* dirname
      ) {
5     getHomeDirectory(dirname);
6   }
7 #ENDIF
```

Listing 2.1: Beispieleinsatz von #IFDEF nach [23]

```
1 int test = 1;
2 #IFNDEF NO_XMALLOC
3   test = memory != NULL;
4 #ENDIF
5 if (test){
6  // Lines of code here..
7  }
```

Listing 2.2: Beispieleinsatz von #IFNDEF nach [17]

```
bool time = msec > 0;
#IFDEF USE_XSMP
  time = time && xsmp_icefd != -1;
  #IFDEF FEAT_MZSCHEME
    time = time || p_mzq > 0;
  #ENDIF
#ENDIF
if (time)
 gettime(&start_tv);
```

Listing 2.3: Beispiel eines verschachtelten Einsatzes von `#IFDEF` nach [17]

```
#IFDEF FEATURE_A && FEATURE_B
  (...)
#ENDIF
(...)
#IFDEF !FEATURE_A && FEATURE_C
  (...)
#ENDIF
```

Listing 2.4: Beispiele von erweiterten Bedingungen nach [25]

Die in den Listings gezeigten Beispiele zeigen jeweils nur den Featurecode in einer Methode beziehungsweise in einer Datei. Fragmente des Featurecodes erstrecken sich jedoch nicht nur möglicherweise mehrfach über eine Datei sondern über mehrere Dateien - der Featurecode ist somit verstreut (englisch: code scattering), um eine Funktionalität des Features in der Gesamtheit der Software zu ermöglichen. Ein Defekt innerhalb eines Fragmentes des Featurecodes kann allerdings dazu führen, dass die gesamte Funktionalität des Features beeinträchtigt oder unterbunden wird, da der Fehler übergreifend wirkt (englisch: cross-cutting). Ebenfalls kann ein solcher Fehler dazu führen, dass die Funktionalität des gesamten Sourcecodes beeinträchtigt wird.

2.2 Machine-Learning-Klassifikation

Das Themengebiet des Machine Learnings (ML) ist in zwei Teilgebiete unterteilt - das unüberwachte ML (englisch: unsupervised ML) und das überwachte ML (englisch: supervised ML). Die Methoden in diesen Teilgebieten verfolgen unterschiedliche Ziele. Im Rahmen des unüberwachten ML werden Prozesse durchgeführt, welche dazu dienen, die Struktur einer unbekannten Eingabemenge an Daten zu erlernen und anschließend zu repräsentieren [30]. Eine gängige Anwendung des unüberwachten ML ist das Clustering. Das überwachte ML beschreibt wiederum einen Prozess, welcher beabsichtigt, Vorhersagen über unbekannte Eingabedaten auf Basis des Trainings einer Abbildungsfunktion zu treffen [30]. Die Attribute „unüberwacht“ und „überwacht“ erhalten die Methoden aufgrund ihrer Art des Lernens beziehungsweise des Trainings. In der Anwendung des unüberwachten ML werden die Eingabedaten erfasst, gegebenenfalls vorverarbeitet, um dann auf deren Basis ein Modell zu erlernen, welches die Darstellung beziehungsweise Repräsentation der Eingabedaten bestimmt [2]. Auf der anderen Seite wird unter Anwendung des überwachten ML, ein Modell auf Basis eines sogenannten „gelabelten“ (beschrifteten) Datensatzes durch Merkmalsextraktion in Form von Attributen und dem Training auf der Grundlage der extrahierten Merkmale erstellt [2]. Der Datensatz, welcher zum Training verwendet wird, wird im gängigen Sprachgebrauch des Machine Learning Datenset (englisch: dataset) genannt. Das aus dem Training resultierende Modell wird Klassifikator (englisch: classifier) genannt. Gängige Anwendungen des überwachten ML sind Regression und Klassifikation. In dieser Arbeit kommt die Klassifikation als Anwendung des überwachten ML zum Einsatz. Der grundlegende Prozess der Machine-Learning-Klassifikation ist in Abbildung 2.2 anhand eines Beispiels dargestellt.

Die Abbildung zeigt den Prozess des überwachten Machine Learnings anhand des Beispiels des Trainings eines Klassifikators zur Erkennung beziehungsweise Vorhersage von geometrischen Formen. Der Prozess beginnt mit den „gelabelten“ Eingabedaten (A). Die Werte der Label (kategorial oder numerisch) stellen dabei die zu vorhersagende Zielklasse dar. In diesem

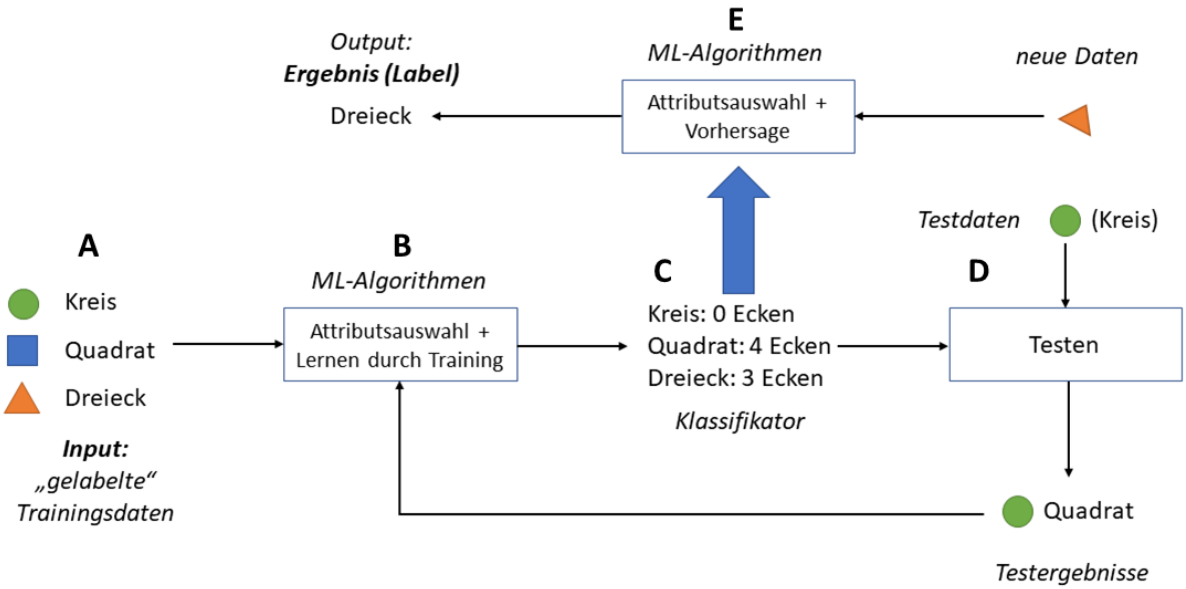


Abbildung 2.2: Allgemeiner Prozess des überwachten Machine Learnings dargestellt anhand eines Beispiels (vereinfacht)

Falle bilden die Namen der geometrischen Formen die Label als kategorischen Wert. Die Rohdaten der Eingabemenge bestehen aus den geometrischen Formen selbst. Beide Datenmengen bilden das Datenset. Um nun einen Klassifikator trainieren zu können, müssen Merkmale der Eingangsdaten ausgewählt werden, anhand derer diese identifiziert werden können (B). Diese zu identifizierenden Charakteristika der Daten werden Attribute genannt. Diese Attribute können bereits vor dem Training festgelegt werden oder automatisiert extrahiert werden. Im vorliegenden Fall wurde die Metrik „Anzahl der Ecken der geometrischen Formen“ als Attribut zum Training ausgewählt. Das Ergebnis ist der fertig trainierte Klassifikator, welcher das antrainierte Wissen auf neue Daten abbilden kann (C). Ein Teil des Datensets wird in der Regel verwendet, um den Klassifikator nach dessen Erstellung zu testen (D). Die in der Regel verwendeten Verhältnisse (englisch: Split-Ratio) zwischen Trainings- und Testdaten betragen 80:20 (basierend auf dem Paretoprinzip) oder 75:25 (zum Beispiel [24]). Diese Testdaten werden dem Klassifikator als Eingabemenge zur Klassifikation ohne Label zur Verfügung gestellt. Die Label sollten jedoch nicht verworfen werden, da sie als Vergleichsgrundlage für die Vorhersageperformanz des Klassifikators dienen. Sie bilden die sogenannte „Ground Truth“ (deutsch: Grundwahrheit). Dazu werden die vom Klassifikator vorhergesagten Label mit denen der Ground Truth verglichen. Sollte dieser Vergleich ergeben, dass die Label große Abweichungen zeigen, so kann der Klassifikator erneut mit anderen Attributen oder einer veränderten Split-Ratio trainiert werden. Erfüllt der Klassifikator die Anforderungen an die Performanz der Vorhersagen, so ist dieser bereit Vorhersagen auf Basis neuer Eingabedaten zu treffen (E). Dazu müssen von den neuen Daten die Attribute ermittelt werden. Auf Basis dieser trifft der Klassifikator die Vorhersage und liefert als Ausgabe das Label des Wertes der Zielklasse. Im Voraus des Testens mit den Testdaten (D) werden in manchen Fällen zudem sogenannte Validierungsdaten verwendet. Dabei handelt es sich um eine eigenständige Teilmenge der Trainingsdaten, welche verwendet wird, um die Klassifikatoren nach jedem Training zu evaluieren, um die Auswahl der Attribute hinsichtlich der Performanz auf Eignung zu prüfen [30]. Die Anwendung der Testdaten erfolgt dann im Anschluss.

Der in Abbildung 2.2 dargestellte Klassifikator stellt einen multinomiellen oder multi-class Klassifikator dar, da er zu drei oder mehr Werten der Zielklasse zuordnen kann [30]. Für viele praktische Anwendungen genügt jedoch ein binärer Klassifikator, welcher Vorhersagen zu

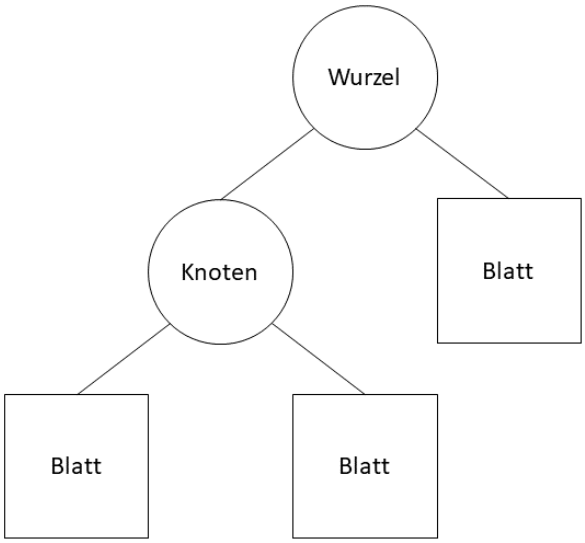


Abbildung 2.3: Grundsätzlicher Aufbau eines Decision Trees

zwei Werten der Zielklasse trifft. Dies trifft auch auf die Klassifikatoren dieser Arbeit zu.

Nachfolgend wird eine Auswahl an Klassifikationsalgorithmen vorgestellt. Diese zählen zu den meist verwendeten Algorithmen und finden auch in dieser Arbeit ihre Anwendung.

Decision Trees

Decision Trees (deutsch: Entscheidungsbäume) zählen zu den meistverwendeten Klassifikatoren im Bereich des supervised Machine Learnings. Studien belegen, dass sie hinsichtlich der Verwendung im Kontext der Fehlererkennung die häufigste Anwendung finden [32]. Decision Trees sind gerichtete und verwurzelte Bäume, die als rekursive Partition der Eingabemenge des Datensets aufgebaut werden [29]. Den Ursprung des Baumes bildet die Wurzel, welche keine eingehenden Kanten besitzt - alle weiteren Knoten besitzen jedoch eine eingehende Kante [29]. Diese Knoten teilen wiederum die Eingabemenge anhand einer vorgegebenen Funktion in zwei oder mehr Unterräume der Menge auf [29]. Meist geschieht dies anhand eines Attributs, sodass die Eingabemenge anhand der Werte des einzelnen Attributs geteilt wird [29]. Die Blätter des Baumes bilden die Zielklassen ab. Eine Klassifizierung kann folglich durchgeführt werden, indem man von der Wurzel bis zu einem Blatt den Kanten anhand der entsprechenden Werte der Eingangsmenge folgt. Es existieren verschiedene Algorithmen zur Erstellung von Decision Trees. Bekannte Stellvertreter dieser sind ID3, C4.5 (J48) und CART [29]. Der grundlegende Aufbau eines Decision Trees ist in Abbildung 2.3 dargestellt.

Eine Besonderheit von Decision Trees stellen sogenannte Random Forests dar. Diese beschreiben eine Menge von Klassifikatoren, bei der mehrere einzelne Decision Trees gleichzeitig erzeugt werden und deren Ergebnisse anschließen aggregiert werden [1]. Dazu erhält jeder Decision Tree eine Teilmenge der Eingabemenge des Datensets [1]. Random Forests eigenen sich besonders zur Anwendung, wenn viele Attribute im Datenset vorhanden sind [1].

k-Nearest-Neighbors

Ein k-Nearest-Neighbor-Klassifikator (deutsch: k-nächste-Nachbarn) basiert auf zwei Konzepten [40]. Das erste Konzept basiert auf der Abstandsmessung zwischen den Werten der zu klassifizierenden Datenmenge und den Werten der Attribute des Datensets [40]. Die Abstandmessung erfolgt in der Regel durch die Berechnung der Euklidischen Distanz $D(p, q)$:

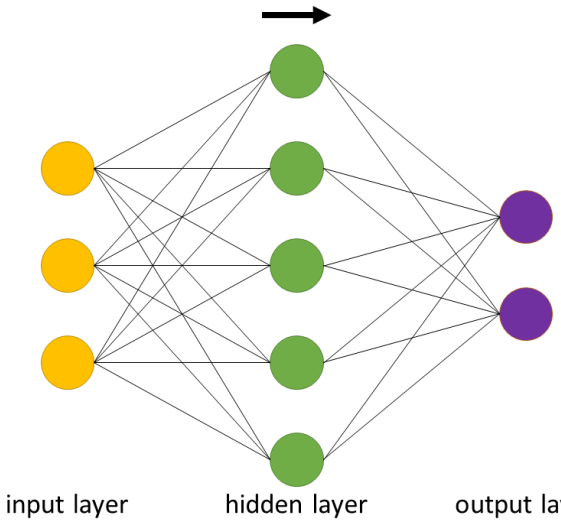


Abbildung 2.4: Grundsätzlicher Aufbau eines KNN mit drei Input-Layer-Neuronen, fünf Hidden-Layer-Neuronen und zwei Output-Layer-Neuronen

$$D(p,q) = \sqrt{\sum_{1}^{n}(p_n - q_n)^2}$$

Die Anzahl der Attribute wird durch den Parameter n wiedergegeben, p und q repräsentieren jeweils die Werte der zu klassifizierenden Datenmenge und die Werte der Attribute des Datensets. Das zweite Konzept bildet der Parameter k, der angibt, wie viele nächste Nachbarn zum Vergleich der zuvor berechneten Abstände in Betracht gezogen werden [40]. Bei einem $k > 1$ wird diejenige Zielklasse gewählt, deren Auftreten innerhalb der nächsten Nachbarn überwiegt.

Künstliche neuronale Netze

Künstliche neuronale Netze (englisch: Artificial Neural Networks) verwenden nicht-lineare Funktionen zur schrittweisen Erzeugung von Beziehungen zwischen der Eingabemenge und den Zielklassen durch einen Lernprozess [15]. Sie sind angelehnt an die Funktionsweise von biologischen Nervensystemen und bestehen aus einer Vielzahl von einander verbundenen Berechnungsknoten, den Neuronen [20]. Der grundsätzliche Aufbau eines künstlichen neuronalen Netzes kann in Abbildung 2.4 eingesehen werden. Der Lernprozess besteht aus zwei Phasen - einer Trainingphase und einer Recall-Phase [15]. In der Trainingsphase werden die Eingabedaten, meist als multidimensionaler Vektor, in den Input-Layer geladen und anschließend an die Hidden-Layer verteilt [20]. In den Hidden-Layers werden dann Entscheidungen anhand der Beziehungen zwischen den Eingabedaten und Zielklassen sowie die den Verbindungen zuvor zugewiesenen Gewichtsfaktoren getroffen [15, 20]. Im Rahmen der Recall-Phase wird die Vorhersage basierend auf der zu klassifizierenden Datenmenge anhand der zuvor getroffenen Entscheidungen der Hidden-Layers getroffen und an die jeweiligen Output-Layer, welche die Werte der Zielklasse repräsentieren, weitergeleitet[15].

Logistische Regression

Logistische-Regressions-Klassifikatoren (englisch: Logistic Regression) basieren auf dem mathematischen Konzept des Logits, welcher den natürlichen Logarithmus eines Chancenverhältnisses beschreibt [22]. Seine Formel lautet:

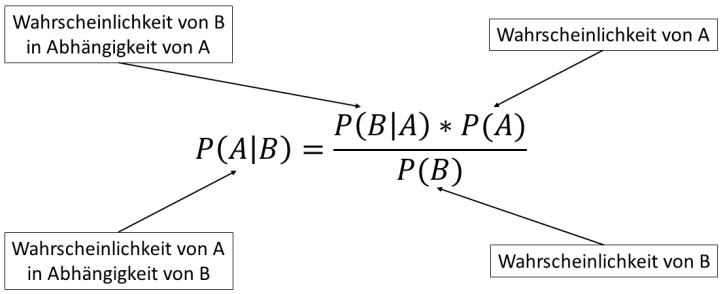


Abbildung 2.5: Satz von Bayes als Grundlage des Naïve-Bayes-Klassifikators

$$logit(Y) = ln(\frac{\pi}{1-\pi})$$

Y beschreibt dabei die zu klassifizierende Datenmenge, wohingegen π die Verhältnisse der Wahrscheinlichkeiten der Werte der Attribute der Eingabemenge bezeichnet. Am besten geeignet ist dieser Klassifikator für eine Kombination aus kategorialen oder numerischen Eingabedaten und kategorischen Zielklassen [22].

Naïve Bayes

Naïve-Bayes-Klassifikatoren zählen zu den linearen Klassifikatoren und basieren auf dem Satz von Bayes. Die Bezeichnung „naiv“ erhält der Klassifikator durch die Annahme, dass die Attribute der Eingabemenge unabhängig voneinander sind [27]. Diese Annahme wird zwar in der realen Verwendung des Klassifikators häufig verletzt, dennoch erzielt er in der Regel eine hohe Performanz [27]. Der Klassifikator gilt als effizient, robust, schnell und einfach implementierbar [27]. Die zur Durchführung einer Klassifikation mittels Naïve Bayes benötigte Formel nach Thomas Bayes ist in Abbildung 2.5 samt Erläuterung der einzelnen Faktoren aufgeführt.

Es existiert zudem eine Mehrzahl an Varianten des Naïve-Bayes-Klassifikators, die verschiedene Annahmen über die Verteilung der Attribute der Eingabemenge machen. Beispiele dafür sind der Gaußsche-Naïve-Bayes (normalverteilte Attribute), der multinomiale Naïve-Bayes (multinomiale Verteilung der Attribute) sowie der Bernoulli-Naïve-Bayes (unabhängige binäre Attribute).

Stochastic Gradient Descent

Ein Stochastic-Gradient-Descent-Klassifikator basiert auf dem Gradientenverfahren (englisch: Gradient Descent), welches das Ziel hat, zu einem gegebenen x das minimale y zu finden [34]. Im Falle der Klassifikation mittels Machine Learning bezeichnet x dabei die zu klassifizierende Eingabemenge und y das erwartete Ergebnis der Klassifikation. Minimiert werden sollen dabei die „Kosten“, die sich aus der Ermittlung der Ergebnisse ergeben.

Support Vector Machines

Support Vector Machines verfolgen das Ziel, eine sogenannte „Hyperplane“ in einem n-dimensionalen Raum (n = Anzahl der Attribute der Eingabemenge) zu finden, welche die Datenpunkte der Eingabemenge eindeutig klassifizieren kann [10]. Die Hyperplane beschreibt eine Trennlinie beziehungsweise Trennfläche, mit deren Hilfe die Daten der zu klassifizierenden Menge den Zielklassen zuordnen lassen [16]. Dabei gilt es, dass die Trennflächen, welche die Eingangsmenge anhand der Attribute in verschiedene Trennungsebenen unterteilen, einen möglichst großen Abstand ohne Datenpunkte voneinander haben [16]. Dies funktioniert sowohl für linear als auch nicht-lineare trennbare Mengen.

2.3 Fehlervorhersage mittels Machine Learning

Der Hintergrund der Fehlervorhersage mittels Machine Learning basiert auf dem zuvor vorgestellten Konzept des überwachten Machine Learnings. Als Grundlage dienen dabei meist Daten, die aus Software-Repositories entnommen beziehungsweise extrahiert werden. Viele Studien und wissenschaftliche Arbeiten setzen jedoch auch auf vorgefertigte Datensets, wie zum Beispiel von der NASA oder von Eclipse [32]. Die zugehörigen Label der Datensets lauten in der Regel „fehlerfrei“ und „defekt“ und können auf verschiedene Weisen ermittelt werden. Eine gängige Methode ist die Identifizierung von korrektiven und fehlereinführenden Commits als Entscheidungsgrundlage für das Label. Die üblichen Vorgehensweisen setzen auf die Einbindung von Bugtracking-Systemen oder die Analyse von Commit-Nachrichten zur Identifizierung der korrektiven Commits [24, 41]. Fehlereinführende Commits können anschließend unter Verwendung von Git-Kommandos oder durch die Anwendung des sogenannten SZZ-Algorithmus ermittelt werden. Dieser Algorithmus wird in dieser Arbeit verwendet und in Abschnitt 3.2 erläutert. Auf Basis dieser Daten werden die Attribute bestimmt. Dabei handelt sich sich in der Regel um Metriken, die entweder die Charakteristika des Sourcecodes (Codemetriken) oder Aktivitäten und Prozesse im Bezug auf Software-Repositories (Prozessmetriken) beschreiben [32, 26]. Mithilfe dieser Attribute werden die Klassifikatoren trainiert. Eine Studie, welche 156 wissenschaftliche Arbeiten zum Thema der Fehlervorhersage mittels Machine Learning analysierte, ergab, dass besonders Entscheidungsbaum-basierte, Bayessche Verfahren, Regression und künstliche neuronale Netze als Klassifikationsalgorithmen zur Anwendung kommen [32]. Diese Algorithmen wurden unter anderem auch in dieser Arbeit verwendet. Erläuterungen können in Abschnitt 4.1 gefunden werden. Die fertig trainierten Klassifikatoren können dann auf Basis neuer Daten Vorhersagen zum Zustand einer Software treffen. Die Vorhersagen beruhen in der Regel auf defekten Dateien im Kontext von Commits oder Releases.

Als Beispiel für zwei konkrete Anwendungen von Machine Learning gestützter Fehlervorhersage werden im Folgenden zwei wissenschaftliche Arbeiten vorgestellt. Die erste Arbeit „Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction“ von Moser et al. [18] stellt eine Methodik zur dateibasierten Fehlervorhersage vor. Die zweite Arbeit „Towards Predicting Feature Defects in Software Product Lines“ von Queiroz et al. [24] knüpft an den zuvor vorgestellten Ansatz der Software-Features an. Beide Literaturquellen werden im weiteren Verlauf dieser Arbeit eine Rolle spielen, die im jeweiligen Abschnitt erläutert wird.

Dateibasierte Fehlervorhersage

Das Beispiel zur dateibasierten Fehlervorhersage stammt aus einer wissenschaftlichen Arbeit von Moser et al. [18]. Sie widmet sich einer vergleichenden Analyse von zwei verschiedenen Mengen von Metriken zur dateibasierten Fehlervorhersage mittels Methoden des Machine Learnings. Die Zuordnung erfolgt in „defekt“ und „defekt-frei“. Als Datenbasis dient ein vorgefertigtes Datenset von Eclipse. Auf Basis dieses Datensets wurden „produktbasierte“ Metriken (Codemetriken) und Prozessmetriken berechnet. Zur Anwendung kamen die Klassifikationsalgorithmen logistische Regression, Naïve Bayes und Entscheidungsbäume. Die Prozessmetriken stellen eine Besonderheit dieser Arbeit dar, da sie in dieser Arbeit zum ersten Mal näher betrachtet und auf ihre Eignung als Attribute zum Training der Klassifikatoren erörtert wurden. Die Metriken berechneten unter anderem die Anzahl der Änderungen an einer Datei, die Anzahl der Autoren einer Datei, die Anzahl der hinzugefügten oder entfernten Zeilen einer Datei oder das Alter einer Datei. Das Resultat der Arbeit lautet, dass Prozessmetriken effektiver

zur Fehlervorhersage genutzt werden können als Codemetriken. Im Folgenden werden die 17 Prozessmetriken samt ihrer Beschreibungen und Abkürzungen (für diese Arbeit) vorgestellt.

- REVISIONS (REVI)
  Anzahl der Revisionen (Bearbeitungen) der Datei.
- REFACTORINGS (REFA)
  Anzahl der Fälle, in denen die Datei in einem Refactoring involviert war. Basierend auf Analyse der Commit-Nachricht auf das Vorhandensein des Begriffs "refactor".
- BUGFIXES (BUGF)
  Anzahl der Fälle, in denen die Datei in einer Fehlerbehebung involviert war.
- AUTHORS (AUTH)
  Anzahl der verschiedenen Autoren, die die Datei in das Repository eingecheckt haben.
- LOC_ADDED (ADDL)
  Summe der zur Datei hinzugefügten Codezeilen über alle Revisionen.
- MAX_LOC_ADDED (ADDM)
  Maximale Anzahl von Codezeilen, die für alle Revisionen hinzugefügt wurden.
- AVE_LOC_ADDED (ADDA)
  Durchschnittlich hinzugefügte Codezeilen pro Revision.
- LOC_DELETED (REML)
  Summe der von der Datei entfernten Codezeilen über alle Revisionen.
- MAX_LOC_DELETED (REMM)
  Maximale Anzahl von Codezeilen, die für alle Revisionen entfernt wurden.
- AVE_LOC_DELETED (REMA)
  Durchschnittlich entfernte Codezeilen pro Revision.
- CODECHURN (CCHN)
  Summe von (hinzugefügte Codezeilen - entfernte Codezeilen) über alle Revisionen.
- MAX_CODECHURN (CCHM)
  Maximaler CODECHURN für alle Revisionen.
- AVE_CODECHURN (CCHA)
  Durchschnittlicher CODECHURN pro Revision.
- MAX_CHANGESET (MAXC)
  Maximale Anzahl von Dateien, die gemeinsam committed wurden.
- AVE_CHANGESET (AVGC)
  Durchschnittliche Anzahl von Dateien, die gemeinsam committed wurden.
- AGE (AAGE)
  Alter der Datei in Wochen (rückwärts zählend bis zu einem bestimmten Release).
- WEIGHTED_AGE (WAGE)
  $WeightedAge = \frac{\sum_{i=1}^{N} Age(i) * LOC\_ADDED(i)}{\sum_{i=1}^{N} LOC\_ADDED(i)}$

Im Rahmen der Evaluation (Abschnitt 5.3) dienen diese dateibasierten Metriken als Grundlage für den Vergleich, ob die Metriken des für diese Arbeit erstellten featurebasierten Datensets einen Einfluss auf die Ergebnisse der Fehlervorhersage auf Dateiebene hervorrufen.

Featurebasierte Fehlervorhersage

Das Beispiel zur featurebasierten Fehlervorhersage stammt aus einer wissenschaftlichen Arbeit von Queiroz et al. [24]. Bei dieser Fallstudie handelt es sich um die erste und bisher einzige Arbeit über Fehlervorhersage mit Bezug zu Software-Features. Sie stellt somit für diese Masterarbeit eine bedeutende literarische Grundlage dar. Der Ablauf des von Queiroz et al. angewandten Prozesses zur Erstellung eines featurebasierten Datensets und dessen Anwendung zum Training von Klassifikatoren orientiert sich am zuvor vorgestellten allgemeinen Prozess des überwachten Machine Learnings.

Die Erläuterung des Beispiels erfolgt anhand von zwei Abbildungen, welche den in der Arbeit von Queiroz et al. vorgestellten Prozess in zwei Teilen visualisieren. Der erste Teil ist in Abbildung 2.6 dargestellt.

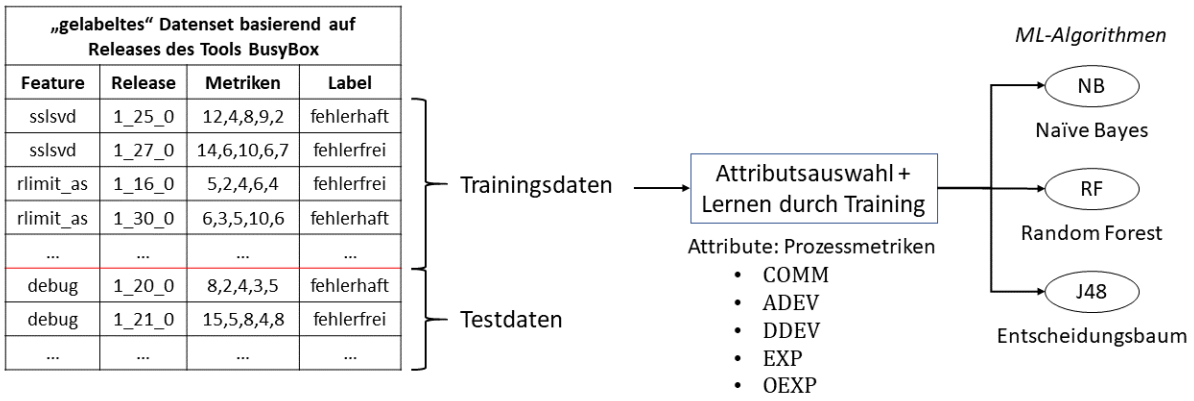


Abbildung 2.6: Teil 1: Featurebasierter Prozess des überwachten Machine Learnings nach [24]

Datenset

Die Datenbasis des Datensets bilden historische Commits des UNIX-Toolkits BusyBox[1], dessen Quellcode frei verfügbar in einem Git-Repository[2] eingesehen und von dort geklont werden kann. Diese Commits wurden wiederum ihren entsprechenden Releases zugeordnet, welche auf der vergebenen Tag-Struktur des Repositories beruhen. Ferner wurden aus den Diffs der Commits die dort bearbeiteten Features extrahiert und anschließend zusammen mit den Release-Informationen in einer MySQL-Datenbank gespeichert. Zusätzlich enthält jeder Datenbankeintrag aggregierte Werte von fünf auf das Feature und den Release bezogenen Prozessmetriken (Erläuterung folgt) sowie das binäre Label, ob ein Feature in einem Release fehlerhaft oder fehlerfrei war. Ein Feature gilt in einem Release als fehlerhaft, sofern in einem Commit des darauffolgenden Releases ein fehlerbehebender Commit bezüglich des Features festgestellt werden konnte. Dies geschieht über die Analyse der Commit-Nachrichten. Sofern eine Commit-Nachricht die Begriffe „bug“ (Fehler), „error“ (schwerwiegender Fehler), „fail“ (fehlschlagen) oder „fix“ (beheben) enthält, werten die Autoren des Papers den Commit als fehlerbehebend. Alternative Methoden zur Durchführung dieser Analyse bestehen aus der Einbindung von Daten aus Bug-Tracking-Systemen, die häufig an Software-Repositories angebunden sind, sowie aus der Anwendung des sogenannten SZZ-Algorithmus, welcher in dieser Arbeit verwendet wurde und in Abschnitt 3.2 erläutert wird [31, 41]. Wie im Rahmen des überwachten Machine Learning üblich, wird das Datenset in Trainings- und Testdaten in einem Verhältnis von 75:25 geteilt.

Metriken und Klassifikation

Die Trainingsdaten werden dann den Klassifikatoren zum Training zur Verfügung gestellt. Als Attribute dienen fünf Prozessmetriken mit spezifischer Betrachtung von Software-Features. Einen Überblick über die Beschreibungen dieser gibt Tabelle 2.1. Diese Metriken werden auch für diese Arbeit im Rahmen der Erstellung des featurebasierten Datensets übernommen. Als Klassifikationsalgorithmen wurden Naïve Bayes, Random Forest und J48-Entscheidungsbäume gewählt.

[1] `https://busybox.net/`
[2] `https://git.busybox.net/busybox/`

Tabelle 2.1: Übersicht der von [24] verwendeten Prozessmetriken

Metrik	Beschreibung
COMM	Anzahl der Commits, die in einem Release dem betreffenden Feature gewidmet sind.
ADEV	Anzahl der Entwickler, die das betreffende Feature in einem Release bearbeitet haben.
DDEV	kummulierte Anzahl der Entwickler, die das betreffende Feature in einem Release bearbeitet haben.
EXP	Geometrisches Mittel der „Erfahrung*“ aller Entwickler, die am betreffenden Feature in einem Release gearbeitet haben.
OEXP	„Erfahrung*“ des Entwicklers, der am meisten zum betreffenden Feature in einem Release beigetragen hat.
*Erfahrung ist definiert als Summe der geänderten, gelöschten oder hinzugefügten Zeilen im zugehörigen Release.	

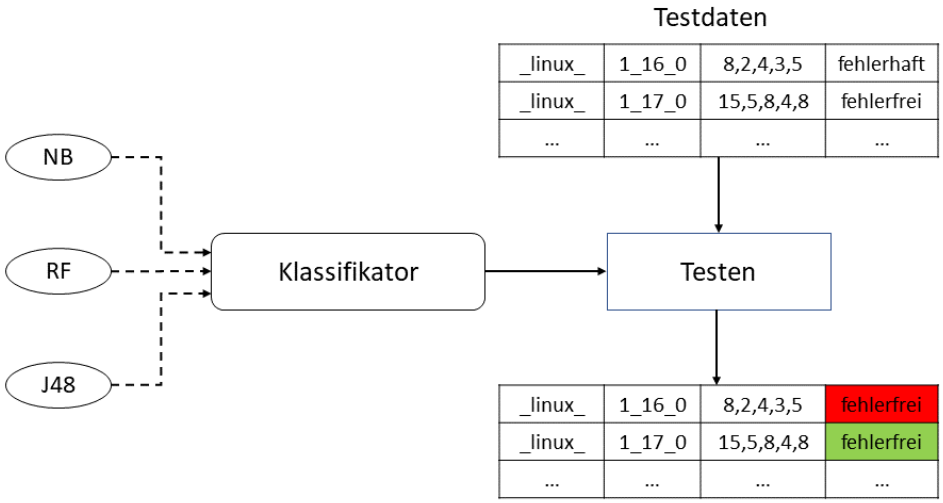


Abbildung 2.7: Teil 2: Featurebasierter Prozess des überwachten Machine Learnings nach [24]

Test der Klassifikatoren

Wie in Abbildung 2.7 dargestellt ist, wird für jeden Klassifikationsalgorithmus ein Klassifikator erstellt, welcher anschließend getestet und evaluiert wird. Dazu werden die jeweiligen Klassifikatoren auf das Testdatenset angewendet, ohne jedoch die Werte der Zielklassen mit anzugeben. Diese werden im Anschluss an den Klassifikationsvorgang mit den vorhergesagten Werten auf Übereinstimmung verglichen. Anhand dieses Vergleiches können die Genauigkeit sowie weitere Metriken zur Bewertung der Leistung der Klassifikatoren gemessen werden. Eine Übersicht von Evaluationsmetriken kann in Abschnitt 5.2.1 gefunden werden.

Die so erstellten Klassifikatoren können dann zur Vorhersage von neuen und unbekannten Daten genutzt werden, um defekte Features eines zukünftigen Releases zu identifizieren. Dazu müssen die fünf zuvor genannten Prozessmetriken dieser Daten berechnet werden.

Kapitel 3

Erstellung eines featurebasierten Datensets

Dieses Kapitel widmet sich der schrittweisen Erläuterung des Prozesses zur Erstellung des featurebasierten Datensets, welches zum Training der Machine-Learning-Klassifikatoren dient. Dazu wird zunächst die Datenauswahl näher beleuchtet. Darauf folgt eine Darlegung der Konstruktion des Datensets sowie der Auswahl und Berechnung der Metriken, welche als Attribute im Rahmen des Trainings der Klassifikatoren dienen. Eine Gliederung der Kapitel kann Abbildung 3.1 entnommen werden.

Abbildung 3.1: Übersicht zur Gliederung des dritten Kapitels

3.1 Datenauswahl

Wie im vorangegangenen Kapitel bereits erwähnt wurde, bildet das Datenset die Grundlage für das Training der Machine-Learning-Klassifikatoren und wird eigens für diese Arbeit auf Basis von Commit-Daten von 13 featurebasierten Softwareprojekten erstellt. Die Auswahl der Softwareprojekte erfolge zum einen anhand von vorheriger Verwendung in wissenschaftlicher Literatur [12, 14, 25, 24]. Wichtig war zum anderen jedoch auch, dass die Variabilität im Sourcecode der Softwareprojekte mittels den in in Abschnitt 2.1 vorgestellten Präprozessor-Direktiven umgesetzt wurde und ihre Git-Repositories eine klare „Revision History“ hinsichtlich von Release-Nummern aufweisen. Ein weiteres hinreichendes Kriterium war die Verwendung einer einheitlichen Sprache. Sämtliche Softwareprojekte entstammen dem englischsprachigen Raum. Die für diese Arbeit verwendeten Softwareprojekte sind samt ihres Einsatzzweckes und ihrer Datenquellen in Tabelle 3.1 aufgeführt.

[1]Links zu den Websites der Softwareprojekte und deren Repositories können im Anhang eingesehen werden.

Tabelle 3.1: Übersicht der verwendeten Softwareprojekte[1]

Projekt	Zweck	Datenquelle
Blender	3D-Modellierungstool	GitHub-Mirror
Busybox	UNIX-Toolkit	Git-Repository
Emacs	Texteditor	GitHub-Mirror
GIMP	Bildbearbeitung	GitLab-Repository
Gnumeric	Tabellenkalkulation	GitLab-Repository
gnuplot	Plotting-Tool	GitHub-Mirror
Irssi	IRC-Client	GitHub-Repository

Projekt	Zweck	Datenquelle
libxml2	XML-Parser	GitLab-Repository
lighttpd	Webserver	Git-Repository
MPSolve	Polynomlöser	GitHub-Repository
Parrot	virtuelle Maschine	GitHub-Repository
Vim	Texteditor	GitHub-Repository
xfig	Grafikeditor	Sourceforge-Repository

Tabelle 3.2: Übersicht der Anzahl der Releases und Commits je Softwareprojekt

Projekt	#Releases	#Commits
Blender	11	19.119
Busybox	14	4.984
Emacs	7	12.805
GIMP	14	7.240
Gnumeric	8	6.025
gnuplot	5	6.619
Irssi	7	253

Projekt	#Releases	#Commits
libxml2	10	732
lighttpd	6	2.597
MPSolve	8	668
Parrot	7	16.245
Vim	7	9.849
xfig	7	18

Um die Commit-Daten der Softwareprojekte zu erhalten wurde die Python-Library PyDriller[2] verwendet [33]. Diese ermöglicht eine einfache Datenextraktion aus Git-Repositories zum Erhalt von Commits, Commit-Nachrichten, Entwicklern, Diffs und mehr (im weiteren Verlauf „Metadaten“ genannt). Ein beispielhafter Sourcecode-Ausschnitt zur Konsolenausgabe von Metadaten eines Commits (Autor, Name der veränderten Dateien, Typ der Veränderung und jeweilige zyklomatische Komplexität der Dateien) kann dem Anhang entnommen werden..

Als Input der eigens erstellten Python-Scripte zum Abruf der Commit-Metadaten dienten jeweils die URLs zu den Git-Repositories der Softwareprojekte. Weiterhin wurden die Metadaten in Commits je Release aufgeteilt. Ermöglicht wurde dies durch die Angabe von Release-Tags im PyDriller-Code, basierend auf der Tag-Struktur von Git-Repositories. Für jede veränderte Datei innerhalb eines Commits und eines Releases wurden die folgenden Metadaten mit Hilfe von PyDriller abgerufen:

- Commit-Hash (eindeutiger Bezeichner des zugehörigen Commits)
- Autor des zugehörigen Commits
- zugehörige Commit-Nachricht
- Name der veränderten Datei
- Lines-of-Code der veränderten Datei
- zyklomatische Komplexität der veränderten Datei
- Anzahl der hinzugefügten Zeilen zur Datei
- Anzahl der entfernten Zeilen von der Datei
- Art der Änderung (ADD, REM, MOD)[3]
- Diff (Changeset) der Veränderung

Die auf diese Weise erhaltenen Daten wurden nach dem Abruf in einer MySQL-Datenbank gespeichert. Für jedes Softwareprojekt wurde eine eigene Tabelle erstellt, in welcher neben der oben stehenden Metadaten zudem der Name des betreffenden Softwareprojekts und die den Commits zugehörigen Release-Nummern gespeichert wurden. Jede veränderte Datei eines Commits erhält eine Zeile der Datenbank-Tabellen. In Tabelle 3.2 kann eingesehen werden, wie viele Releases je Softwareprojekt abgerufen wurden und wie viele Commits daraus resultieren.

[2]https://github.com/ishepard/pydriller

[3]Diese Information fand in der weiteren Erstellung des Datensets keine Verwendung.

Diese „Rohdaten“ dienen zur weiteren Verarbeitung hinsichtlich der Erstellung des Datensets und der anschließenden Berechnung der Metriken. Eine Erläuterung der weiteren Verarbeitung der Daten folgt im kommenden Abschnitt.

RQ1a: WELCHE DATEN KOMMEN FÜR DIE ERSTELLUNG DES DATENSETS IN FRAGE?

Es kommen die Daten von 13 featurebasierten Softwareprojekten zur Verwendung. Mithilfe der Python-Library PyDriller wurden die Metadaten der Commits, aufgeteilt nach Commits pro Release, aus den Git-Repositories extrahiert und in MySQL-Datenbanken gespeichert. Ausgewählt wurden die Softwareprojekte aufgrund einer vorherigen Verwendung in der wissenschaftlichen Literatur [12, 14, 24].

3.2 Konstruktion des Datensets

Die Konstruktion des Datensets gliedert sich in mehrere Phasen der Datenverarbeitung und -optimierung. Diese werden im folgenden vorgestellt.

Identifikation von Features

Die erste Phase besteht aus der Extraktion der involvierten Features einer veränderten Datei. Dazu wurden mithilfe eines Python-Scripts die Präprozessor-Anweisungen `#IFDEF` und `#IFNDEF` in den Diffs der veränderten Dateien identifiziert und anschließend die den Direktiven folgende Zeichenfolge bis zum Ende der Codezeile als Feature gespeichert. Die Identifizierung erfolgte mittels regulären Ausdrücken. Gespeichert werden die je Datei identifizierten Features in einer zusätzlichen Spalte in den jeweiligen MySQL-Tabellen der Metadaten der Softwareprojekte. Für den Fall, dass ein Feature hinter der Direktive `#IFNDEF` identifiziert wurde, wird das Feature mit einem vorangestellten „not“ gespeichert. Es wird somit als eigenständiges Feature, neben seiner nicht-negierten Form, gespeichert. Kombinationen von Features werden in ihrer identifizierten Form gespeichert. Konnte kein Feature identifiziert werden, wird entsprechend „`none`“ gespeichert.

Dieser Weg der Identifizierung birgt einige Hindernisse. Diese können, neben dem Normalfall, in Abbildung 3.2 gesehen werden. In einigen C-Programmierparadigmen ist es üblich, Header-Dateien mittels Präprozessor-Direktiven im Sourcecode einzubinden, sodass sie wie Features scheinen (siehe erster unerwünschter Fall in Abbildung 3.2). Diese „Header-Features“, wie sie im weiteren Verlauf genannt werden, sollten jedoch ignoriert werden, da sie im gesamten Sourcecode keine Variabilität erzeugen. In der Regel sind diese Header-Features identifizierbar durch ihre Namensgebung in Form eines angehängten `_h_` an den Featurenamen, wie beispielsweise `featurename_h_`. Dieser angehängte Teil erlaubt es, die Header-Features mittels regulärer Ausdrücke zu erkennen und auszufiltern.

Ebenfalls besteht die Möglichkeit, dass „falsche“ Features identifiziert werden können. Beispiele dafür können von `#IFDEFs` stammen, welche in Kommentaren verwendet wurden (siehe zweiter unerwünschter Fall in Abbildung 3.2). Solche falschen Features wurden in einer manuellen Sichtung der identifizierten Features entfernt und durch „`none`“ ersetzt.

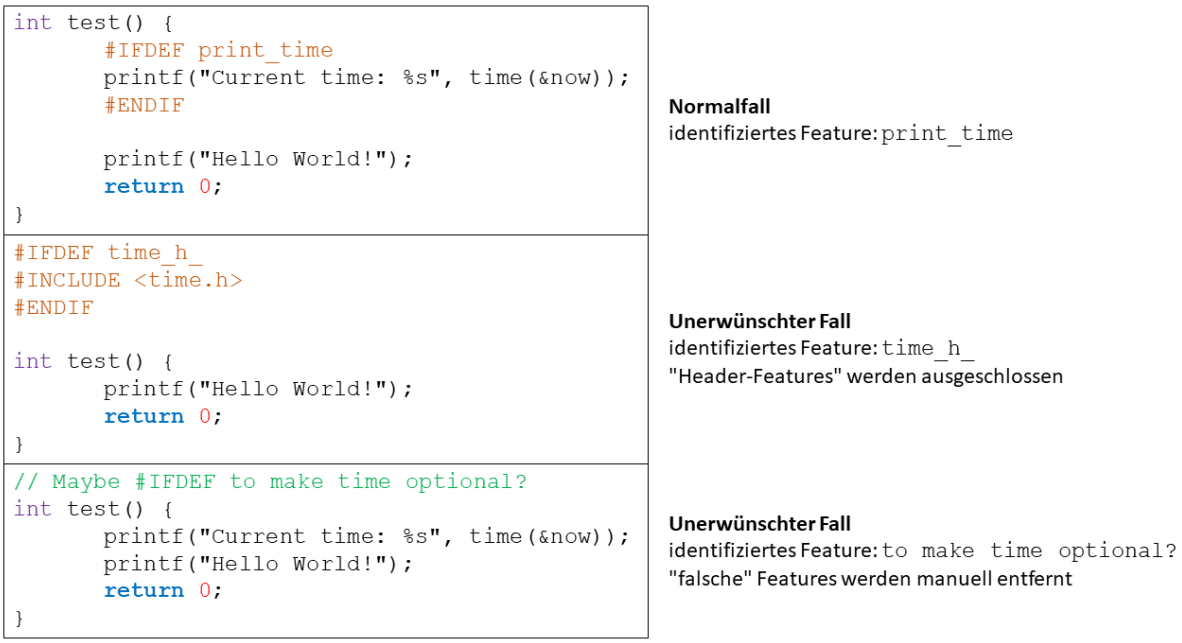


Abbildung 3.2: Normalfall und unerwünschte Fälle bei der Identifizierung der Features

Identifikation von korrektiven Commits

Die nächste Phase der Verarbeitung besteht aus der Identifizierung von korrektiven Commits. Eine dafür gängige Methode, die auch in dieser Arbeit Anwendung fand, besteht aus der Analyse der Commit-Nachrichten auf das Vorhandensein von bestimmten Schlagwörtern [41]. Bei den verwendeten Schlagwörtern handelt es sich um „bug“,„bugs“, „bugfix“, „error“, „fail“, „fix“, „fixed“ und „fixes“. Durchgeführt wurde die Analyse mittels eines Python-Skripts, welches überprüft, ob sich eines der Schlagworte alleinstehend oder innerhalb einer Wortkombination in der Commit-Nachricht befindet. Dabei wurde die Identifizierung auf die jeweils erste Zeile der Commit-Nachrichten beschränkt (siehe Anmerkung in Abschnitt 5.1). Die Ergebnisse wurden in einer weiteren Spalte („corrective“) der MySQL-Tabellen (true = korrektiv, false = nicht korrektiv) gespeichert.

Identifikation fehlereinführender Commits

Der Suche nach korrektiven Commits folgt eine Analyse nach fehlereinführenden Commits. Dazu wurde eine PyDriller-Implementierung des SZZ-Algorithmus nach Sliwerski, Zimmermann und Zeller verwendet [31, 33]. Dieser Algorithmus erlaubt es fehlereinführende Commits in lokal gespeicherten Software-Repositories zu finden [5]. Dafür setzt er voraus, dass die korrektiven Commits bereits identifiziert wurden, da sie als Eingabemenge des Algorithmus dienen [5]. Die Identifikation der fehlereinführenden Commits ist in mehrere Schritte unterteilt und wird in Abbildung 3.3 dargestellt. Die Erläuterungen der mit Buchstaben versehenen Schritte erfolgt im Anschluss. Die für diese Arbeit verwendete PyDriller-Implementierung des Algorithmus folgt dem gezeigten Ablauf.

Der SZZ-Algorithmus, der als Input eine Liste der Commit-Hashes der zuvor erkannten korrektiven Commits (a) erhält, beginnt mit der Ausführung eines `git blame` Befehls (b) zur Identifizierung sämtlicher Commits, in denen Veränderungen an den selben Dateien und Codezeilen vorgenommen wurden wie in den korrektiven Commits [5]. Daraus resultieren mögliche

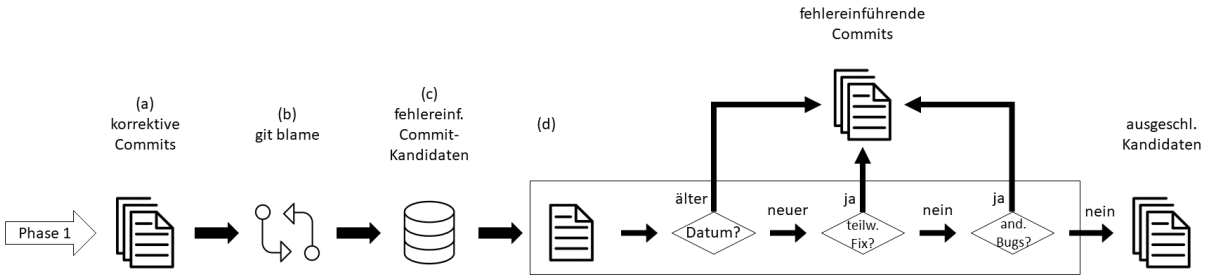


Abbildung 3.3: Ablauf der zweiten Phase des SZZ-Algorithmus (übersetzt, [5])

Tabelle 3.3: Anzahl der korrektiven und fehlereinführenden Commits sowie Anzahl der identifizierten Features je Softwareprojekt

Projekt	#korrektiv	#fehlereinführend	#Features
Blender	7.760	3.776	1.400
Busybox	1.236	802	628
Emacs	4.269	2.532	718
GIMP	1.380	854	204
Gnumeric	1.498	1.191	637
gnuplot	854	1.215	558
Irssi	52	22	9
libxml2	324	88	200
lighttpd	1.078	929	230
MPSolve	151	211	54
Parrot	3.109	3.072	397
Vim	371	696	1.158
xfig	0*	0*	137
* Siehe Abschnitt 5.1			

fehlereinführende Commit-Kandidaten (c). Für jeden dieser Commit-Kandidaten wird dann erörtert, ob er fehlereinführend ist (d). Dazu wird zunächst das Datum des Commit-Kandidaten mit dem zugehörigen korrektiven Commits verglichen. Liegt dieses vor dem Datum des korrektiven Commits, so gilt der Kandidat als tatsächlich fehlereinführend [5]. Liegt das Datum jedoch später, so kann der Kandidat nur fehlereinführend sein, sofern er teilweise den vorhandenen Fehler löst (teilweiser Fix) oder für einen anderen Fehler verantwortlich ist, der nicht dem korrektiven Commit zugehörig ist (Kandidat ist Fehlerursache eines anderen korrektiven Commits) [5]. Die Ausgabe ist eine Liste von Commit-Hashes von fehlereinführenden Commits für jede Datei eines korrektiven Commits. Diese neuen Informationen werden in einer zusätzlichen Spalte in den MySQL-Tabellen für jeden Eintrag gespeichert (true = fehlereinführend, false = nicht fehlereinführend).

Eine Übersicht der Anzahl der korrektiven und fehlereinführenden Commits sowie der Anzahl der identifizierten Features je Softwareprojekt ist in Tabelle 3.3 aufgeführt. Des Weiteren ist eine Übersicht des Schemas der nun vollständigen initialen MySQL-Tabellen (im folgenden Haupttabellen genannt) in Tabelle 3.4 aufgeführt. Wie bereits zuvor erwähnt, umfasst diese Tabelle für jede veränderte Datei eines Commits eine Zeile. Sollten in einem Diff einer veränderten Datei mehrere Features identifiziert worden sein, so wird für jedes Feature die entsprechende Zeile dupliziert.

Ein reales Beispiel aus der Haupttabelle des Softwareprojektes Vim, welches die Diffs eines korrektiven (A) und eines fehlereinführenden (B) Commits zu einem Feature `FEAT_TEXT_PROP` zeigt, ist in Abbildung 3.4 dargestellt. Folgende Informationen können zu der betreffenden Datei (das Feature befindet sich sowohl im korrektiven als auch im fehlereinführenden Fall in

Tabelle 3.4: Übersicht des Schemas der MySQL-Haupttabellen

Spaltenname	Beschreibung
name	Name des Softwareprojekts
release_number	zugehörige Release-Version basierend auf vergebenen Tags
commit_hash	eindeutiger Bezeichner eines Commits
commit_author	Autor eines Commits
commit_msg	Nachricht eines Commits
filename	Name der geänderten Datei
nloc	„Lines of code“ der geänderten Datei
cycomplexity	Zyklomatische Komplexität der geänderten Datei
lines_added	Anzahl der hinzugefügten Zeilen zur geänderten Datei
lines_removed	Anzahl der entfernten Zeilen von der geänderten Datei
change_type	Art der Änderung (*ohne weitere Verwendung*)
diff	Diff der geänderten Datei
corrective	Indikator, ob Commit korrektiv war
bug_introducing	Indikator, ob Commit fehlereinführend war
feature	Namen der zugehörigen Features der geänderten Datei

der selben Datei) des korrektiven Commits[4] (A) aus den Daten der Haupttabellen entnommen werden:

- Datei: `screen.c`
- Commit-Hash: `1748c7f77ea864c669b7e5cfb2be0c34ce45e36e`
- Commit-Nachrich: `patch 8.1.1495: memory access` **`error.`** `problem: memory access` **`error.`** `solution: use the correct size for clearing the popup mask.`

Der Ausschnitt des Diffs zeigt zudem, dass der Methodenaufruf `vim_memset` mit abweichenden Argumenten ersetzt wurde. Laut zugehöriger Commit-Nachricht führte der ursprüngliche Methodenaufruf zu einem „Memory Access Error“. Dieser Commit wurde als korrektiv identifiziert, da die Commit-Nachricht das Schlagwort „error“enthält. Der entsprechende Eintrag in der Haupttabelle von Vim erhält somit in der Spalte „corrective“ den Wert `true`. Mithilfe des SZZ-Algorithmus konnte, unter Angabe des Commit-Hashes des korrektiven Commits, der fehlereinführende Commit[5] (B) der betroffenen Datei ermittelt werden. In dessen Ausschnitt des Diffs ist zu erkennen, dass mit diesem Commit das Feature `FEAT_TEXT_PROP` in der Datei mit dem fehlerhaften Methodenaufruf eingepflegt wurde. Folglich bekommt dieser in der Haupttabelle in der Spalte „bug_introducing“ den Wert `true` zugewiesen.

Auf Basis der Daten der Haupttabellen können nun die für das Training der Klassifikatoren benötigten Metriken berechnet werden.

3.3 Metriken

Wie bereits in Abschnitt 2.2 erwähnt wurde, dienen Attribute zum Training der Machine-Learning-Klassifikatoren. Im hier vorliegenden Szenario werden sogenannte Metriken als Attribute verwendet. Bei Metriken handelt sich sich um Zahlenwerte, die Eigenschaften eines

[4]Link zum Commit im Git-Repository von Vim: `https://github.com/vim/vim/commit/1748c7f77ea864c669b7e5cfb2be0c34ce45e36e`

[5]Link zum Commit im Git-Repository von Vim: `https://github.com/vim/vim/commit/33796b39b9f00b42ca57fa00dbbb52316d9d38ff`

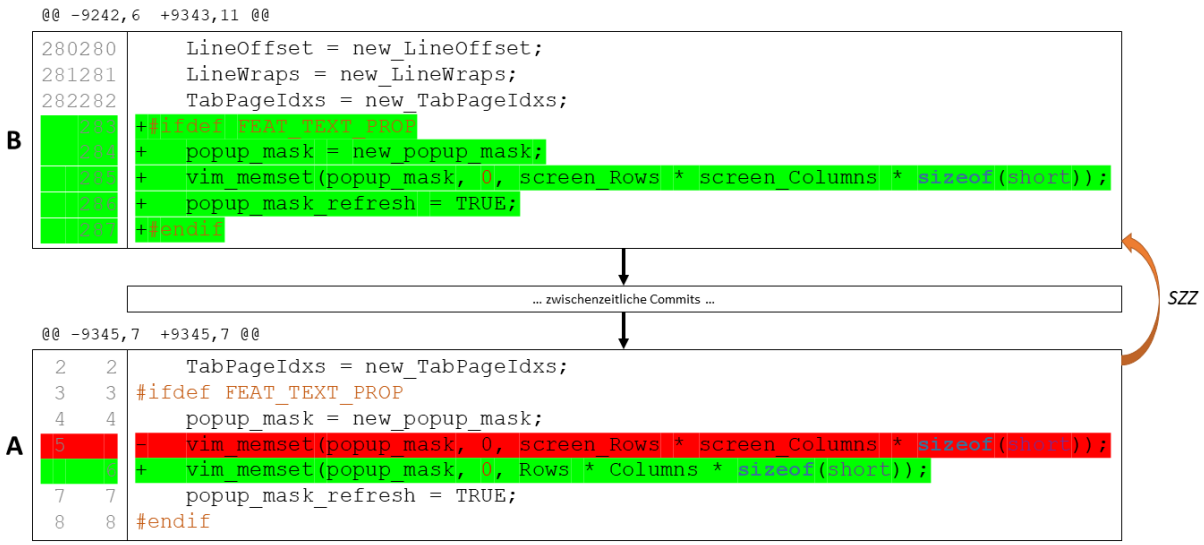


Abbildung 3.4: Reales Beispiel eines Fehlers mit korrektivem (A) und fehlereinführendem (B) Commit

Softwareprojekts quantifizieren. Die Metriken werden im hier vorliegenden Fall in die üblichen Kategorien Codemetriken und Prozessmetriken aufgeteilt und jeweils anhand der vorhandenen Rohdaten der Haupttabellen berechnet [26]. Codemetriken werden genutzt um Eigenschaften von Sourcecode, wie zum Beispiel „Größe“ oder Komplexität, zu messen [26]. Prozessmetriken dienen hingegen zur Messung von Eigenschaften, die anhand von Metadaten aus Software-Repositories erörtert werden können [26]. Beispiele dafür sind die Anzahl der Veränderungen einer bestimmten Datei oder die Anzahl der aktiven Entwickler an einem Projekt. Für diese Arbeit wurden elf featurebasierte Metriken errechnet, aufgeteilt in sieben Prozess- und vier Codemetriken. Fünf der Prozessmetriken wurden aus wissenschaftlichen Arbeiten [26, 24] entnommen. Die weiteren sechs Metriken wurden auf Basis der von PyDriller erhaltenen Metadaten der Commits berechnet. Eine Auflistung der elf Metriken samt Beschreibungen ist in Tabelle 3.5 aufgeführt. Die Berechnung der einzelnen Metriken erfolgte entweder direkt mittels SQL-Abfragen oder kombiniert mittels SQL-Abfragen und Berechnungen durch ein Python-Skript.

Im Hinblick auf die im nächsten Kapitel folgende Evaluation des featurebasierten Datensets, wurden zwei weitere Datensets erstellt, um die Auswirkungen der featurebasierten Metriken auf die Vorhersagen der Klassifikatoren vergleichbar zu machen. Beide weiteren Datensets verfolgen einen klassischen dateibasierten Ansatz, wie er in der Machine-Learning-gestützten Fehlererkennung üblich ist und basieren auf der Methodik, die von Moser et al. [18], die bereits in Abschnitt 2.3 vorgestellt wurde. Ebenfalls wurden die 17 Prozessmetriken dieser wissenschaftlichen Arbeit übernommen. Eine Visualisierung zur Unterscheidung der drei Datensets ist in Abbildung 3.5 dargestellt.

Die Abbildung zeigt die Wege der Erstellung des featurebasierten Datensets (Pfad A - G) und der dateibasierten Datensets nach [18]. Diese unterteilen sich in das „einfache“ dateibasierte Datenset (Pfad A, B, H, I) und das erweiterte dateibasierte Datenset (Kombination beider Pfade). Die Erstellung des featurebasierten Datensets wurde bereits umfassend bis zu diesem Abschnitt erläutert. Weitere Informationen zur Finalisierung des Datensets folgen im Verlauf dieses Abschnitts.

Die Erstellung der dateibasierten Datensets erfolgt ebenfalls auf Basis der mit PyDriller erhaltenen Rohdaten der Commits der Softwareprojekte und den daraus extrahierten veränderten Dateien (A + B). Es ist somit keine weitere Verarbeitung der Rohdaten nötig. Sie dienen dann

Tabelle 3.5: Übersicht der berechneten Metriken des featurebasierten Datensets

	Metrik	Beschreibung	Quelle
Prozessmetriken	Anzahl der Commits (COMM)	Anzahl der Commits, die dem Feature in einem Release zugeordnet sind.	[26, 24]
	Anzahl der aktiven Entwickler (ADEV)	Anzahl der Entwickler, die innerhalb eines Releases das Feature bearbeitet (geändert, gelöscht oder hinzugefügt) haben.	[26, 24]
	eindeutige Entwickleranzahl (DDEV)	kumulierte Anzahl der Entwickler, die innerhalb eines Releases das Feature bearbeitet (geändert, gelöscht oder hinzugefügt) haben.	[26, 24]
	Erfahrung aller Entwickler (EXP)	geometrisches Mittel der „Erfahrung“ aller Entwickler, die innerhalb eines Releases das Feature bearbeitet (geändert, gelöscht oder hinzugefügt) haben. Erfahrung ist definiert als Summe der geänderten, gelöschten oder hinzugefügten Zeilen in den dem Feature zugeordneten Dateien.	[26, 24]
	Erfahrung des meist beteiligten Entwicklers (OEXP)	„Erfahrung“ des Entwicklers, die innerhalb eines Releases das Feature am häufigsten bearbeitet (geändert, gelöscht oder hinzugefügt) hat. Erfahrung ist definiert als Summe der geänderten, gelöschten oder hinzugefügten Zeilen in den dem Feature zugeordneten Dateien.	[26, 24]
	Grad der Änderungen (MODD)	Anzahl der Bearbeitungen (Änderung, Entfernung, Erweiterung) des Features innerhalb eines Releases.	*
	Umfang der Änderungen (MODS)	Anzahl der bearbeiteten Features innerhalb eines Releases (featureübergreifender Wert). Idee: Je mehr Features in einem Release bearbeitet worden sind, desto fehleranfälliger scheinen diese zu sein.	*
Codemetriken	Anzahl der Codezeilen (NLOC)	Durchschnittliche Anzahl der Codezeilen der dem Feature zugeordneten Dateien innerhalb eines Releases.	*
	Zyklomatische Komplexität (CYCO)	Durchschnittliche zyklomatische Komplexität der dem Feature zugeordneten Dateien innerhalb eines Releases.	*
	Anzahl der hinzugefügten Zeilen (ADDL)	Durchschnittliche Anzahl der hinzugefügten Codezeilen zu den dem Feature zugeordneten Dateien innerhalb eines Releases.	*
	Anzahl der entfernten Zeilen (REML)	Durchschnittliche Anzahl der gelöschten Codezeilen von den dem Feature zugeordneten Dateien innerhalb eines Releases.	*
** Diese Werte wurden auf Basis der mit PyDriller erhaltenen Metadaten berechnet. Die Berechnung der Metriken auf Feature-Level erfolgte auf Basis der Metadaten der ihnen zugrunde liegenden Dateien.*			

als Grundlage zur Berechnung der 17 Metriken (H) nach [18]. Eine Übersicht der Metriken erfolgte bereits in Abschnitt 2.3. Für die Berechnung der zeitbasierten Metriken `AAGE` und `WAGE` mussten die Rohdaten beziehungsweise die in den Rohdaten gelisteten Commits mit ihrem Veröffentlichungsdatum ergänzt werden. Durchgeführt wurde dies ebenfalls mit PyDriller. Als Startpunkt für die Berechnung der vergangenen Wochen wurde jeweils das Datum des ersten Commits eines Releases gewählt.

Das featurebasierte sowie das „einfache“ dateibasierte Datenset (G + I) bestehen aus den jeweils berechneten feature- (F) und dateibezogenen (H) Metriken sowie den Labeln der Zielklasse. Berechnet werden die Werte der Metriken für die Daten jedes Softwareprojektes. Die daraus resultierenden Tabellen enthalten als Spalten die Werte der elf beziehungsweise 17 Metriken sowie das Label (Zielklasse) „defekt“ oder „fehlerfrei“ und als Zeilen die Features bzw. Dateien aggregiert nach Release. Dies bedeutet, dass für den Fall, dass ein Feature oder eine Datei in einem Release mehrfach bearbeitet wurden (d.h. es wird in mehreren Commits bearbeitet), der Durchschnittswert der jeweiligen Metriken innerhalb des Releases berechnet und gespeichert wird. Das Verfahren zur Bestimmung des Labels erfolgt für jede Veränderung eines Features beziehungsweise einer Datei anhand des folgenden Musters:

fehlereinführend	+	korrektiv	=	defekt
fehlereinführend	+	nicht korrektiv	=	defekt
nicht fehlereinführend	+	korrektiv	=	fehlerfrei
nicht fehlereinführend	+	nicht korrektiv	=	fehlerfrei

Die Informationen zum Status eines Features basieren dabei auf den ihm zugrundliegenden Dateien. Für den Fall, dass ein Feature oder eine Datei mehrfach innerhalb eines Releases bearbeitet sein sollte, erfolgt die Bestimmung des Labels anhand der folgenden Regeln:

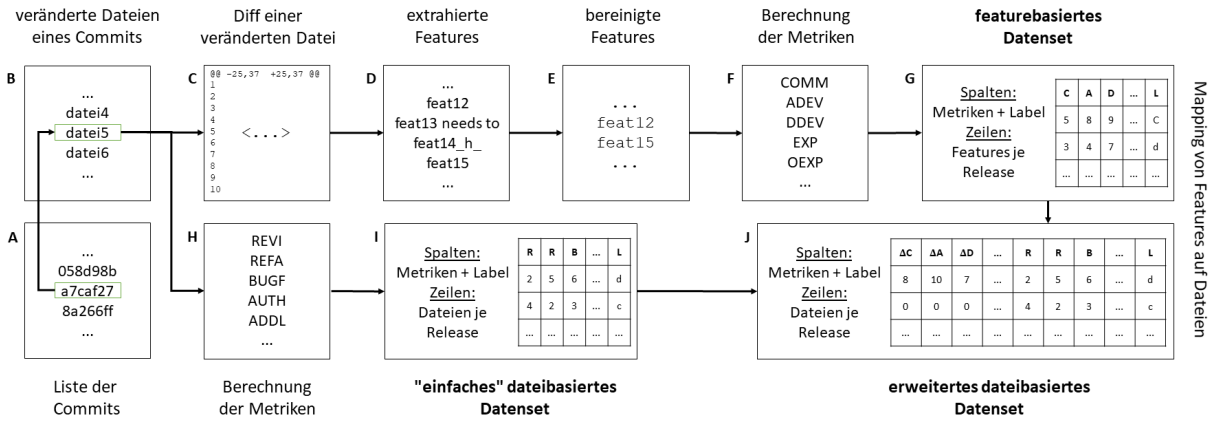


Abbildung 3.5: Visualisierung zur Unterscheidung der Datensets

- im Fall von Features wird überprüft, ob innerhalb des Releases das Feature mindestens ein Mal als „defekt“ markiert wurde. Ist dies der Fall, so wird angenommen, dass das Feature in dem betreffenden Release defekt ist. Tritt dieser Fall nicht ein, so gilt das Feature als fehlerfrei.
- im Fall von Dateien wird der jeweils letzte Commit der Dateien im betreffenden Release überprüft. Ist die Datei dort als „fehlerfrei“ markiert, so wird angenommen, dass sie fehlerfrei ist. Ist sie als „defekt“ markiert, so wird angenommen, dass sie in diesem Release defekt ist.

Die so erstellten einzelnen Tabellen werden anschließend zu jeweils einer gemeinsamen Tabelle konkateniert, sodass eine umfangreiche Auflistung an Metriken inklusive der zugehörigen Labels entsteht. Diese Auflistung gibt an, welche Charakteristika ein Feature beziehungsweise eine Datei aufweisen muss, um als „fehlerfrei“ oder „defekt“ eingestuft zu werden und dient als Trainingsgrundlage der Klassifikatoren für zukünftige Vorhersagen.

Eine Besonderheit stellt das zweite für den Vergleich im Rahmen der Evaluation erstellte dateibasierte Datenset dar (J). Es basiert auf dem „einfachen“ dateibasierten Datenset (I) mit den Metriken aus [18], umfasst aber zusätzlich die elf Metriken des featurebasierten Datensets (G) aus Tabelle 3.5. Dazu wurden die Werte der featurebasierten Metriken auf Dateiebene „gemapped“ beziehungsweise übertragen. Dies bedeutetet, dass für jede im dateibasierte Datenset referenzierte Datei eines Releases analysiert wurde, welche Features in der jeweiligen Datei erwähnt wurde. Von diesen Features wurden dann die zugehörigen Metriken des featurebasierten Datensets ermittelt und der Durchschnittswert berechnet um im erteiterten Datenset eingetragen. Sollte in einer Datei keine Features erwähnt worden sein, so wird für die featurebasierten Metriken jeweils der Wert 0 gespeichert. Wie zu Beginn des Abschnitts erwähnt wurde, kann auf diesem Weg das featurebasierte Datenset mit einem klassischen dateibasierten Datenset verglichen werden, da die Bedingungen für den Vergleich geschaffen wurden. Ein direkter Vergleich zwischen den verschiedenen Datensets ist aus den genannten Gründen nicht praktikabel.

Einige ergänzende Kennzahlen zu den Datensets sind in Tabelle 3.6 aufgeführt. Die Zeile „unique“ gibt dabei an, wie viele einzigartige Zeilen in den Datensets enthalten sind. Die auf die zuvor beschriebene Weise erstellten Datensets können nun zum Training der Klassifikatoren verwendet werden. Dieser Vorgang wird im kommenden Kapitel erläutert.

Tabelle 3.6: Kennzahlen der Datensets

	featurebasiertes Datenset	**einfaches dateibasiertes Datenset**	**erweitertes dateibasiertes Datenset**
Anzahl Attribute	11 + Label	17 + Label	28 + Label
Anzahl Datensätze	14.059	76.986	76.986
- davon defekt	2.735	1.899	1.899
- davon fehlerfrei	11.324	75.087	75.087
- davon „unique“	8.447	52.564	52.783

RQ1b: WIE WEIT MÜSSEN DIE DATEN VORVERARBEITET WERDEN, UM SIE FÜR DAS TRAINING NUTZBAR ZU MACHEN?

Die Verarbeitungsschritte für die Daten, die aus den Repositories erhalten wurden, bestehen aus: Extraktion der Features inklusive Bereinigung, Identifizierung von korrektiven Commits mittels Analyse der Commit-Nachrichten, Analyse nach fehlereinführenden Commits unter Zuhilfenahme des SZZ-Algorithmus sowie Berechnung von elf Metriken, welche als Attribute für das Training der Klassifikatoren dienen.

Kapitel 4

Training der Machine-Learning-Klassifikatoren

Dieses Kapitel gibt einen Überblick über das Training der Machine-Learning-Klassifikatoren. Dazu werden zunächst die Auswahl des verwendeten Werkzeugs und die Auswahl der Klassifikationsalgorithmen erläutert. Anschließend finde eine Darlegung des Trainingsprozesses statt.

4.1 Auswahl des Werkzeugs und der Klassifikationsalgorithmen

Obwohl im Rahmen der Erstellung der Datensets oft auf das Programmieren mit der Programmiersprache Python zurückgegriffen wurde, fiel die Wahl eines Machine-Learning-Werkzeugs nicht auf die Library scikit-learn [21], sondern auf eine eigenständige Lösung. Zur Anwendung kommt die WEKA-Workbench[1] als Machine-Learning-Werkzeug. Im Rahmen der strukturierten Literaturanalyse zu Beginn der Erarbeitung der Masterarbeit, erwies sich dieses Werkzeug aufgrund zahlreicher Zitierungen in wissenschaftlichen Arbeiten (unter anderem in [11, 24, 28]) als geeignet für die zugrundeliegende Aufgabe. Die WEKA-Workbench (WEKA als Akronym für **W**aikato **E**nvironment for **K**nowledge **A**nalysis) wurde an der University of Waikato in Neuseeland entwickelt und bietet eine große Kollektion an Machine-Learning-Algorithmen und Preprocessing-Tools zur Verwendung innerhalb einer grafischen Benutzeroberfläche [9]. Es existieren zudem Schnittstellen für die Programmiersprache Java [9].

Eine Übersicht über die ausgewählten Klassifikationsalgorithmen befindet sich in Tabelle 4.1. Diese umfasst auch die Abkürzungen der Klassifikatoren, die im weiteren Verlauf der Arbeit verwendet werden. Ausführliche Erläuterungen zu den Algorithmen wurden bereits zuvor in Abschnitt 2.1 gegeben.

Alle zuvor vorgestellten Klassifikationsalgorithmen sind bereits im Werkzeug WEKA integriert. Es erhält als Eingabe die finalen Datensets in einem proprietären Dateiformat, deren Erstellung im vorherigen Kapitel erläutert wurde. Die 13 berechneten Metriken bilden dabei die Attribute, wohingegen die Zielklasse durch die Label „defekt“ und „fehlerfrei“ abgebildet wird.

[1] https://www.cs.waikato.ac.nz/ml/weka/

Tabelle 4.1: Zum Training verwendete Klassifikationsalgorithmen

Algorithmus	**Abkürzung**
J48 Decision Trees	J48
k-Nearest-Neighbors	KNN
logistische Regression	LR
Naïve Bayes	NB
künstliche neuronale Netze	NN
Random Forest	RF
Stochastic Gradient Descent	SGD
Support Vector Machines	SVM

RQ2: WELCHE MACHINE-LEARNING-KLASSIFIKATOREN KOMMEN FÜR DIE GEGEBENE AUFGABE IN FRAGE?

Es werden acht verschiedene Klassifikationsalgorithmen zur Anwendung kommen, deren Hauptkriterium für die Auswahl die vorherige Verwendung im Rahmen der wissenschaftlichen Literatur war [32]. Die Algorithmen lauten: J48, KNN, LR, NB, NN, RF, SGD, SVM. Als Werkzeug für die Durchführung des Trainings der Klassifikatoren kommt der WEKA Workbench [9] zum Einsatz.

4.2 Trainingsprozess der Klassifikatoren

Der Trainingsprozess der Klassifikatoren erfolgte innerhalb der grafischen Benutzeroberfläche von WEKA. Vor dem Training wurden die Split-Ratios für die Aufteilung der Datensets in Trainingsdaten und Testdaten festgelegt. Sie wurde für jedes der verwendeten Softwareprojekte individuell festgelegt und basiert auf der Anzahl der jeweils vorhandenen Releases. Dabei wurde jedoch stets darauf geachtet, dass die üblich verwendeten Split-Ratios von 80:20 und 75:25 sowie 70:30 angenähert wurden. Die Größenordnungen der Trainingsdaten reichen von 67% bis 80% der Datensätze der Datensets. Die früheren Releases wurden den Trainingsdaten zugeordnet. Die Trainingsdaten enthalten die Datensätze der späteren Releases. Eine Übersicht der Aufteilung in Trainings- und Testdaten je Softwareprojekt ist in Abbildung 4.1 dargestellt. Die daraus resultierten Split-Ratios für die gesamten Datensets lauten: 69:31 (featurebasiertes Datenset) und 71:29 („einfache“ und erweiterte dateibasierte Datenset).

Das Training der Klassifikatoren in WEKA erfolgte für jeden Klassifikationsalgorithmus mit den jeweiligen Standardeinstellungen. Lediglich für die Algorithmen NN und RF wurden weitere Konfigurationen vorgenommen. Für den RF-Algorithmus wurde eine Instanzanzahl von 200 festgelegt. Dies bedeutet, dass gleichzeitig 200 Entscheidungsbäume parallel Verarbeitungen durchführen. Es gibt keine eindeutigen Empfehlungen, wie viele Instanzen gewählt werden sollten. Der ausgewählte Wert von 200 wurde somit unter Berücksichtigung des Umfangs der Datensets sowie der hohen Anzahl an Attributen eigenständig festgelegt. Für den NN-Algorithmus wurde eine eigenständig festgelegte Hidden-Layer-Struktur von `(13,13,13` gewählt. Dies bedeutet, dass das künstliche neuronale Netz drei Hidden-Layer-Schichten à 13 Hidden-Layer-Neuronen besitzt. Dies ermöglicht ihnen, effizienter die große Anzahl an Attributen zu verarbeiten. Darüber hinaus wurden keine Validierungsdaten erzeugt, da es nicht vorgesehen ist, eine Attributsauswahl durchzuführen oder weitere Einstellungen der Klassifikatoren anzupassen.

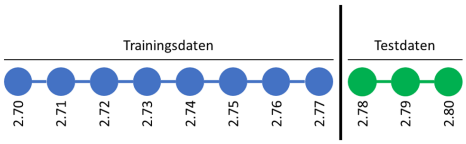
Eine Analyse des Trainingsprozesses zeigte zudem, dass die dateibasierten Datensets hinsichtlich der Zielklasse stark unbalanciert sind. Mit einem Wert von etwa 97% existieren weitaus

Tabelle 4.2: Kennzahlen der Datensets

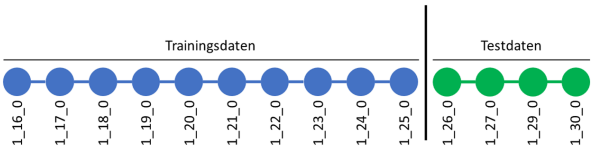
	„einfaches“ dateibasiertes Datenset		**erweitertes dateibasiertes Datenset**	
	vorher	**nachher**	**vorher**	**nachher**
Anzahl Attribute	17 + Label	17 + Label	28 + Label	28 + Label
Anzahl Datensätze	76.986	111.706	76.986	112.706
- davon defekt	1.899	37.619	1.899	37.619
- davon fehlerfrei	75.087	75.087	75.087	75.087
- davon „unique“	52.564	86.155	52.783	86.381
Gesamt-Split-Ratio	71:29	81:19	71:29	81:19

mehr Einträge, die dem Label „fehlerfrei“ zugeordnet sind. Balanciertheit, also ein ausgeglichenes Verhältnis (50:50 ist im binären Fall nicht zwingend notwendig) innerhalb der Zielklassen, ist jedoch eine Voraussetzung für das korrekte Training der meisten Klassifikatoren. Eine Nichtbeachtung dieses Problem kann zu einer irreführenden Accuracy (Treffergenauigkeit des Klassifikators) führen, da die meisten Datensätze korrekt der überrepräsentierten Klasse zugeordnet werden und ein grundsätzliches Problem der Accuracy-Metrik darstellt. Als Lösung dieses Problems wurde der sogenannte SMOTE-Algorithmus auf die dateibasierten Datensets angewendet [8]. Der Algorithmus, dessen Akronym für **S**ynthetic **M**inority **O**ver-sampling **Te**chnique steht, führt ein Oversampling der unterrepräsentierten Klasse durch [8]. Anhand von nächste-Nachbarn-Berechnungen auf Basis der Euklidischen Distanz zwischen den Attributwerten der einzelnen Datensätze der Datensets, werden neue synthetische Datensätze hinzugefügt (Oversampling), sodass sich die Anzahl der Datensätze der relevanten Klasse erhöht [8]. Im hier durchgeführten Fall wurde der Prozentsatz für die Generierung der synthetischen Datensätze auf 2000 festgelegt, sodass für jeden vorhandenen Datensatz der unterrepräsentierten Klasse 20 zusätzliche synthetische Datensätze erzeugt wurden. So konnte der Anteil der Datensätze mit dem Label „fehlerhaft“ auf etwa 41% erhöht werden. Gleichzeitig wuchsen die Datensätze um etwa 60%. Angewendet wurde der Algorithmus nur auf die Trainingsdaten [8] . Eine Anwendung auf die Testdaten ist nicht vorhergesehen, damit die „ground truth“ nicht verzerrt beziehungsweise verfälscht wird. Infolge dessen änderten sich zudem die Kennzahlen der dateibasierten Datensets. Diese sind in Tabelle 4.2 dargestellt.

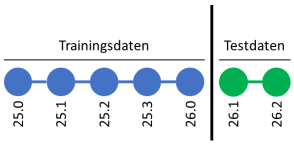
Die Ergebnisse, die mithilfe der Testdaten erzielt wurden und die Performanz der einzelnen Klassifikatoren widerspiegeln, werden im folgenden Kapitel im Rahmen der Evaluation vorgestellt.



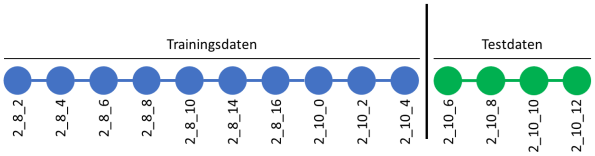
(a) Blender
Split-Ratio: 73:27



(b) Busybox
Split-Ratio: 71:29



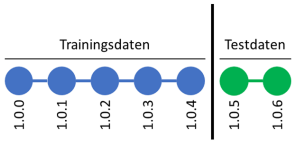
(c) Emacs
Split-Ratio: 71:29



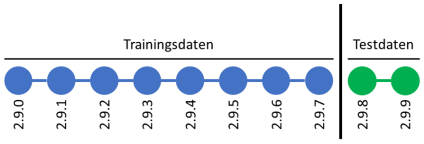
(d) GIMP
Split-Ratio: 71:29

(e) Gnumeric
Split-Ratio: 75:25

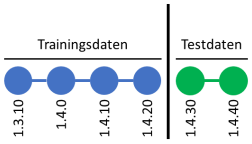
(f) gnuplot
Split-Ratio: 80:20



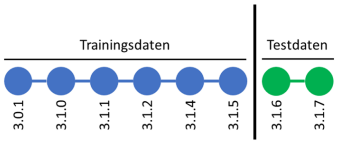
(g) Irssi
Split-Ratio: 71:29



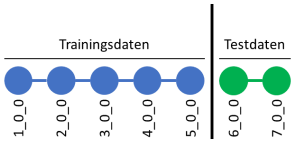
(h) libxml2
Split-Ratio: 80:20



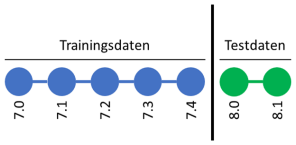
(i) lighttpd
Split-Ratio: 67:33



(j) MPSolve
Split-Ratio: 75:25



(k) Parrot
Split-Ratio: 71:29



(l) Vim
Split-Ratio: 71:29

Abbildung 4.1: Übersicht der Aufteilung in Trainings- und Testdaten

Kapitel 5

Evaluation

Dieses Kapitel dient der Evaluation der im vorangegangenen Kapitel erläuterten Klassifikatoren. Dies geschieht durch verschiedene Evaluationsmetriken, welche in diesem Kapitel vorgestellt werden und auf Werten von sogenannten Konfusionsmatrizen basieren. Zudem umfasst dieses Kapitel eine Erläuterung von Herausforderungen und Limitationen, die im Laufe der Bearbeitung der Arbeit festgestellt worden sind. Abgeschlossen wird dieses Kapitel mit dem Vergleich zwischen dem „einfachen“ und erweiterten dateibasierten Datenset zur Messung der Einflüsse der featurebasierten Metriken auf die Performanz der Vorhersagen.

5.1 Herausforderungen und Limitationen

In diesem Abschnitt werden eine Reihe von Herausforderungen und Limitationen aufgezählt und erläutert, die im Rahmen der Erarbeitung der Forschungsziele auffällig wurden.

Identifikation von Features

Die grundsätzliche Frage, die im Rahmen der Identifikation der Features aufkam, war: „Was wird als Feature gezählt?“. Wie bereits in Abschnitt 3.2 erwähnt wurde, barg die Identifikation der Features einige Herausforderungen. So gestaltete sich die erste Herausforderung in der Ausfilterung von „Header-Features“, die in einigen Programmierparadigmen verwendet werden, um Header-Dateien im Sourcecode einzubinden. Diese Header-Features erzeugen jedoch keine Variabilität im Code, sodass sie unerwünscht sind. Identifizierbar waren die meisten dieser Header-Features an ihren vergebenen Namen, welche ein `_h_` aufwiesen. Auf diesem Weg konnten sie mittels regulärer Ausdrücke schnell ermittelt und ausgefiltert werden. Es besteht jedoch auch die Möglichkeit, dass in einigen Softwareprojekten die Header-Features nicht explizit durch ihre Namensgebung kenntlich gemacht werden. Sie lassen sich somit nur schwer identifizieren, beispielsweise durch eine manuelle Sichtung der Kontexte der Features im Sourcecode. Dies wäre jedoch im vorliegenden Fall aufgrund der großen Menge an Features sehr zeitaufwändig und wurde aus diesem Grund nicht durchgeführt. Die Entfernung der erkennbaren Header-Features zeigte, dass ein erheblicher Teil der zuvor identifizierten Features unerwünscht war. Diese Methode erwies sich somit als effektiv. Eine Lösung, die zu dem genannten Problem beitragen könnte, wäre ein Tool zum Parsen von Sourcecode zur korrekten Identifikation von Features mittels automatisierter Analyse des Kontextes des Features. So würden

auch die erwähnten „falschen“ Features ermittelt werden. Ein solches Tool existiert momentan jedoch nicht. Verfügbare Tools zur Identifikation von Features verwenden einen ähnlichen Ansatz, wie er in dieser Arbeit verwendet wurde (reguläre Ausdrücke).

Einbindung des Bezugs zu Features

Wie in Kapitel 3 festgestellt werden konnte, basiert der Bezug zu den Features auf den ihnen zugrundeliegenden Dateien. Dazu wurden die Diffs der veränderten Dateien analysiert. Ein Feature gilt somit als relevant, wenn es in einem Diff Erwähnung findet. Es kann jedoch auch vorkommen, dass der enthaltene Featurecode nicht an der im Diff beschriebenen Veränderung beteiligt war, sondern nur im als „Hunk“ bezeichneten Teil des Diffs erwähnt wurde. Dieser bezeichnet einen Überhang des eigenliches Kontexts der beschriebenen Veränderungen in Form von weiteren Codezeilen, die dieser Veränderung folgen. Es findet somit keine „in-depth“-Analyse des Sourcecodes statt. Dieser Weg wurde jedoch auch von der dieser Arbeit zugrundeliegenden wissenschaftlichen Arbeit gewählt [24]. Ebenfalls werden die Metriken der Features auf Basis der Metadaten der zugrundeliegenden Dateien berechnet. Es findet somit eine Überapproximierung statt.

Heuristik zur Erkennung von korrektiven Commits

Die Heuristik zur Erkennung von korrektiven Commits wurde im Verlauf der Erarbeitung der Arbeit geändert. Zunächst wurde die gesamte Commit-Nachricht auf das Vorhandensein der Schlagworte analysiert. Dies führte jedoch dazu, dass einige Commits fälschlicherweise als korrektiv identifiziert wurden. Der Grund dafür war, dass die Commit-Nachrichten in einigen der verwendeten Softwareprojekte sehr umfangreich in der Anzahl der Wörter waren. In diesen Nachrichten wurden ausnahmslos alle Veränderungen, die mit den Commits vorgenommen wurden, erwähnt. Dabei handelte es sich jedoch meist um für diesen Zweck irrelevante Veränderungen. Es wurde festgestellt, dass sich die Hauptaussage beziehungsweise der Hauptgrund des Commits in der ersten Zeile der Commit-Nachricht befindet. Die Heuristik wurde daraufhin dementsprechend angepasst. Eine Stichprobe von korrektiven Commits vor und nach der Anpassung der Heuristik zeigte, dass die Veränderung dazu führte, dass tatsächlich irrelevante korrektive Commits entfernt wurden.

Unpräzisiertheit des SZZ-Algorithmus

Eine Limitation, die im Laufe der Erstellung des Datensets durch Literaturrecherchen festgestellt wurde, bezieht sich auf den SZZ-Algorithmus. Dieser wurde genutzt, um fehlereinführende Commits auf Basis der Commit-Hashes der korrektiven Commits zu identifizieren. Analysen des Algorithmus ergaben, dass momentan verfügbare Implementationen und somit auch die des verwendeten Python-Tools PyDriller lediglich etwa 69% der tatsächlich existierenden fehlereinführenden Commits identifizieren können [38]. Darüber hinaus wurde herausgefunden, dass etwa 64% der identifizierten Commits falsch ermittelt wurden [38]. Der Algorithmus gilt somit als unpräzise [38]. Die Begründung dafür lautet wie folgt:

> The reason is that the implicit assumptions of the SZZ algorithm are violated by the insufficient file coverage and statement direct coverage between bug-inducing and bug-fixing commits. - [38]

Der Grund dafür ist, dass die impliziten Annahmen des SZZ-Algorithmus durch die unzureichende „file coverage“ und die unzureichende „statement direct coverage“ der Aussage zwischen fehlereinführenden und korrektiven Commits verletzt werden.

Ferner stellten die Autoren der Studie in eigenen durchgeführten Tests fest, dass die Ergebnisse von acht von zehn früheren Studien durch den unpräzisen Algorithmus signifikant beeinflusst wurden [38]. Dies kann somit auch auf diese Arbeit zutreffen. Es existiert jedoch momentan keine alternative Methode zur Identifizierung von fehlereinführenden Commits. Sollte eine neue Methode oder eine verbesserte Version des SZZ-Algorithmus veröffentlicht werden, so würde es sich anbieten, die Hauptschritte dieser Arbeit unter Berücksichtigung der neuen Methode zu wiederholen, um sie mit den hier vorliegenden Ergebnissen zu vergleichen, um die Einflüsse des SZZ-Algorithmus herauszustellen.

Unzureichende Metadaten von xfig

Im Rahmen der Identifizierung der korrektiven Commits wurde festgestellt, dass für das Softwareprojekt xfig keine Ergebnisse erzielt werden konnte. Folglich konnten somit auch keine fehlereinführenden Commits festgestellt werden. Zu sehen ist dies auch in Tabelle 3.3. Der Grund für die fehlende Identifizierung der korrektiven Commits ist, dass die Commit-Nachrichten des Softwareprojekts keine der festgelegten Schlagworte enthalten. Sie bestehen lediglich aus der Angabe der mit dem Commit freigegebenen Releasenummer. Beispiele dafür sind:

- Commit `af30126616c5c5a8db3ba017dedbcbdf48fbc528`
  Nachricht: `xfig-3.2.7b`
- Commit `a444a8ae7995dbfd2ebce4696ed2cca7ad33b6e1`
  Nachricht: `xfig-3.2.6.tar.xz`
- Commit `f3706bcafe9049247eee1a88d64f9f8b4e98c076`
  Nachricht: `xfig.3.0.tar.gz`

Der Umstand, dass weder korrektive noch fehlereinführende Commits identifiziert wurden, führt dazu, dass jedes Feature und jede Datei automatisch und möglicherweise fälschlicherweise als „fehlerfrei“ eingestuft werden. In Anbetracht dieser Tatsache wurde entschieden, xfig bei der Zusammenstellung der finalen Datensets nicht mit einzubinden.

Vorhersageziel

Die Festlegung des Vorhersageziels war bis zum Ende der sechsmonatigen Bearbeitungszeit dieser Arbeit stets ein Diskussionsthema zwischen den beteiligten Personen. Wie im Kapitel zur Erstellung der Datensets bereits gezeigt wurde, sind die Werte der Metriken der Features und Dateien für jedes Datenset nach Releases in Form des Mittelwertes aggregiert. Somit läge es auch nahe, zukünftige Releases als den Input neuer Daten zur Vorhersage zu verwenden. Es werden dann also defekte Features oder Dateien in einem Release vorhergesagt. Eine weitere Möglichkeit liegt in der Betrachtung von Commits. Dazu müssen jedoch dann die Metriken von einem Release-Level auf ein Commit-Level abstrahiert werden.

Wie ist es in der wissenschaftl. Literatur?
Finale Festlegung?
... yet to be determined ...

5.2 Evaluationsmetriken

Die zum Vergleich der Klassifikatoren erhobenen Evaluationmetriken entstammen dem Themengebiet des Information Retrieval und gelten als Standardmesswerte für ihren Einsatzzweck [30]. Die meisten dieser Metriken lassen sich anhand von Werten einer sogenannten Konfusionsmatrix berechnen und messen allesamt die Performanz der Vorhersagen von Klassifikatoren unter verschiedenen Betrachtungsweisen. Im Falle einer binären Klassifikation, wie in dieser Arbeit, besteht diese Matrix aus vier Gruppen, deren Werte angeben, ob der jeweilige Klassifikator ein Objekt korrekt oder falsch einer der beiden Zielklassen zuordnen konnte [30]. Im Zusammenhang mit solchen Matrizen werden die beiden Zielklassen „positiv“ und „negativ“ genannt. Für diese Arbeit werden die positive Klasse dem Label „defekt“und die negative Klasse dem Label „fehlerfrei“ zugeordnet. Die Form einer allgemeinen Konfusionsmatrix ist in Abbildung 5.1 dargestellt.

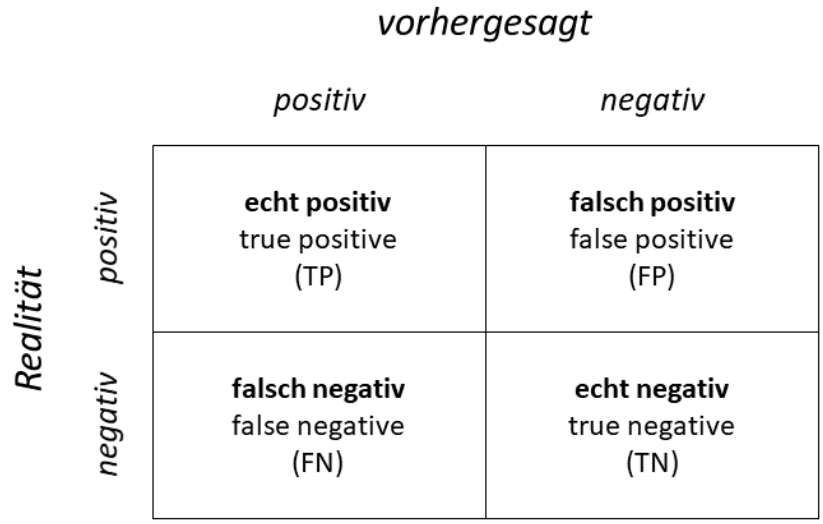


Abbildung 5.1: allgemeine Konfusionsmatrix

Das zum Training der Klassifikatoren verwendete Werkzeug WEKA besitzt die Option, Konfusionsmatrizen zu den durchgeführten Tests der Klassifikatoren auszugeben. Anhand der Werte der Zuordnungen zu den zuvor genannten Gruppen wurden die folgenden Evaluationsmetriken berechnet:

- Treffergenauigkeit (Accuracy)
  Dieser Wert misst die Treffergenauigkeit der Vorhersagen des Klassifikators und gibt an, inwieweit dessen Vorhersagen mit der modellierten Realität übereinstimmen [30]. Die Formel zur Berechnung der Accuracy lautet:

  $$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

  Das Ergebnis der Berechnung ist ein prozentualer Wert. 100% stellen damit die bestmögliche Accuracy dar.

- Echt-Positiv-Rate / Trefferquote (TP-Rate / Recall)
  Dieser Wert gibt den Anteil der korrekt als positiv gewerteten Vorhersagen sämtlicher als positiv gewerteter Vorhersagen an [2]. Die Formel zur Berechnung der TP-Rate bzw. des

Recalls lautet:

$$TP - Rate = \frac{TP}{TP + FN}$$

Das Ergebnis der Berechnung ist ein prozentualer Wert. 100% stellen damit die bestmögliche TP-Rate bzw. den bestmöglichen Recall dar. Beide Begriffe werden parallel für die Berechnung der gezeigten Formel verwendet.

- Positiver Vorhersagewert (Precision)
  Dieser Wert gibt die Anzahl der positiven Vorhersagen an, die auch tatsächlich zur positiven Klasse gehören [30]. Die Formel zur Bestimmung der Precision lautet:

  $$Precision = \frac{TP}{TP + FP}$$

  Das Ergebnis der Berechnung ist ein prozentualer Wert. 100% stellen damit die bestmögliche Precision dar.

- F-Maß (F-Score)
  Dieser Wert berechnet das harmonische Mittel der Werte Precision und Recall und liegt somit zwischen diesen beiden Werten, jedoch näher am kleineren Wert [30]. Die Formel zur Berechnung des F-Scores lautet:

  $$F - Score = \frac{2TP}{2TP + FP + FN}$$

  Das Ergebnis der Berechnung ist ein prozentualer Wert. 100% stellen damit den bestmöglichen F-Score dar.

Darüber hinaus wurden die sogenannten ROC-Kurven (ROC curve) der einzelnen Klassifikatoren ermittelt. Diese Wahrscheinlichkeitskurven bzw. -graphen (ROC = Receiver Operating characteristic, Betriebsverhalten des Empfängers), beschreiben das Verhältnis zwischen der TP-Rate (y-Achse) und der FP-Rate (x-Achse) [30, 19]. Die Falsch-Positiv-Rate (FP-Rate) gibt dabei den Anteil der fälschlicherweise als positiv gewerteten Vorhersagen an [2]. Sie wird mittels der folgenden Formel berechnet:

$$FP - Rate = \frac{FP}{FP + TN}$$

Wie bei allen Metriken ist das Ergebnis der FP-Rate ein prozentualer Wert, der bestenfalls möglichst gering ausfallen sollte. Sowohl die TP-Rate als auch die FP-Rate geben nur singuläre Werte an, aus welchen sich keine Graphen herleiten lassen. Jeder Klassifikator errechnet jedoch im Rahmen der Vorhersage eines Datensatzes Wahrscheinlichkeiten, die die Zugehörigkeit zu den Werten der Zielklasse darstellen [13]. In der Regel wird ein Datenpunkt der positiven Klasse zugeordnet, wenn die Wahrscheinlichkeit einen Schwellenwert von 0,5 übersteigt - Datenpunkte, die diesen Schwellenwert unterschreiten werden wiederum der negativen Klasse zugeordnet [13]. Wird der Schwellenwert erhöht, so werden weniger Datenpunkte der positiven Klasse zugeordnet, wohingegen im Falle einer Absenkung des Schwellenwertes mehr Datenpunkte der positiven Klasse zugeordnet werden [13]. Für die Erstellung der ROC-Kurven werden somit die Werte der TP-Rate und der FP-Rate unter der Berücksichtigung der Schwellenwerte im Bereich von 0,0 bis 1,0 gegenübergestellt.

Eine weitere Metrik, die in Verbindung mit der ROC-Kurve auftritt, ist der ROC-Bereich (ROC area). Dieser Wert, der anhand der ROC-Kurve berechnet wird und auch AUC-Bereich (AUC = Area Under Curve, Bereich unter der Kurve) genannt wird, gibt an, in wieweit ein Klassifikator

in der Lage ist, zwischen den Werten der Zielklassen zu unterscheiden [19]. Je höher dieser Wert ist (1,0 ist das Maximum), desto besser trifft der Klassifikator korrekte Vorhersagen [19].

Am Beispiel des Schwellenwertes 0,5 wird nun die Interpretation der ROC-Kurve und des ROC-Bereiches vorgenommen. Unterstützt wird dies durch grafische Beispiele, die in Abbildung 5.2 gezeigt werden. Der Idealfall ist in den (a) und (b) dargestellt. Die Wahrscheinlichkeitskurven der Zielklasse (a) weisen keine Überlappung auf. Die Werte der Zielklasse sind somit allesamt korrekt zugeordnet worden, sodass der Klassifikator korrekt zwischen diesen unterscheiden kann. Die diesem Fall entsprechende ROC-Kurve ist in (b) dargestellt. Der ROC-Bereich beträt in diesem Fall 1,0. Ein „Normalfall" ist in (c) und (d) dargestellt. Es ist in (c) zu erkennen, dass die Wahrscheinlichkeitskurven überlappen, sodass falsche Zuordnungen getroffen werden. Die entsprechende ROC-Kurve ist in (d) dargestellt. Der entsprechende ROC-Bereich beträgt im hier gezeigten Fall 0,7. Dies bedeutet, dass 70% der Zuordnungen richtig getroffen werden. Verbessert werden können die Werte möglicherweise, wenn der Schwellenwert verändert wird. Der „Worst Case", der bei der Performanzmessung mittels ROC-Kurven auftreten kann, ist in (e) und (f) dargestellt. In (e) ist zu erkennen, dass sich die Wahrscheinlichkeitskurven vollständig überlappen. Es findet somit eine willkürliche Zuordnung statt. Die zugehörige ROC-Kruve (f) entspricht einer Winkelhalbierenden. Ein weiterer Fehlerfall ist in (g) und (h) dargestellt. Die Wahrscheinlichkeitskurven (g) zeigen, dass die Zuordnungen gegenteilig erfolgen und somit der jeweils dem anderen Wert der Zielklasse falsch zugeordnet werden. Der entsprechende ROC-Bereich beträgt 0, da wie in (h) zu sehen ist, keine Fläche unter der Kurve vorhanden ist.

Alle vorgestellten Metriken werden automatisiert von WEKA berechnet. Ferner besitzt das Werkzeug die Fähigkeit, ROC-Kurven mit den entsprechenden ROC-Werten auszugeben. Die im Rahmen der Evaluation der Klassifikatoren ermittelten Werte der Metriken sowie die ROC-Kurven und Werte der ROC-Bereiche werden im folgenden Abschnitt aufgeführt sowie verglichen und interpretiert.

RQ3a: WELCHE MITEINANDER VERGLEICHBAREN MERKMALE BESITZEN DIE KLASSIFIKATOREN?

Die Merkmale zum Vergleich stellen Evaluationsmetriken dar, welche auf Basis der Ergebnisse des Tests der Klassifikatoren errechnet werden und die Performanz der Vorhersagen auf verschiedene Weisen messen. Welche Metriken berechnet wurden beantwortet Forschungsfrage RQ3b.

RQ3b: WELCHE METRIKEN KÖNNEN FÜR DEN VERGLEICH VERWENDET WERDEN?

Für den Vergleich der Klassifikatoren werden klassische Evaluationsmetriken verwendet, welche auf Basis von Konfusionsmatrizen berechnet werden. Die betrachteten Metriken lauten: Accuracy, TP-Rate / Recall, Precision und F-Score. Ebenfalls hinzugezogen werden die jeweiligen ROC-Kurven der Klassifikatoren inklusive der ROC-Bereiche. Diese Metriken stellen einen Standard für die Messung der Performanz der Vorhersagen von Klassifikatoren dar.

5.3 Ergebnisse und Diskussion

5.3.1 Featurebasiertes Datenset

Konfusionsmatrizen

Die Konfusionsmatrizen dienen als Basis der Ergebnisse der Evaluation der Klassifikatoren anhand der zuvor vorgestellten Metriken. Die Matrizen des featurebasierten Datensets sind in Tabelle 5.1 aufgeführt. Dabei bilden die Spalten die von den Klassifikatoren vorhergesagten Label ab. Die Zeilen bilden wiederum die „ground truth“, also die im Rahmen der Erstellung der Datensets ermittelte Realität, ab. Außerdem werden die ermittelten Werte der jeweiligen Klassen in den Spalten „total“zusammengezählt. Da für jeden Klassifikator das selbe Testset gewählt wurde, sind die Werte der „total“-Spalte identisch. Die für die Klassifikatoren verwendeten Abkürzungen können Tabelle 4.1 entnommen werden.

Tabelle 5.1: Konfusionsmatrizen des featurebasierten Datensets

	Ermittelt ->	**defekt**	**fehlerfrei**	**total**
J48	Realität defekt	446	320	766
	Realität fehlerfrei	483	3.142	3.625
	total	929	3.462	4.391
KNN	Realität defekt	371	395	766
	Realität fehlerfrei	569	3.056	3.625
	total	940	3.451	4.391
LR	Realität defekt	309	457	766
	Realität fehlerfrei	1.020	2.605	3.625
	total	1.329	3.061	4.391
NB	Realität defekt	322	444	766
	Realität fehlerfrei	265	3.360	3.625
	total	587	3.804	4.391
NN	Realität defekt	327	439	766
	Realität fehlerfrei	1.019	2.606	3.625
	total	1.346	3.045	4.391
RF	Realität defekt	504	262	766
	Realität fehlerfrei	450	3.175	3.625
	total	954	3.437	4.391
SGD	Realität defekt	251	515	766
	Realität fehlerfrei	938	2.687	3.625
	total	1.189	3.202	4.391
SVM	Realität defekt	176	590	766
	Realität fehlerfrei	899	2.726	3.625
	total	1.075	3.316	4.391

Accuracies

Die erste Metrik, die verglichen wird, ist die Accuracy. Diese gibt die Treffergenauigkeit der Klassifikatoren an. Dargestellt werden die Ergebnisse zum besseren Vergleich als Balkendiagramme. Das Diagramm des feturebasierten Datensets ist in Abbildung 5.3 dargestellt. Die konkreten Zahlenwerte können im Anhang eingesehen werden.

Die Ergebnisse zeigen, dass die Accuracies der Klassifikatoren zwischen 66% und 84% liegen. Die höchsten Werte erreichten die Klassifikatoren NB und RF mit einer Treffergenauigkeit

von jeweils 84%. Mit einem Wert von 82% erreichte auch der J48-Klassifikator eine überdurchschnittlich hohe Performanz. Darauf folgt der KNN-Klassifikator mit 78%. Mit Werten von 66% beziehungsweise 67% schnitten die Klassifikatoren LR, NN, SGD und SVM am schlechtesten ab. Sie ermitteln somit in über 30% der Vorhersagen ein falsches Ergebnis. Auffällig an diesen Ergebnissen ist die hohe Performanz beider Entscheidungsbaum-basierter Klassifikatoren J48 und RF. Sie scheinen somit die Eingabemenge am besten verarbeiten zu können, um Rückschlüsse auf die neuen Daten in Form der Testdaten schließen zu können.

Weitere Evaluationsmetriken

Dieser Abschnitt stellt die Ergebnisse der weiteren Evaluationsmetriken TP-Rate / Recall, FP-Rate, Precision und F-Score vor. Diese können in Tabelle 5.2 für das featurebasierte Datenset eingesehen werden. Aufgeteilt werden die Ergebnisse nach den Werten der Zielklasse „defekt“ und „fehlerfrei“. Zudem wird der Mittelwert der Ergebnisse beider Werte der Zielklasse angegeben. Er zeigt die Performanz aggregiert für beide Werte an und liegt im Idealfall bei 1,00. Die vollständigen Tabellen der Ergebnisse der Evaluation, inklusive einer weiteren Metrik, können im Anhang gefunden werden.

Tabelle 5.2: Ergebnisse der Evaluationsmetriken des featurebasierten Datensets

		defekt	**fehlerfrei**	**Mittel**
J48	TP-Rate / Recall	0,58	0,87	0,73
	FP-Rate	0,13	0,42	0,28
	Precision	0,48	0,91	0,70
	F-Score	0,53	0,89	0,71
KNN	TP-Rate / Recall	0,48	0,86	0,67
	FP-Rate	0,16	0,52	0,68
	Precision	0,40	0,89	0,65
	F-Score	0,44	0,86	0,66
LR	TP-Rate / Recall	0,40	0,72	0,56
	FP-Rate	0,28	0,60	0,44
	Precision	0,23	0,85	0,54
	F-Score	0,30	0,78	0,54
NB	TP-Rate / Recall	0,42	0,93	0,68
	FP-Rate	0,07	0,58	0,33
	Precision	0,55	0,88	0,72
	F-Score	0,48	0,91	0,70
NN	TP-Rate / Recall	0,43	0,72	0,58
	FP-Rate	0,28	0,57	0,43
	Precision	0,24	0,86	0,55
	F-Score	0,31	0,78	0,55
RF	TP-Rate / Recall	0,66	0,88	0,77
	FP-Rate	0,12	0,34	0,23
	Precision	0,53	0,92	0,73
	F-Score	0,59	0,90	0,75
SGD	TP-Rate / Recall	0,34	0,74	0,54
	FP-Rate	0,56	0,67	0,62
	Precision	0,21	0,84	0,53
	F-Score	0,26	0,79	0,53
SVM	TP-Rate / Recall	0,23	0,75	0,48
	FP-Rate	0,25	0,77	0,51
	Precision	0,16	0,82	0,49
	F-Score	0,19	0,79	0,49

Eine erste Betrachtung der Ergebnisse der Evaluationsmetriken zeigt, dass die Resultate (mit Ausnahme der FP-Rate) bezogen auf das Label „defekt“ schlechter ausfallen, als die des Labels „fehlerfrei“. Die im Vergleich besten Ergebnisse wurden vom RF-Klassifikator für das Label „defekt“ erreicht. 66% der tatsächlich defekten Datensätze wurden auch als solche vorhergesagt und lediglich 12% der Datensätze wurden hingegen fälschlicherweise als defekt vorhergesagt. Die Werte der Precision und des F-Scores liegen bei 53% respektive 59% auf einem durchschnittlichen Niveau. Vergleichbare Ergebnisse erzielte der J48-Klassifikator. Die insgesamt schlechtesten Werte weisen SGD und SVM auf. Ihre Genauigkeit der Erkennung der tatsächlich defekten Datensätze betragen nur 34% beziehungsweise 23%. Die weiteren Klassifikatoren operierten auf einem durchschnittlichen Level.

Die jeweiligen Mittelwerte können betrachtet werden, um die Gesamtperformanz der Klassifikatoren für beide Label zu vergleichen. Hier schneidet erneut der RF-Klassifikator am besten ab. Die Werte für die TP-Rate, die Precision und den F-Score liegen im Umfeld von über 73% und sind damit vergleichsweise überdurchschnittlich hoch. Die FP-Rate in Höhe von 23%, die hauptsächlich durch die Ergebnisse hinsichtlich des Labels „fehlerfrei“ geschuldet ist, liegt auf dem niedrigsten Niveau aller Klassifikatoren. Erneut sind die Ergebnisse des ebenfalls Entscheidungsbaum-basierten Klassifikators J48 auf einem ähnlichen Niveau. Die Klassifikatoren KNN und NB weisen durch ihre hohen Werte der FP-Rate insgesamt durchschnitte Performanz auf. In der Gesamtheit betrachtet zeigen die Werte der Klassifikatoren LR, NN, SGD und SVM, dass diese im Vergleich die schlechteste Performanz besitzen. Insbesondere die FP-Raten von über 43% sind auf einem hohen, nicht wünschenswerten Niveau.

ROC-Kurven und ROC-Bereiche

Die Interpretation der ROC-Kurven und ROC-Bereiche erfolgt anhand des in Abschnitt 5.2.1 vorgestellten Schemas. Die ROC-Kurven samt der Werte der ROC-Bereiche (repräsentiert durch „AUC“) sind in Abbildung 5.4 dargestellt. Zu sehen sind jeweils die von WEKA ausgegebenen und unveränderten Plots. Die dargestellten Farbverläufe der Kurven verdeutlichen keine für diesen Zweck relevanten Informationen und können somit ignoriert werden.

Es ist zu erkennen, dass keine ROC-Kurve den in Abschnitt 5.2.1 vorgestellten Idealfall annähert. Die beste Vorhersageperformanz zeigt erneut der RF-Klassifikator mit einem ROC-Bereich von 0,84. Der NB-Klassifikator zeigt eine minimal geringere Performanz mit einem ROC-Bereich von 0,80. Eine durchschnittliche Performanz von 0,77 beziehungsweise 0,72, bezogen auf den ROC-Bereich, weisen die Klassifikatoren J48 und KNN auf. Die Performanz der LR- und NN-Klassifikatoren liegen im unteren Durchschnitt. Den unerwünschten Fall einer winkelhalbierenden ROC-Kurve zeigen die SGD- und SVM-Klassifikatoren. Dies spiegeln auch die ROC-Bereiche wider. Sie „raten“ somit, statt präzise Vorhersagen zu treffen.

Zusammenfassung

In jeder der betrachteten Kategorien von Metriken erwiesen sich die Entscheidungsbaum-basierten Klassifikatoren J48 und RF als am performantesten im Vergleich zu den weiteren Klassifikatoren. Der RF-Klassifikator erreichte dabei meist zudem eine etwas höhere Performanz. Die den Klassifikatoren zugrundeliegenden Algorithmen scheinen das gegebene featurebasierten Datenset mit seinen elf Attributen am besten hinsichtlich der Entwicklung der Ableitungsfunktion zur Vorhersage neuer Daten am besten verarbeiten zu können. Die Klassifikatoren SGD

und SVM zeigten für jede Kategorie von Evaluationsmetriken stets die im Vergleich schlechtesten Performanzen. Es wurde zudem festgestellt, dass die Klassifikatoren raten, statt Vorhersagen zu treffen. Das featurebasierte Datenset scheint somit für diese Klassifikationsalgorithmen nicht geeignet zu sein. Gründe dafür können eine zu geringe Anzahl an Instanzen oder eine zu hohe Anzahl an Attributen sein.

5.3.2 Dateibasierter Vergleich

Dieser Abschnitt dient zur Evaluation und zum Vergleich der dateienbasierten Datensets, deren Erstellung in Abschnitt 3.3 erläutert wurde. Der Grund dieses Vergleiches ist die Messung der Einflüsse der featurebasierten Metriken auf die Vorhersagen einer klassischen dateibasierten Methode, die der wissenschaftlichen Literatur entnommen wurde [18]. Die Aufteilung dieses Abschnitts ist analog zum vorherigen Abschnitt.

Konfusionsmatrizen

Die Konfusionsmatrizen der dateibasierten Datensets sind in Tabelle 5.3 aufgeführt. Die übergeordneten Spalten sind dabei in in das „einfache“ und das erweiterte Datenset angeordnet. Beide Datensets umfassen die selbe Anzahl an Datensätzen, somit sind die „total“-Spalten identisch. Eine kurze Analyse der „defekt“- „Realität defekt“- Felder zeigt, dass nur sehr wenige tatsächlich defekten Datensätze auch wirklich als defekt vorhergesagt wurden. Dies wird sich auch in den weiteren Ergebnissen der Evaluationsmetriken widerspiegeln.

Tabelle 5.3: Konfusionsmatrizen der dateibasierten Datensets

		„einfaches“ dateibasiertes Datenset			**erweitertes dateibasiertes Datenset**		
	Ermittelt ->	**defekt**	**fehlerfrei**	**total**	**defekt**	**fehlerfrei**	**total**
J48	Realität defekt	11	102	113	15	98	113
	Realität fehlerfrei	911	20.923	21.834	1.120	20.714	21.834
	total	922	21.025	21.947	1.135	2.0812	21.947
KNN	Realität defekt	29	84	113	20	93	113
	Realität fehlerfrei	5.209	16.625	21.834	4.861	16.973	21.834
	total	5.238	16.709	21.947	4.881	17.066	21.947
LR	Realität defekt	36	77	113	32	81	113
	Realität fehlerfrei	2.520	19.314	21.834	2.578	19.256	21.834
	total	2.556	19.391	21.947	2.610	19.337	21.947
NB	Realität defekt	77	36	113	83	30	113
	Realität fehlerfrei	18.700	3.134	21.834	18.750	3.084	21.834
	total	18.777	3.170	21.947	18.833	3.114	21.947
NN	Realität defekt	6	107	113	1	112	113
	Realität fehlerfrei	4.341	17.493	21.834	141	21.693	21.834
	total	4.347	17.600	21.947	142	21.805	21.947
RF	Realität defekt	7	106	113	4	109	113
	Realität fehlerfrei	382	21.452	21.834	366	21.468	21.834
	total	389	21.558	21.947	370	21.577	21.947
SGD	Realität defekt	22	91	113	20	93	113
	Realität fehlerfrei	1.260	20.574	21.834	1.226	20.608	21.834
	total	1.282	20.665	21.947	1.246	20.701	21.947
SVM	Realität defekt	22	91	113	21	92	113
	Realität fehlerfrei	1.041	20.793	21.834	991	20.843	21.834
	total	1.063	20.884	21.947	1.012	20.935	21.947

Accuracies

Die Accuracies der Klassifikatoren der dateibasierten Datensets sind in Abbildung 5.5 dargestellt. Die konkreten Zahlenwerte können dem Anhang entnommen werden.

Der Vergleich zwischen den Datensets zeigt, dass die Einbindung der featurebasierten Metriken für sechs von acht Klassifikatoren keinen signifikanten Einfluss auf die Performanz der Vorhersagen hat. Die erzielten Werte der Accuracies sind dort jeweils auf dem selben Niveau. Die im Vergleich schlechteste Accuracy erzielt der NB-Klassifikator mit etwa 15% Treffergenauigkeit. Dieser scheint die hohe Anzahl an Datensätzen nicht korrekt verarbeiten zu können. Die Klassifikatoren KNN und LR erreichten mit etwa 76% beziehungsweise 87% überdurchschnittlich hohe Ergebnisse. Mit Werten von über 90% besitzen die Klassifikatoren J48, RF, SGD und SVM äußerst hohe Werte, deren Aussagekraft zunächst infrage gestellt werden und die weiteren Evaluationsmetriken betrachtet werden sollten. Eine Auffälligkeit zeigt der NN-Klassifikator. Es ist ein deutlicher Sprung der Werte von etwa 18% zwischen dem „einfachen“ und dem erweiterten Datenset zu erkennen. Diese Auffälligkeit sollte ebenfalls anhand der weiteren Evaluationsmetriken analysiert und interpretiert werden. Sie kann einerseits verdeutlichen, dass der Klassifikator des erweiterten Datensets durch die Hinzunahme der zusätzlichen Attribute genauere Vorhersagen treffen kann oder durch diesen Umstand die Vorhersagen negativ beeinflusst werden, sodass die Ergebnisse verzerrt werden.

Weitere Evaluationsmetriken

Dieser Abschnitt stellt die Ergebnisse der weiteren Evaluationsmetriken der dateibasierten Datensets in Tabelle 5.4 vor. Die übergeordneten Spalten werden erneut durch die beiden Datensets repräsentiert. Die vollständigen Tabellen der Ergebnisse der Evaluation, inklusive einer weiteren Metrik, können im Anhang gefunden werden.

Die Ergebnisse der weiteren Evaluationsmetriken beider Datensets zeigt erneut den Trend, dass nur wenige defekte Datensätze auch tatsächlich als defekt vorhergesagt wurden. Mit TP-Raten von 68% beziehungsweise 74% erreichten die NB-Klassifikatoren die höchsten Trefferquoten. Die weiteren Klassifikatoren liegen mit Werten von höchstens 32% deutlich darunter. Die weiteren Metriken der „defekt“-Spalten repräsentieren ebenfalls äußerst schwache Ergebnisse. Auffällig sind die sehr hohen FP-Raten der NB-Klassifikatoren, die die guten Ergebnisse der TP-Raten deutlich abschwächen.

Die Werte der „fehlerfrei“-Spalten zeigen hingegen sehr gute Werte. Lediglich die jeweiligen FP-Raten mit über 68% sind deutlich zu hoch und zeigen, dass die Klassifikatoren viele Datensätze zu Unrecht als „fehlerfrei“ vorhersagen. Eine Ausnahme davon bilden erneut die NB-Klassifkatoren, die zwar gute Werte bezüglich der FP-Raten und der Precision erzielten, allerdings Mängel bei den TP-Raten und folglich auch dem F-Score aufweisen. Zudem lassen sich die verschiedenen Accuracies der NN-Klassifikatoren mit den Ergebnissen der weiteren Evaluationsmetriken begründen. Es ist zu erkennen, dass sich zwischen den Datensets die Werte der TP-Raten und FP-Raten für beide Label unterscheiden. Infolgedessen weichen auch die F-Scores ab. Diese Abweichungen der Werte führen zu den unterschiedlichen Werten der Accuracies.

Die Mittelwerte zeigen, dass die Gesamtergebnisse durch die Werte der „defekt“ -Spalten stark nach unten beeinflusst werden. Sie zeigen, dass alle Klassifikatoren auf einem durchschnittlichen bis unterdurchschnittlichen Niveau Vorhersagen treffen.

Tabelle 5.4: Ergebnisse der Evaluationsmetriken der dateibasierten Datensets (TPR = TP-Rate)

		„einfaches“ dateibasiertes Datenset			**erweitertes dateibasiertes Datenset**		
		defekt	**fehlerfrei**	**Mittel**	**defekt**	**fehlerfrei**	**Mittel**
J48	TPR / Recall	0,10	0,96	0,53	0,13	0,95	0,54
	FP-Rate	0,04	0,90	0,47	0,05	0,87	0,46
	Precision	0,01	1,00	0,51	0,01	1,00	0,51
	F-Score	0,02	0,98	0,50	0,02	0,97	0,50
KNN	TPR / Recall	0,26	0,76	0,51	0,18	0,78	0,48
	FP-Rate	0,24	0,74	0,49	0,22	0,82	0,47
	Precision	0,01	1,00	0,51	0,00	1,00	0,50
	F-Score	0,01	0,86	0,44	0,01	0,87	0,44
LR	TPR / Recall	0,32	0,89	0,61	0,28	0,88	0,58
	FP-Rate	0,12	0,68	0,40	0,12	0,72	0,42
	Precision	0,01	1,00	0,51	0,01	1,00	0,51
	F-Score	0,03	0,94	0,49	0,02	0,94	0,48
NB	TPR / Recall	0,68	0,14	0,41	0,74	0,14	0,44
	FP-Rate	0,86	0,32	0,59	0,86	0,27	0,57
	Precision	0,00	0,99	0,50	0,00	0,99	0,50
	F-Score	0,01	0,25	0,13	0,01	0,25	0,13
NN	TPR / Recall	0,05	0,80	0,43	0,01	0,99	0,50
	FP-Rate	0,20	0,95	0,58	0,01	0,99	0,50
	Precision	0,00	0,99	0,50	0,01	1,00	0,51
	F-Score	0,00	0,89	0,45	0,01	0,99	0,50
RF	TPR / Recall	0,06	0,98	0,52	0,04	0,98	0,51
	FP-Rate	0,02	0,94	0,48	0,02	0,97	0,50
	Precision	0,02	1,00	0,51	0,01	1,00	0,51
	F-Score	0,03	0,90	0,47	0,02	0,99	0,51
SGD	TPR / Recall	0,20	0,94	0,57	0,18	0,94	0,56
	FP-Rate	0,06	0,81	0,44	0,06	0,82	0,44
	Precision	0,02	1,00	0,51	0,02	1,00	0,51
	F-Score	0,03	0,97	0,50	0,03	0,97	0,50
SVM	TPR / Recall	0,20	0,95	0,58	0,19	1,00	0,60
	FP-Rate	0,05	0,81	0,43	0,05	0,81	0,43
	Precision	0,02	1,00	0,51	0,02	1,00	0,51
	F-Score	0,04	0,97	0,51	0,04	0,98	0,51

ROC-Kurven und ROC-Bereiche

Die ROC-Kurven samt ihrer zugehörigen ROC-Bereiche (als „AUC“ vermerkt) der dateibasierten Datensets sind in Abbildung 5.6 abgebildet. Die dargestellten Farbverläufe der Kurven verdeutlichen erneut keine für diesen Zweck relevanten Informationen und können somit ignoriert werden.

Ein erster Blick auf die ROC-Kurven der Klassifikatoren zeigt ein relativ einheitliches Bild von Kurven, die sich sehr stark an Winkelhalbierenden orientieren und zwischen den Datensets einheitlich sind. Mit der Ausnahme der RF Klassifikatoren beider Datensets zeigen die Kurven mit ihren zugehörigen ROC-Bereichen eine unterdurchschnittliche bis unerwünschte Performanz („Raten“ der Label) an. Mit einem ROC-Bereich von 0,71 besitzen die RF-Klassifikatoren eine überdurchschnittliche Performanz im übergeordneten Vergleich.

Der Vergleich zwischen den NN-Klassifikatoren zeigt, dass der Klassifikator des einfachen Datensets das unerwünschte Verhalten zeigt und gleichzeitig inverse Vorhersagen trifft. Dies bedeutet, dass er in manchen Fällen statt dem Label „fehlerfrei“ das Label „defekt“ vorhersagt und umgekehrt. Die Kurve sowie der ROC-Bereich des Klassifikators des erweiterten Datensets zeigen hingegen nur das unerwünschte Verhalten in Form des „Ratens“ der Vorhersagen. Dies zeigt, dass die in Abbildung 5.5 dargestellten hohen Accuracies der NN-Klassifikatoren durch Zufall entstanden sind. In einer weiteren Anwendung der Testdaten hätten die Ergebnisse anders ausfallen können.

Zusammenfassung

Die Ergebnisse sämtlicher Metriken zeigte ein undifferenziertes und undeutliches Bild der Performanz der Klassifikatoren je Datenset. Der Grund für die gezeigten Ergebnisse ist die Unbalanciertheit der jeweiligen Testdatensets mit einem Anteil von „defekt“-Instanzen von 1%. Damit für dieses Label gute Ergebnisse erzielt werden können, müssen viele korrekte Vorhersagen der Klassifikatoren getroffen werden. Dies war jedoch für das vorhandene Testdatenset nicht der Fall, sodass viele Vorhersagen fälschlicherweise dem dominierenden Label „fehlerfrei“ zugeordnet wurden, die dann allerdings infolgedessen hohe FP-Raten aufweisen. Diesen Umstand spiegeln auch die Mittelwerte in Tabelle 5.4 wider. Aufgrund der deutlich schwächeren Werte der „defekt“-Spalten, werden die Gesamtwerte, die durch den Mittelwert repräsentiert werden, nach unten beeinflusst. Wie jedoch schon in Abschnitt 4.2 erläutert wurde, ist es nicht vorgesehen, die Unbalaniertheit der Testdatensets mithilfe des SMOTE-Algorithmus auszugleichen.

Um ein differenzierteres Bild der Performanz der Klassifikatoren zu erhalten, wurde eine weitere Testmethode, als Alternative zur Aufteilung der Datensets in Training- und Testdaten, hinzugezogen. Dabei handelt es sich um die sogenannte „n-fold-cross-validation“. Bei dieser Methode werden die Trainingsdaten in `n` gleich große Datenmengen („folds“) aufgeteilt, welche dann jeweils mit einer Split-Ratio von 90:10 zum Training der einzelnen Klassifikatoren dienen [39]. Die Ergebnisse der Evaluationsmetriken bilden sich dann aus dem Mittelwert der Einzelergebnisse der `n` Vorgänge [39]. Diese Methode findet auch in der wissenschaftlichen Literatur Anwendung (zum Beispiel [1, 8, 3]. In diesem Fall wird eine 10-fold-cross-validation durchgeführt. Die Ergebnisse der Evaluationsmetriken sind in Tabelle 5.5 aufgeführt. Ein Überblick über die Ergebnisse zeigt, dass diese wesentlich differenzierter und nachvollziehbarer ausgefallen sind sowie keine signifikanten Unterschiede zwischen den Datensets aufweisen. Es zeigt sich, dass die beiden Entscheidungsbaum-basierten Klassifikatoren J48 und RF die im Vergleich höchste Performanz mit einer Accuracy von 96% beziehungsweise 98% besitzen. Die FP-Rate

der Klassifikatoren ist außerdem äußerst gering. Gleiches zeigt sich für den KNN-Klassifikator. Die weiteren Ergebnisse der Metriken bestätigen die hohe Performanz. Die Klassifikatoren SGD und SVM zeigen anhand ihrer ROC-Bereiche (siehe ROC-Area), dass sie unerwünschte Vorhersagen in Form des „Ratens“ durchführen. Auffällig ist, dass die Werte zwischen den NN-Klassifikatoren beider Datensets nicht mehr abweichen. Beide Klassifikatoren arbeiten wie die weiteren, bisher unerwähnten, Klassifikatoren auf einem durchschnittlichen Niveau.

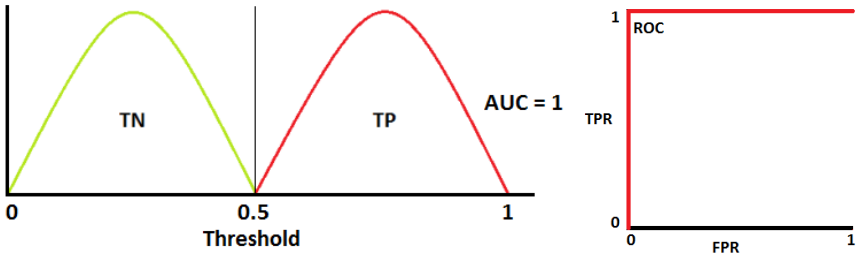
Zusammenfassend lässt sich anhand der Ergebnisse beider Tests feststellen, dass der Einbezug der featurebasierten Metriken keinen nennenswerten positiven Einfluss auf die Ergebnisse der dateibasierten Fehlervorhersage besitzt. Hinzugefügt werden muss jedoch, dass sie die Ergebnisse auch nicht negativ beeinflussen. Ein Grund dafür ist, dass die Auswahl an Attributen für beide dateibasierten Datensets sehr hoch ist. Das „einfache“ Datenset umfasst 17 Attribute (+ Label) und das erweiterte Datenset umfasst 28 Attribute (+ Label). Eine Reduzierung der Attribute in Form einer Attributsauswahl könnte dazu beitragen, den Einfluss der featurebasierten näher zu analysieren. Die Attributsauswahl beschreibt einen automatisierten Prozess, welcher dazu dient, die optimale Anzahl und Auswahl von Attributen für das Training eines Klassifikators zu bestimmen. Eine solche Funktion bietet das für die Arbeit verwendete Werkzeug WEKA. Die Autoren einer wissenschaftlichen Arbeit, die sich mit Codemetriken als Attributen zur Fehlervorhersage beschäftigte, kam zu dem Resultat, dass bereits drei Attribute ausreichend seien [37]. **To be determined: Attributauswahl durchführen?**

RQ3c: WIE LASSEN SICH DIE KLASSIFIKATOREN MIT WEITEREN VORHERSAGETECHNIKEN, DIE KEINE FEATURES NUTZEN, VERGLEICHEN?

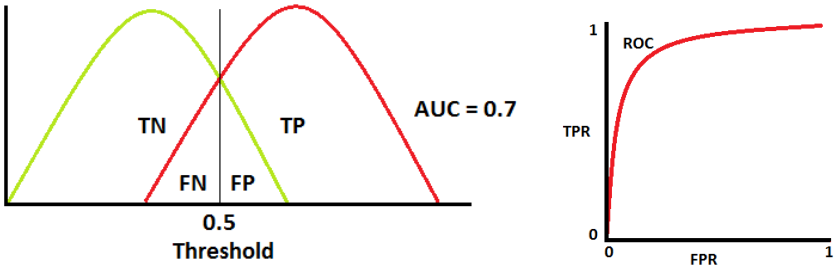
Es wurde auf eine Methode zur Erstellung eines dateibasierten Datensets aus der wissenschaftlichen Literatur zurückgegriffen [18]. Dieses Datenset basiert auf den mittels PyDriller abgerufenen Daten und umfasst 17 Prozessmetriken als Attribute. Zum besseren Vergleich wurde ein zusätzliches erweitertes dateibasiertes Datensets erstellt, welches die selben Prozessmetriken, aber auch zusätzlich die featurebasierten Metriken auf Dateiebene gemapped, umfasst.

RQ3d: WIE BEEINFLUSST DIE VERWENDUNG VON FEATUREBASIERTEN METRIKEN DIE FEHLERVORHERSAGE?

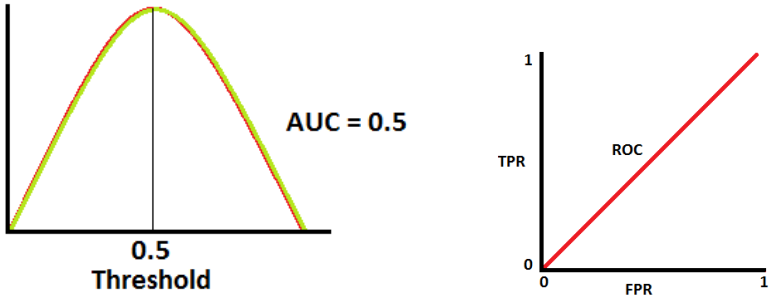
Die Auswirkungen des Einbezugs der featurebasierten Metriken sind kaum messbar. Sowohl die Klassifikatoren des „einfachen“ dateibasierten Datensets als auch die des erweiterten dateibasierten Datensets, erzielen die selben Ergebnisse mit geringen Abweichungen. Sie sind somit jeweils gleich performant bezüglich der Vorhersagen.



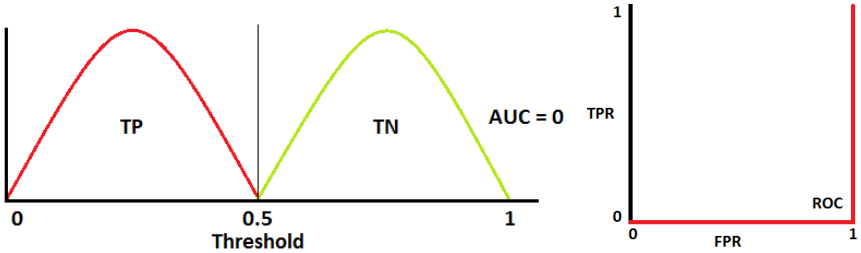
(a) Idealfall **(b)** Idealfall



(c) „Normalfall“ **(d)** „Normalfall“



(e) „Worst Case“ **(f)** „Worst Case“



(g) Fehlerfall **(h)** Fehlerfall

Abbildung 5.2: Beispiel zur Interpretation der ROC-Kurve und des ROC-Bereiches (TPR = TP-Rate, FPR = FP-Rate, Threshold = Schwellenwert) [19]

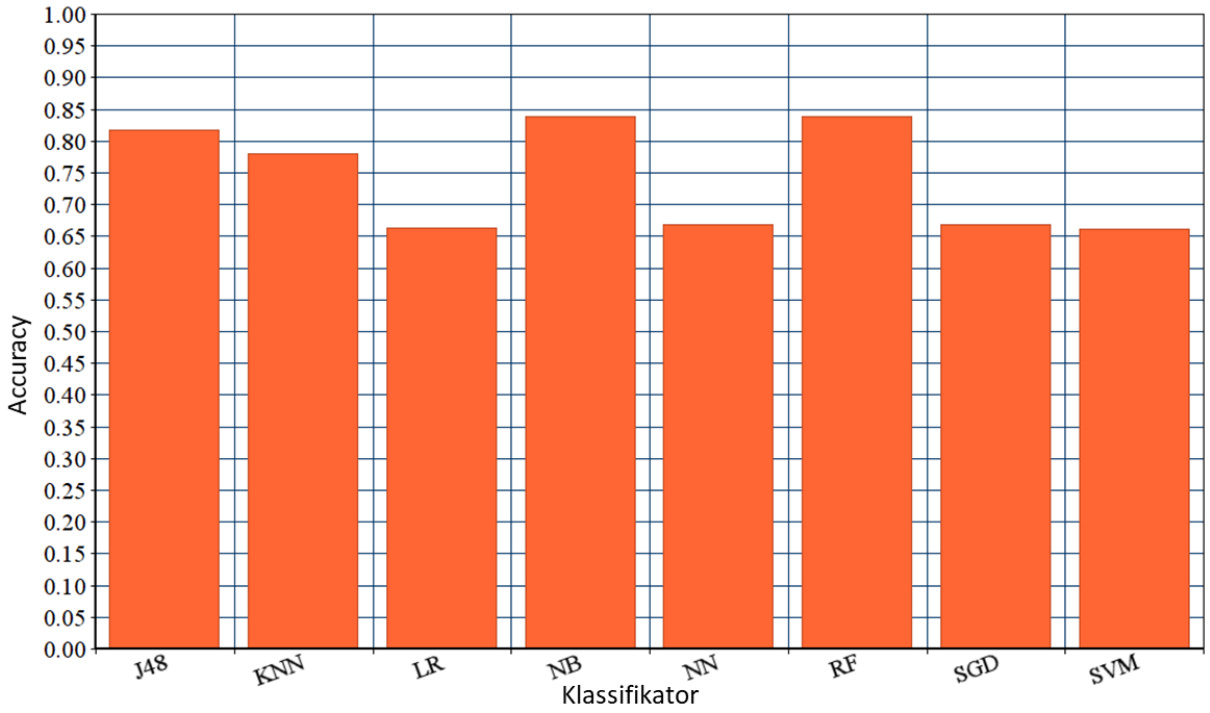
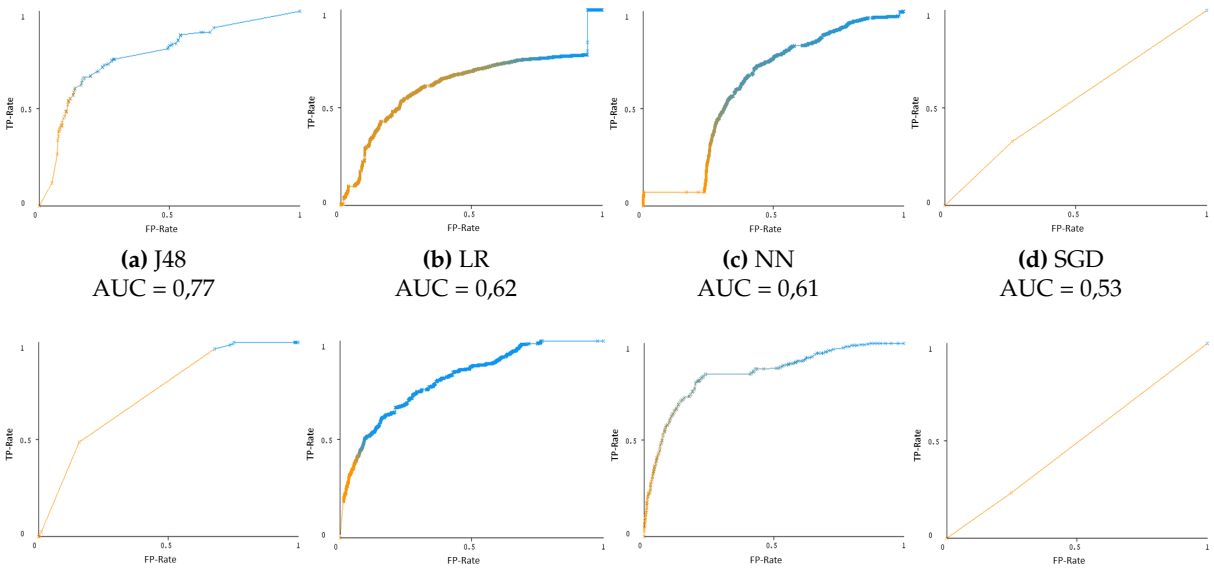


Abbildung 5.3: Vergleich der Accuracies des featurebasierten Datensets



(a) J48
AUC = 0,77

(b) LR
AUC = 0,62

(c) NN
AUC = 0,61

(d) SGD
AUC = 0,53

(e) KNN
AUC = 0,73

(f) NB
AUC = 0,80

(g) RF
AUC = 0,84

(h) SVM
AUC = 0,49

Abbildung 5.4: ROC-Kurven des featurebasierten Datensets

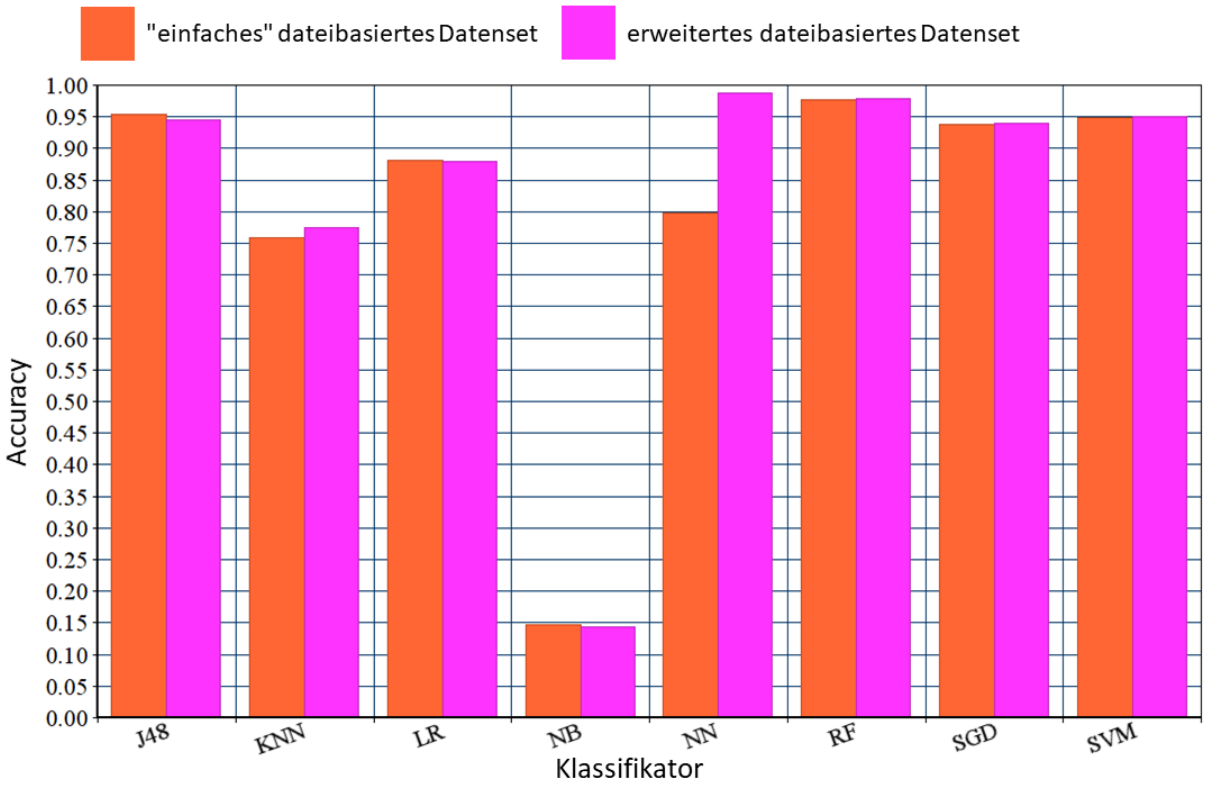
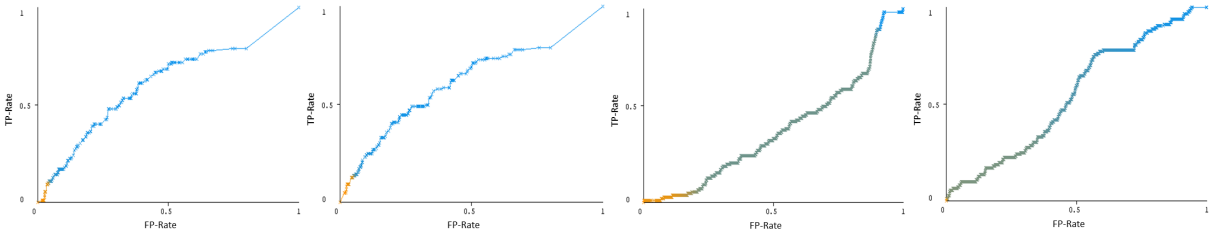


Abbildung 5.5: Vergleich der Accuracies der dateibasierten Datensets

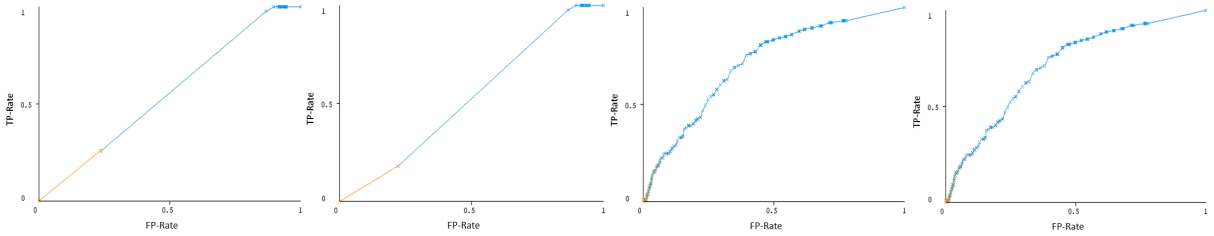


(a) J48 einf.
AUC = 0,61

(b) J48 erw.
AUC = 0,61

(c) NN einf.
AUC = 0,37

(d) NN erw.
AUC = 0,54

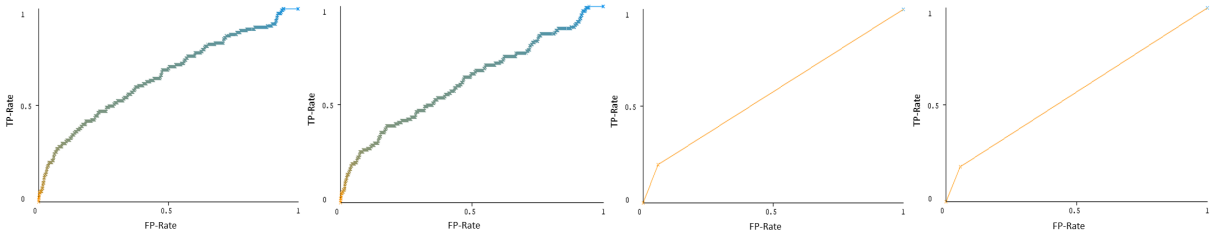


(e) KNN einf.
AUC = 0,55

(f) KNN erw.
AUC = 0,52

(g) RF einf.
AUC = 0,71

(h) RF erw.
AUC = 0,71

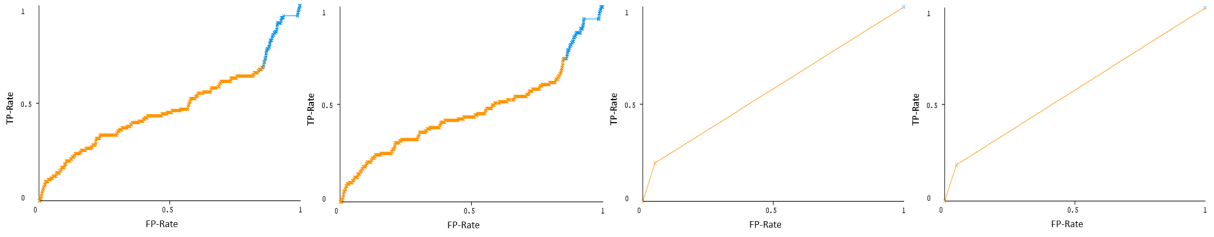


(i) LR einf.
AUC = 0,65

(j) LR erw.
AUC = 0,62

(k) SGD einf.
AUC = 0,57

(l) SGD erw.
AUC = 0,56



(m) NB einf.
AUC = 0,48

(n) NB erw.
AUC = 0,47

(o) SVM einf.
AUC = 0,57

(p) SVM erw.
AUC = 0,57

Abbildung 5.6: ROC-Kurven der datenbasierten Datensets

Tabelle 5.5: Ergebnisse der Evaluationsmetriken auf Basis der 10-fold-cross-validation (TPR = TP-Rate, Rec. = Recall)

		„einfaches“ dateibasiertes Datenset			erweitertes dateibasiertes Datenset		
		defekt	**fehlerfrei**	**Mittel**	**defekt**	**fehlerfrei**	**Mittel**
J48	Accuracy		gesamt:	0,96		gesamt:	0,96
	TPR/Rec.	0,94	0,97	0,96	0,94	0,97	0,96
	FP-Rate	0,03	0,06	0,05	0,03	0,06	0,05
	Precision	0,96	0,96	0,96	0,96	0,96	0,86
	F-Score	0,95	0,97	0,96	0,95	0,96	0,96
	ROC-Area	0,97	0,97	0,97	0,97	0,97	0,97
KNN	Accuracy		gesamt:	0,91		gesamt:	0,91
	TPR/Rec.	0,87	0,94	0,91	0,87	0,94	0,91
	FP-Rate	0,06	0,13	0,10	0,06	0,13	0,10
	Precision	0,92	0,91	0,92	0,91	0,91	0,91
	F-Score	0,89	0,93	0,91	0,89	0,92	0,91
	ROC-Area	0,93	0,93	0,93	0,93	0,93	0,93
LR	Accuracy		gesamt:	0,65		gesamt:	0,65
	TPR/Rec.	0,34	0,88	0,61	0,35	0,87	0,61
	FP-Rate	0,12	0,66	0,39	0,13	0,65	0,39
	Precision	0,66	0,65	0,66	0,66	0,66	0,66
	F-Score	0,45	0,75	0,60	0,46	0,75	0,61
	ROC-Area	0,74	0,74	0,74	0,74	0,74	0,74
NB	Accuracy		gesamt:	0,52		gesamt:	0,52
	TPR/Rec.	0,93	0,23	0,58	0,94	0,22	0,58
	FP-Rate	0,77	0,07	0,42	0,78	0,06	0,42
	Precision	0,46	0,83	0,65	0,46	0,84	0,65
	F-Score	0,62	0,36	0,49	0,62	0,35	0,49
	ROC-Area	0,65	0,65	0,65	0,65	0,65	0,65
NN	Accuracy		gesamt:	0,71		gesamt:	0,70
	TPR/Rec.	0,55	0,82	0,69	0,54	0,82	0,68
	FP-Rate	0,18	0,45	0,32	0,18	0,46	0,32
	Precision	0,69	0,72	0,71	0,68	0,72	0,70
	F-Score	0,71	0,77	0,74	0,60	0,76	0,68
	ROC-Area	0,78	0,78	0,78	0,79	0,79	0,79
RF	Accuracy		gesamt:	0,98		gesamt:	0,98
	TPR/Rec.	0,97	0,99	0,98	0,97	0,99	0,98
	FP-Rate	0,01	0,04	0,3	0,01	0,03	0,2
	Precision	0,98	0,98	0,98	0,98	0,98	0,98
	F-Score	0,97	0,98	0,98	0,97	0,98	0,98
	ROC-Area	0,99	0,99	0,99	1,00	1,00	1,00
SGD	Accuracy		gesamt:	0,63		gesamt:	0,63
	TPR/Rec.	0,19	0,94	0,57	0,18	0,94	0,56
	FP-Rate	0,06	0,81	0,44	0,06	0,82	0,44
	Precision	0,69	0,62	0,66	0,69	0,62	0,66
	F-Score	0,30	0,75	0,53	0,29	0,75	0,52
	ROC-Area	0,57	0,57	0,57	0,56	0,56	0,56
SVM	Accuracy		gesamt:	0,62		gesamt:	0,62
	TPR/Rec.	0,16	0,95	0,56	0,15	0,95	0,55
	FP-Rate	0,05	0,84	0,45	0,05	0,85	0,46
	Precision	0,69	0,62	0,66	0,69	0,61	0,66
	F-Score	0,26	0,75	0,51	0,25	0,75	0,50
	ROC-Area	0,55	0,55	0,55	0,55	0,55	0,55

Kapitel 6

Fazit

Das abschließende Kapitel dieser Arbeit dient zur Zusammenfassung der Ergebnisse der vorangegangenen Kapitel. Ebenfalls wird ein Ausblick auf eine mögliche Weiterführung dieser Arbeit gegeben.

6.1 Zusammenfassung

Diese Masterarbeit gab einen detaillierten Einblick in die Entwicklung einer Methode zur Vorhersage von Softwarefehlern basierend auf dem Konzept der Software-Features unter Anwendung von Methoden des Machine Learning. Zusätzlich diente diese Arbeit zur Vermittung von Wissen zu den grundlegenden Themenkomplexen „featurebasierte Softwareentwicklung“, „Machine-Learning-Klassifikation“ und „Fehlervorhersage mittels Machine Learning“.

Die Entwicklung der Methode zur Fehlervorhersage gliederte sich dabei in drei Teile: die Erstellung eines featurebasierten Datensets unter der Einbindung von elf Metriken, das Training von Klassifikatoren auf der Basis von acht Klassifikationsalgorithmen und die anschließende Evaluation der Klassifikatoren. Dabei umfasste die Evaluation zudem einen Vergleich zwischen zwei dateibasierten Datensets, deren Metriken aus der wissenschaftlichen Literatur entnommen wurde. Eines dieser Datensets wurde dabei mit den featurebasierten Metriken erweitert, um im direkten Vergleich die Auswirkungen der featurebasierten Metriken auf die Performanz der Vorhersagen zu messen.

Die Ergebnisse der Evaluation zeigten dabei, dass die Klassifikatoren des featurebasierten Datensets eine Treffergenauigkeit von bis zu 84% erreichten und damit überdurchschnittlich präzise Vorhersagen bezüglich der Testdaten treffen können. Der Vergleich zwischen den dateibasierten Datensets zeigte allerdings, dass der Einfluss der featurebasierten Metriken, welche auf Dateiebene gemapped wurden, äußerst gering ist.

6.2 Ausblick

Der Ausblick für zukünftige Arbeiten an dem zugrundeliegenden Thema lässt sich an den Erkenntnissen des Abschnitts „Herausforderungen und Limitationen“ ableiten. Zur präziseren Identifikation von Features im Sourcecode wird ein sogenannter „Parser“ beziehungsweise ein

„Parsing-Tool“ benötigt. Momentan verfügbare Werkzeuge zur Analyse von Featurecode verwenden einen gleichen Ansatz in der Identifikation der Features, so wie er in dieser Arbeit in Form von regulären Ausdrücken zur Anwendung kam. Wie bereits erwähnt, führt diese Methode jedoch dazu, dass viele unerwünschte Features identifiziert werden. Je nach Umfang der umgesetzten Variabilität innerhalb eines Softwareprojekts, kann eine manuelle Ausfilterung der unerwünschten Features einen enormen Aufwand erzeugen. Die Entwicklung einer automatisierten Methode dazu stellt ebenfalls nur eine temporäre Lösung dar, da sie dann nicht universell für weitere Softwareprojekte einsetzbar ist. Ein Parser beziehungsweise ein Parsing-Tool könnte so konfiguriert sein, dass es softwareübergreifend die gewünschten Features korrekt identifizieren kann und unerwünschte Features unbeachtet lässt. Zur Erstellung eines solchen Werkzeugs ist jedoch eine umfangreiche Analyse von featurebasierten Softwareprojekten nötig, um einen Überblick über möglichst viele Implementierungen der Features zu erhalten.

Ein Parsing-Tool kann auch hilfreich für eine bessere Einbindung des Featurebezugs sein. Es könnte verwendet werden, um zu ermitteln, ob die mit einem Commit veröffentliche Veränderung einer Datei tatsächlich im Zusammenhang mit einem Feature steht. Dazu muss in den Diffs analysiert werden, ob ein Feature erwähnt wird (siehe oben) und ob innerhalb des Featurecodes eine Veränderung vorgenommen wurde. Das Pasing-Tool muss somit eine „in-depth“-Analyse der Diffs und gegebenenfalls des Sourcecodes durchführen.

Sollte in der Zukunft ein solches Tool entwickelt werden, so würde es sich anbieten, die zentralen Arbeitsschritte dieser Arbeit unter dessen Zuhilfenahme zu wiederholen, um die verwendete Methodik zu ergänzen und die neuen Ergebnisse mit den Ergebnissen dieser Arbeit zu vergleichen. Gleiches sollte durchgeführt werden, wenn eine Alternative Methodik zur Identifizierung von fehlereinführenden Commits vorgestellt wird. Die bisher genutzte SZZ-Methodik wurde im Rahmen einer wissenschaftlichen Studie als unpräzise „enttarnt“.

Literatur

[1] Mohammed S. Alam und Son T. Vuong. „Random Forest Classification for Detecting Android Malware“. In: *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. IEEE, Aug. 2013. DOI: 10.1109/greencom-ithings-cpscom.2013.122.

[2] Ethem Alpaydin. *Introduction to Machine Learning*. Second Edi. Cambridge, Massachusetts: The MIT Press, 2010. ISBN: 9780262012430.

[3] Abdullah Alsaeedi und Mohammad Zubair Khan. „Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques: A Comparative Study“. In: *Journal of Software Engineering and Applications* 12.05 (2019), S. 85–100. DOI: 10.4236/jsea.2019.125007.

[4] Sven Apel u. a. *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-37521-7.

[5] Markus Borg u. a. „SZZ unleashed: an open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project“. In: *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaLTeSQuE 2019*. ACM Press, 2019. DOI: 10.1145/3340482.3342742.

[6] Venkata Udaya B. Challagulla u. a. „Empirical assessment of machine learning based software defect prediction techniques“. In: *International Journal on Artificial Intelligence Tools* 17.2 (2008), S. 389–400. ISSN: 02182130. DOI: 10.1142/S0218213008003947.

[7] Pete Chapman u. a. „CRISP-DM 1.0“. In: *CRISP-DM Consortium* (2000), S. 76. ISSN: 0957-4174. DOI: 10.1109/ICETET.2008.239.

[8] N. V. Chawla u. a. „SMOTE: Synthetic Minority Over-sampling Technique“. In: *Journal of Artificial Intelligence Research* 16 (Juni 2002), S. 321–357. DOI: 10.1613/jair.953.

[9] Eibe Frank, Mark A. Hall und Ian H. Witten. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Fourth Edition. Morgan Kaufmann, 2016.

[10] Rohith Gandhi. *Support Vector Machine — Introduction to Machine Learning Algorithms*. Online. Zuletzt abgerufen am 20.03.2020. Juni 2018. URL: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.

[11] Awni Hammouri u. a. „Software Bug Prediction using Machine Learning Approach“. In: *International Journal of Advanced Computer Science and Applications* 9.2 (2018). DOI: 10.14569/ijacsa.2018.090212.

[12] Claus Hunsen u. a. „Preprocessor-based variability in open-source and industrial software systems: An empirical study“. In: *Empirical Software Engineering* 21.2 (Apr. 2015), S. 449–482. DOI: 10.1007/s10664-015-9360-1.

[13] KNIMETV. *What is an ROC Curve?* Video on YouTube. Zuletzt abgerufen am 20.03.2020.. Sep. 2019. URL: https://www.youtube.com/watch?v=kWDHmroVWs8.

[14] Jörg Liebig u. a. „An analysis of the variability in forty preprocessor-based software product lines“. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. ACM Press, 2010. DOI: 10.1145/1806799.1806819.

[15] Roland Linder, Jeannine Geier und Mathias Kölliker. „Artificial neural networks, classification trees and regression: Which method for which customer base?“ In: *Journal of Database Marketing & Customer Strategy Management* 11.4 (Juli 2004), S. 344–356. DOI: 10.1057/palgrave.dbm.3240233.

[16] Stefan Luber und Nico Litzel. *Was ist eine Support Vector Machine?* Online. Zuletzt abgerufen am 20.03.2020. Nov. 2019. URL: https://www.bigdata-insider.de/was-ist-eine-support-vector-machine-a-880134/.

[17] Flavio Medeiros u. a. „Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell“. In: *IEEE Transactions on Software Engineering* 44.5 (Mai 2018), S. 453–469. DOI: 10.1109/tse.2017.2688333.

[18] Raimund Moser, Witold Pedrycz und Giancarlo Succi. „A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction“. In: *Proceedings of the 13th international conference on Software engineering - ICSE '08*. ACM Press, 2008. DOI: 10.1145/1368088.1368114.

[19] Sarang Narkhede. *Understanding AUC - ROC Curve*. Online. Zuletzt abgerufen am 20.03.2020. Juni 2018. URL: https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5.

[20] Keiron O'Shea und Ryan Nash. „An Introduction to Convolutional Neural Networks“. In: (26. Nov. 2015). arXiv: http://arxiv.org/abs/1511.08458v2 [cs.NE].

[21] F. Pedregosa u. a. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830.

[22] Chao-Ying Joanne Peng, Kuk Lida Lee und Gary M. Ingersoll. „An Introduction to Logistic Regression Analysis and Reporting“. In: *The Journal of Educational Research* 96.1 (Sep. 2002), S. 3–14. DOI: 10.1080/00220670209598786.

[23] Christopher Preschern. „Patterns to escape the #ifdef hell“. In: *Proceedings of the 24th European Conference on Pattern Languages of Programs - EuroPLop '19*. ACM Press, 2019. DOI: 10.1145/3361149.3361151.

[24] Rodrigo Queiroz, Thorsten Berger und Krzysztof Czarnecki. „Towards predicting feature defects in software product lines“. In: *Proceedings of the 7th International Workshop on Feature-Oriented Software Development - FOSD 2016*. ACM Press, 2016. DOI: 10.1145/3001867.3001874.

[25] Rodrigo Queiroz u. a. „The shape of feature code: an analysis of twenty C-preprocessor-based systems“. In: *Software & Systems Modeling* 16.1 (Juli 2015), S. 77–96. DOI: 10.1007/s10270-015-0483-z.

[26] Foyzur Rahman und Premkumar Devanbu. „How, and why, process metrics are better“. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, Mai 2013. DOI: 10.1109/icse.2013.6606589.

[27] Sebastian Raschka. „Naive Bayes and Text Classification I - Introduction and Theory“. In: (16. Okt. 2014). arXiv: http://arxiv.org/abs/1410.5329v4 [cs.LG].

[28] Jacek Ratzinger, Thomas Sigmund und Harald C. Gall. „On the relation of refactorings and software defect prediction“. In: *Proceedings of the 2008 international workshop on Mining software repositories - MSR '08*. ACM Press, 2008. DOI: 10.1145/1370750.1370759.

[29] Lior Rokach und Oded Maimon. „Decision Trees“. In: *Data Mining and Knowledge Discovery Handbook*. Springer-Verlag, 2005, S. 165–192. DOI: 10.1007/0-387-25465-x_9.

[30] Claude Sammut und Geoffrey I. Webb, Hrsg. *Encyclopedia of Machine Learning and Data Mining*. Springer US, 2017. DOI: 10.1007/978-1-4899-7687-1.

[31] Jacek Śliwerski, Thomas Zimmermann und Andreas Zeller. „When do changes induce fixes?“ In: *ACM SIGSOFT Software Engineering Notes* 30.4 (Juli 2005), S. 1. DOI: 10.1145/1082983.1083147.

[32] Le Son u. a. „Empirical Study of Software Defect Prediction: A Systematic Mapping“. In: *Symmetry* 11.2 (Feb. 2019), S. 212. DOI: 10.3390/sym11020212.

[33] Davide Spadini, Mauricio Aniche und Alberto Bacchelli. „PyDriller: Python framework for mining software repositories“. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press, 2018. DOI: 10.1145/3236024.3264598.

[34] Aishwarya V Srinivasan. *Stochastic Gradient Descent — Clearly Explained !!* Online. Zuletzt abgerufen am 20.03.2020. Sep. 2019. URL: https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31.

[35] Richard M. Stallmann und Zachary Weinberg. *The C Preprocessor*. For GCC version 6.3.0. Free Software Foundation, Inc. 2016. URL: https://scicomp.ethz.ch/public/manual/gcc/6.3.0/cpp.pdf.

[36] Thomas Thüm u. a. „A Classification and Survey of Analysis Strategies for Software Product Lines“. In: *ACM Computing Surveys* 47.1 (Juni 2014), S. 1–45. DOI: 10.1145/2580950.

[37] Huanjing Wang, Taghi M. Khoshgoftaar und Naeem Seliya. „How many software metrics should be selected for defect prediction?“ In: *Proceedings of the 24th International Florida Artificial Intelligence Research Society, FLAIRS - 24* Mi (2011), S. 69–74.

[38] Ming Wen u. a. „Exploring and exploiting the correlations between bug-inducing and bug-fixing commits“. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*. ACM Press, 2019. DOI: 10.1145/3338906.3338962.

[39] Ian H. Witten. *Data Mining with WEKA: Cross-valiadation*. Online-Video. Zuletzt abgerufen am 20.03.2020. URL: https://www.futurelearn.com/courses/data-mining-with-weka/0/steps/25384.

[40] Zhongheng Zhang. „Introduction to machine learning: k-nearest neighbors“. In: *Annals of Translational Medicine* 4.11 (Juni 2016), S. 218–218. DOI: 10.21037/atm.2016.03.37.

[41] Thomas Zimmermann, Rahul Premraj und Andreas Zeller. „Predicting Defects for Eclipse“. In: *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, Mai 2007. DOI: 10.1109/promise.2007.10.

Anhang A

PyDriller-Sourcecode-Ausschnitt zur Konsolenausgabe von Metadaten eines Commits

```
1 for commit in RepositoryMining("link_to_repo").traverse_commits():
2   for m in commit.modifications:
3     print(
4           "Author {}".format(commit.author.name),
5           " modified {}".format(m.filename),
6           " with a change type of {}".format(m.change_type.name),
7           " and the complexity is {}".format(m.complexity)
8     )
```

Listing A.1: Beispielhafter PyDriller-Code zur Ausgabe von Metadaten von Commits

Folgende Metadaten eines Commits können mit diesem Code-Ausschnitt abgerufen werden: Autor des Commits (Zeile 4), Name der veränderten Dateien (Zeile 5), Typ der Veränderung (Zeile 6) und jeweilige zyklomatische Komplexität der Dateien (Zeile 7).

Anhang B

Links der für die Erstellung des Datensets verwendeten Softwareprojekte

	Link zur Website	**Link zum Repository**
Blender	https://www.blender.org/	https://github.com/sobotka/blender
Busybox	https://busybox.net/	https://git.busybox.net/busybox/
Emacs	https://www.gnu.org/software/emacs/	https://github.com/emacs-mirror/emacs
GIMP	https://www.gimp.org/	https://gitlab.gnome.org/GNOME/gimp
Gnumeric	http://www.gnumeric.org/	https://gitlab.gnome.org/GNOME/gnumeric
gnuplot	http://gnuplot.info/	https://github.com/gnuplot/gnuplot
Irssi	https://irssi.org/	https://github.com/irssi/irssi
libxml2	http://www.xmlsoft.org/	https://gitlab.gnome.org/GNOME/libxml2
lighttpd	https://www.lighttpd.net/	https://git.lighttpd.net/lighttpd/lighttpd1.4.git/
MPSolve	https://numpi.dm.unipi.it/software/mpsolve	https://github.com/robol/MPSolve
Parrot	http://parrot.org/	https://github.com/parrot/parrot
Vim	https://www.vim.org/	https://github.com/vim/vim
xfig	https://sourceforge.net/projects/mcj/	https://sourceforge.net/p/mcj/xfig/ci/master/tree/
Websites zuletzt abgerufen am 13. Januar 2020.		

Anhang C

Accuracies der Klassifikatoren je Datenset

Tabelle C.1: Accuracies des featurebasierten Datensets

Klassifikator	**Accuracies**
J48	0,82
KNN	0,78
LR	0,66
NB	0,84
NN	0,67
RF	0,84
SGD	0,67
SVM	0,66

Tabelle C.2: Accuracies der Datensets

Klassifikator	**„einfaches“ dateibasiertes Datenset**	**erweitertes dateibasiertes Datenset**
J48	0,95	0,94
KNN	0,76	0,77
LR	0,88	0,89
NB	0,15	0,14
NN	0,80	0,99
RF	0,98	0,98
SGD	0,94	0,94
SVM	0,95	0,95

Anhang D

Erweiterte Ergebnisse der Evaluationsmetriken

Die nachfolgenden Tabellen enthalten die Ergebnisse der Evaluation mit zusätzlichen Metriken. Die Metrik „PRC-Area“ ist wie folgt definiert:

- PRC-Bereich (PRC-Area)
  Dieser Wert unterliegt der Messung der PRC (Precision-Recall-Curve), welche die Werte der Precision und des Recalls gegenüberstellt. Die Messung und Interpretation des Bereiches unter dieser Kurve erfolgt analog zum ROC-Bereich.

Tabelle D.1: Erweiterte Ergebnisse der Evaluationsmetriken des featurebasierten Datensets

		defekt	fehlerfrei	Mittel
J48	TP-Rate	0,58	0,87	0,73
	FP-Rate	0,13	0,42	0,28
	Precision	0,48	0,91	0,70
	Recall	0,58	0,87	0,73
	F-Score	0,53	0,89	0,71
	ROC-Area	0,77	0,77	0,77
	PRC-Area	0,39	0,93	0,66
KNN	TP-Rate	0,48	0,86	0,67
	FP-Rate	0,16	0,52	0,68
	Precision	0,40	0,89	0,65
	Recall	0,48	0,86	0,67
	F-Score	0,44	0,86	0,66
	ROC-Area	0,73	0,73	0,73
	PRC-Area	0,31	0,91	0,61
LR	TP-Rate	0,40	0,72	0,56
	FP-Rate	0,28	0,60	0,44
	Precision	0,23	0,85	0,54
	Recall	0,40	0,72	0,56
	F-Score	0,30	0,78	0,54
	ROC-Area	0,62	0,62	0,62
	PRC-Area	0,25	0,89	0,57
NB	TP-Rate	0,42	0,93	0,68
	FP-Rate	0,07	0,58	0,33
	Precision	0,55	0,88	0,72
	Recall	0,42	0,93	0,68
	F-Score	0,48	0,91	0,70
	ROC-Area	0,80	0,80	0,80
	PRC-Area	0,50	0,95	0,73
	TP-Rate	0,43	0,72	0,58
	FP-Rate	0,28	0,57	0,43

Tabelle D.2: Erweiterte Ergebnisse der Evaluationsmetriken der dateibasierten Datensets

		„einfaches“ dateibasiertes Datenset			erweitertes dateibasiertes Datenset		
		defekt	**fehlerfrei**	**Mittel**	**defekt**	**fehlerfrei**	**Mittel**
J48	TP-Rate	0,10	0,96	0,53	0,13	0,95	0,54
	FP-Rate	0,04	0,90	0,47	0,05	0,87	0,46
	Precision	0,01	1,00	0,51	0,01	1,00	0,51
	Recall	0,10	0,96	0,53	0,13	0,95	0,54
	F-Score	0,02	0,98	0,50	0,02	0,97	0,50
	ROC-Area	0,61	0,61	0,61	0,61	0,61	0,61
	PRC-Area	0,01	1,00	0,51	0,01	1,00	0,51
KNN	TP-Rate	0,26	0,76	0,51	0,18	0,78	0,48
	FP-Rate	0,24	0,74	0,49	0,22	0,82	0,47
	Precision	0,01	1,00	0,51	0,00	1,00	0,50
	Recall	0,26	0,76	0,51	0,18	0,78	0,48
	F-Score	0,01	0,86	0,44	0,01	0,87	0,44
	ROC-Area	0,55	0,55	0,55	0,52	0,52	0,52
	PRC-Area	0,01	1,00	0,51	0,01	1,00	0,51
LR	TP-Rate	0,32	0,89	0,61	0,28	0,88	0,58
	FP-Rate	0,12	0,68	0,40	0,12	0,72	0,42
	Precision	0,01	1,00	0,51	0,01	1,00	0,51
	Recall	0,32	0,86	0,61	0,28	0,88	0,58
	F-Score	0,03	0,94	0,49	0,02	0,94	0,48
	ROC-Area	0,65	0,65	0,65	0,62	0,62	0,62
	PRC-Area	0,02	1,00	0,51	0,01	1,00	0,51
NB	TP-Rate	0,68	0,14	0,41	0,74	0,14	0,44
	FP-Rate	0,86	0,32	0,59	0,86	0,27	0,57
	Precision	0,00	0,99	0,50	0,00	0,99	0,50
	Recall	0,68	0,14	0,41	0,74	0,14	0,44
	F-Score	0,01	0,25	0,13	0,01	0,25	0,13
	ROC-Area	0,48	0,48	0,48	0,47	0,47	0,47
	PRC-Area	0,01	0,99	0,50	0,01	0,99	0,50
NN	TP-Rate	0,05	0,80	0,43	0,01	0,99	0,50
	FP-Rate	0,20	0,95	0,58	0,01	0,99	0,50
	Precision	0,00	0,99	0,50	0,01	1,00	0,51
	Recall	0,05	0,80	0,43	0,01	0,99	0,50
	F-Score	0,00	0,89	0,45	0,01	0,99	0,50
	ROC-Area	0,37	0,37	0,37	0,54	0,54	0,54
	PRC-Area	0,00	0,99	0,50	0,01	1,00	0,51
RF	TP-Rate	0,06	0,98	0,52	0,04	0,98	0,51
	FP-Rate	0,02	0,94	0,48	0,02	0,97	0,50
	Precision	0,02	1,00	0,51	0,01	1,00	0,51
	Recall	0,06	0,98	0,52	0,04	0,98	0,51
	F-Score	0,03	0,90	0,47	0,02	0,99	0,51
	ROC-Area	0,71	0,71	0,71	0,71	0,71	0,71
	PRC-Area	0,02	1,00	0,51	0,01	1,00	0,51
SGD	TP-Rate	0,20	0,94	0,57	0,18	0,94	0,56
	FP-Rate	0,06	0,81	0,44	0,06	0,82	0,44
	Precision	0,02	1,00	0,51	0,02	1,00	0,51
	Recall	0,20	0,94	0,57	0,18	0,94	0,56
	F-Score	0,03	0,97	0,50	0,03	0,97	0,50
	ROC-Area	0,57	0,57	0,57	0,56	0,56	0,56
	PRC-Area	0,01	1,00	0,51	0,01	1,00	0,51
SVM	TP-Rate	0,20	0,95	0,58	0,19	1,00	0,60
	FP-Rate	0,05	0,81	0,43	0,05	0,81	0,43
	Precision	0,02	1,00	0,51	0,02	1,00	0,51
	Recall	0,20	0,95	0,58	0,19	0,96	0,60
	F-Score	0,04	0,97	0,51	0,04	0,98	0,51
	ROC-Area	0,57	0,57	0,57	0,57	0,57	0,57
	PRC-Area	0,01	1,00	0,51	0,01	1,00	0,51