Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Bachelorarbeit (Informatik)

Fehlervorhersage in Java-Code mittels Machine Learning

Autoren	Tobias Meier Yacine Mekesser
Hauptbetreuung	Mark Cieliebak

Datum	10.06.2016

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Erklärung betreffend das selbständige Verfassen einer Projektarbeit an der School of Engineering

Mit der Abgabe dieser Projektarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Projektarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

……

Unterschriften:

……

……..

……

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Projektarbeiten zu Beginn der Dokumentation nach dem Titelblatt mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Zusammenfassung

Zunehmend komplexe Softwaresysteme und agile Entwicklungsmethoden machen es immer schwieriger, die Code-Qualität eines Softwareprojektes zu kontrollieren. Deshalb wäre ein System zur Fehlervorhersage wünschenswert. Wir glauben, dass Machine Learning das Potenzial bietet, um ein solches System zu ermöglichen. Diese Bachelorarbeit hat zum Ziel, bestehende Ansätze mit Konzepten der Textanalyse, insbesondere N-Grams, zu erweitern. Ausserdem soll eine Grundlage für zukünftige Arbeiten zu diesem Themengebiet geschaffen werden. Dafür wurde ein umfassendes und modulares Toolset entwickelt. Dieses ist in der Lage, Git-Repositories beliebiger Java-Projekte zu analysieren. Die dabei extrahierten Daten können dann als Lernbasis für einen Machine-Learning-Algorithmus verwendet werden. Damit versuchen wir vorherzusagen, wie viele Bugfixes eine Dateiversion in den kommenden Monaten erfahren wird. Die implementierte Lösung zeigte, dass statistisch signifikante Zusammenhänge zwischen den genutzten Features und der Fehleranfälligkeit von Java-Dateien bestehen. Jedoch konnten die Resultate der Experimente den Nutzen der N-Grams nicht bestätigen.

Schlüsselwörter: Fehlervorhersage, Machine Learning, Regression, N-Grams, Repository Mining, Software Metrics, Feature Design

Abstract

The rising complexity of software systems and agile development methods makes it increasingly difficult to control the quality of a software project. This makes a system for defect prediction desirable. We assume that machine learning offers the potential to realise such a solution. The goal of this bachelor thesis is to expand existing approaches by incorporating concepts originating in text analysis, especially N-Grams. Furthermore, a foundation for future work on this topic should be created. For this, a comprehensive and modular toolset was developed. It is able to analyse the Git repository of arbitrary Java projects. The extracted data can then be used as the basis for training a machine learning algorithm. With the resulting model, we try to predict how many bugfixes a file version will receive in the coming months. The implemented solution shows a significant correlation between the features used and the error-proneness of Java files. However, the results of our experiments could not conclusively prove the usefulness of N-Grams in defect prediction.

Keywords: Defect prediction, Machine Learning, Regression, N-Grams, Repository Mining, Software Metrics, Feature Design

Vorwort

Vielen Dank an Mark Cielibak, welcher diese Arbeit ermöglicht hat. Wir danken auch Dominic Egger und Fatih Uzdilli für ihre vielen kreativen Inputs. Ausserdem bedanken wir uns auch bei Ramona Ent, Philipp Riesen und Paul Keller für das Korrekturlesen unserer Arbeit. Nicht zu vergessen ist auch die Unterstützung von Freunden und Familien.

Inhaltsverzeichnis

1 Einleitung

In vielen Bereichen unseres Lebens sind softwaregestützte Systeme anzutreffen. Nicht nur die Menge an Softwareprojekten, sondern auch deren Komplexität nimmt stetig zu. Damit steigt sowohl die Wahrscheinlichkeit als auch die Auswirkung von Softwarefehlern. Ausserdem kämpfen viele Firmen mit einem grossen Umfang an Code, der sich gerade mit agilen Entwicklungsmethoden dauernd verändert. Damit wird die Qualitätskontrolle einer solchen Codebasis enorm schwierig.

Die Motivation dieser Bachelorarbeit besteht darin, die Grundlage für ein Tool zu schaffen, welches in Zukunft dazu in der Lage ist, Bugs im Programmcode vorherzusagen. Damit könnten Entwickler Fehler schneller finden und Projektmanager ihre Ressourcen effizienter auf kritische Komponenten einplanen.

Diese Arbeit befasst sich mit statischer Java-Code- und Metadaten-Analyse zur Vorhersage von Softwarefehlern. Im Gegensatz zu den meisten bestehenden Fehleranalyseprodukten versuchen wir, eine Fehlervorhersage mittels Machine Learning (ML) zu realisieren. Ausserdem wird untersucht, ob Konzepte, die normalerweise in der Textanalyse verwendet werden, auch in der Analyse von Source Code anwendbar sind.

Als Grundlage der Fehlervorhersagen dienen verschiedene Metriken. Diese werden aus dem bestehenden Source Code eines Softwareprojekts, dessen Versionsverwaltung und Issue-Tracking berechnet. Aus den daraus generierten Features wird dann ein Trainingsdatensatz gebildet. Damit kann ein ML-Modell trainiert werden, sodass es anschliessend möglichst zuverlässige Vorhersagen über neue Datensätze treffen kann.

1.1 Übersicht

Der Einsatz von ML erfordert Trainings- und Testdaten. Die Arbeit beschränkt sich deshalb nicht nur auf das Feature Design und das Implementieren eines ML-Modells, sondern befasst sich auch mit dem Data Mining. Dafür wurde ein Repository Mining Tool implementiert, welches den ersten Teil der Arbeit bildet.

Das anschliessende Generieren der Features über den Source Code und die gesammelten Projektmetadaten erledigt ein Feature Extractor.

Am Ende werden die generierten Features von einer ML-Komponente zum Trainieren eines Modells verwendet. Als Trainings- und Testdaten dienen Dateiversionen aus verschiedenen Zeiträumen eines beliebigen Java-Projekts.

1.2 Nötiges Vorwissen

Erfahrung im Umgang mit Versionierungs- und Issue-Tracking-Systemen ist von Vorteil. Dem Leser sollten die Begriffe Dateiversion, “Commit”, “Bug” und “Enhancement” im Bezug auf solche Systeme bekannt sein. Mögliche Einstiegspunkte in dieses Thema sind die GitHub Guides [1] sowie die Dokumentation von Git [2].

Rudimentäre Kenntnisse von statischer Code-Analyse und deren gängigsten Metriken sind nicht zwingend notwendig, aber zweckdienlich. Ebenfalls von Vorteil ist statistisches Vorwissen.

Grundlegende Kenntnisse im Themenfeld “Machine Learning” sind für das Verständnis unabdingbar. Für einen umfassenden Einstieg in ML empfehlen wir den Coursera-Kurs *Machine Learning* von Andrew Ng [3], welcher auch uns beim Einarbeiten in dieses umfassende Thema geholfen hat.

Details zu den genutzten ML-Algorithmen werden im Kapitel 2.1 beschrieben. Im Kapitel 2.2 wird auf die verwendeten Regressions-Metriken grob eingegangen.

1.3 Ausgangslage

Fehlervorhersage hat in der Softwareentwicklung einen grossen Stellenwert. Dementsprechend ist sie ein grosses Gebiet, in dem bereits Arbeiten und Produkte zu Dutzenden Ansätzen bestehen. Im Folgenden soll deshalb der Kontext dieser Arbeit erläutert werden. Es werden zunächst einige bestehende Code-Analyse-Lösungen aufgezeigt. Anschliessend wird auf die Erkenntnisse unserer Literaturrecherche zu den Themen ML und Fehlervorhersage eingegangen.

1.3.1 Statische und dynamische Code-Analyse

Statische Code-Analyse (engl. static program analysis, manchmal auch als *linter* bezeichnet) ist eine typische Analyseform für Computersoftware, welche versucht, Fehler oder qualitative Mängel (“Code Smell”) in Programmcode zu finden, ohne dass das Programm dafür ausgeführt werden muss. Dafür wird meistens direkt der Source Code analysiert, manchmal aber auch der Maschinencode oder die erzeugte Intermediate Language.

Man kann die statische Code-Analyse den White-Box-Testverfahren zuordnen, da der Source Code dazu benötigt wird. Je nach Tool variiert die Analysetiefe und der Detailgrad, manche markieren bloss syntaktische Fehler im Editor, andere versuchen z. B. die Korrektheit des Codes mittels formaler Methoden mathematisch zu beweisen. Insbesondere für Dienstleister mit hohen Anforderungen an Sicherheit, wie z. B. Banken, Verkehrsbetrieben oder Kernkraftwerken, ist ein hohes Mass an Fehlerfreiheit nötig.

Im Unterschied zur statischen Code-Analyse erfordert die dynamische Code-Analyse die Ausführung von Programme auf einem realen oder virtuellen Prozessor, um sie zu analysieren. Beispiele für dynamische Code-Analyse-Tools sind: *Valgrind* [4], ein umfassendes Dynamic-Analysis-Framework für C, *Cobertura* [5], ein Code-Coverage-Utility für Java, oder ferner *American Fuzzy Lop* [6], ein security-orientiertes, leistungfähiges Fuzzing-Tool.

Für Java gibt es eine Vielzahl von statischen Code-Analyse-Tools: Zum Beispiel sucht *FindBugs* [7] den Java-Bytecode nach Fehlermustern ab, *Checkstyle* [8] prüft das Einhalten von Coding Standards und *PMD* [9] versucht regelbasiert ineffizienten Code aufzuspüren. Ebenfalls erwähnenswert sind *Lint* [10] für C wegen seiner historischen Relevanz und *Code Climate* [11], da es nicht nur multilingual ist, sondern auch durch seinen Platform as a Service (PaaS) auffällt.

Diese Arbeit nutzt ausschliesslich den Source Code, Informationen des Issue-Trackings und der Versionsverwaltung eines Projekts. Ein Ausführen des Codes ist nicht nötig. Deshalb lässt sie sich der statischen Code-Analyse zuordnen.

1.3.2 Bestehende Arbeiten

Im Folgenden wird auf erwähnenswerte Arbeiten zum Thema statische Code-Analyse mit ML eingegangen.

Einen guten Einstieg bietet ein Artikel von Chris Lewis und Rong Ou [12], welcher auf die Bug-Prediction von Google eingeht. Ihr Ziel ist es primär, sogenannte "Hot-Spots" in einer Code-Base zu finden, also Komponenten, welche speziell fehleranfällig sind, um Entwickler warnen und zur besonderen Vorsicht mahnen zu können.

Lewis und Ou weisen darauf hin, dass Rahman et al. [13] einen einfachen, aber sehr wirksamen Algorithmus gefunden haben, der fast so gut performt wie komplexere Algorithmen. Die Hot-Spots werden gefunden, indem die Source-Code-Dateien nach der Anzahl vergangener Bug Fixing Commits geordnet werden. Der Nachteil dieser Methode ist, dass sie schlecht auf Änderungen reagieren kann. Auch wenn das Entwicklungsteam den Hot-Spot löst, werden diese Dateien noch immer hoch in der Liste erscheinen. Als Lösung entwickelten die Google-Ingenieure eine Formel, die bewirkt, dass ältere Commits weniger gewichtet werden. In unserer Arbeit wird dieses Mass inklusive der Zeitabhängigkeit ebenfalls aufgegriffen und als "temporale Features" implementiert (siehe Kapitel 3.4.5).

Als Vorbild für weitere temporale Features, wie z. B. die Anzahl Änderungen pro Zeiteinheit, diente die Arbeit von Nagappan und Ball [14] resp. Giger et al. [15]. Beide versuchen den "Code Churn", also die Menge von vergangenen Änderungen an einer Komponente, für die Fehlervorhersage zu verwenden. Letztere versuchen zusätzlich, Änderungen mit einer noch feineren Granularität zu untersuchen. Damit soll unterschieden werden, ob beispielsweise nur ein Fileheader geändert wurde, welcher sicher keine Fehlerquelle sein kann, oder kritische Stellen. Die Umsetzung dieser Erkenntnis würde den Rahmen unserer Arbeit sprengen, Giger et al. haben aber gezeigt, dass die Masse sehr gute Ergebnisse liefern können.

Relevanz hat auch Barstad et al. [16]. Mit ihrer Untersuchung prüften sie, ob es möglich ist, Vorhersagen mit statischer Code-Analyse und ML über die Qualität von Source Code zu treffen. Dabei unterscheiden Barstad et al. zwischen "well written" und "badly written". Sie schlagen dazu ein Eclipse-Plugin vor und setzen auf eine Kombination aus Peer Reviews, statischer Code-Analyse und Klassifikationsmethoden. Als Trainingsdaten nutzen sie Datensätze des PROMISE-Forums [17] sowie Hand-Ins von Studenten. Das PROMISE-Forum ist eine internationale Konferenz für Predictive Models und Data Analytics im Bereich Software Engineering, welche diverse Datensätze von annotierten Daten zur Verfügung stellt [18]. Leider basieren die meisten Datensätze auf relativ altem C-Code und scheinbar sind nur die verarbeiteten Features, nicht der ursprüngliche Source Code, hinterlegt. Dadurch ist es für unsere Zwecke nicht zweckmässig. Wie die Code-Qualität der Hand-Ins bewertet wurde, wird von Barstad et al. nicht genau erläutert.

Barstad et al. setzen auf Komplexitätsmasse nach McCabe [19] und Halstead [20], welche in unserer Arbeit ebenso miteinbezogen werden (siehe Kapitel 3.4.3). Eine weitere Parallele findet sich darin, dass hier ebenfalls mit einer zentralen MySQL-Datenbank gearbeitet wurde. Die Datenbank (DB) dient als Zwischenspeicher für Features, welche anschliessend von einem Klassifikationsalgorithmus verarbeitet werden. Die besten Resultate erhalten sie mit einer naiven Bayes-Klassifikation. Sie erkennen damit insbesondere die "well-written" Methoden

gut, die "badly-written" werden je nach Datenset höchstens zur Hälfte richtig erkannt.

Sharafat et al. [21] versuchen, nicht Fehler, sondern generell Änderungen in einer Codebasis vorherzusagen. Die dafür eingesetzten Metriken erwiesen sich für diese Arbeit als relevant. Zu den Metriken gehören unter anderem Komplexitätsmasse, Lines of Code, Anzahl Codeobjekte etc. Ebenfalls interessant sind Dependency-Metriken, welche sie aus UML-Diagrammen beziehen. Ein solcher Ansatz wird von unserer Arbeit aber nicht abgedeckt.

Rhaman et al. [22] vergleichen die Effektivität von statischer Code-Analyse, welche nur den Source Code analysiert, mit einer statistischen Analyse von historischen Daten der Codebasis (hier temporale Features). Dabei stellen sie fest, dass die Performance beider Ansätze je nach Problemstellung vergleichbar ist und dass in bestimmten Situationen eine Kombination die besten Resultate liefern kann. Mit unserer Arbeit verfolgen wir unter anderem ein ähnliches Ziel, nämlich ein Vergleich verschiedener Feature-Gruppen. Dies gerade auch, weil Rhaman et al. darauf hinweisen, dass ihre Befunde vermutlich nicht generalisierbar sind, da sie stark von den als Testdaten verwendeten Projekten abhängen könnten [23]. Es ist anzunehmen, dass dies auch für unsere Arbeit gilt.

Ebenfalls fassen Rhaman et al. eine wichtige Unterscheidung zusammen: Das Finden von Bugs (bei Rhaman et al. sog."Static Bug-Finding") ist eine andere Aufgabe als das Vorhersagen von Bugs (bei Rhaman et al. sog. "Defect Prediction"). Für ersteres werden vor allem Pattern-Matching und statische Data-Flow-Analyse eingesetzt (z. B. mit den bereits beschriebenen Tools *PMD* und *FindBugs*). Für die Vorhersage hingegen, so glauben sie, bieten sich statistische Methoden wie ML sehr an, was von Arisholm et al. [24], D'Ambros et al. [25] und Lesmann et al. [26] unterstützt wird.

Inspirierend für das Feature Design war für uns die Masterarbeit von Liljeson und Mohlin [27], welche sich ebenfalls mit Fehlervorhersage mittels ML beschäftigt. Das Ziel von Liljeson und Mohlin war, die Effektivität der Metriken von Source Code mit denen des zugehörigen Testcodes zu vergleichen. Sie schliessen darauf, dass die Kombination von Source- und Testcode-Metriken zu schlechteren Resultaten führte als Source-Code-Metriken alleine. Der F-Score der Source-Code-Metriken betrug 0.647. Interessanterweise erzielten die Testcode-Metriken alleine immerhin 0.578, was bemerkenswert ist, da diese nur über male Informationen über den Source Code verfügten. Dies zeigt, dass Source- und Testcode sehr eng miteinander gekoppelt sind.

Im Zuge ihrer Arbeit nutzten Liljeson und Mohlin eine Vielzahl von Code-Metriken als Features, allesamt gut beschrieben, was sich als sehr wertvoll erwiesen hat. Zu den Metriken gehören:

- Lines of Code
- Komplexitäts- und objektorientierte Metriken
- Dependency-Metriken
- Prozessmetriken (temporale Metriken)
- Testmetriken

Dependency-Metriken, nach Schröter et al. [28], wären für uns interessant gewesen, konnten aufgrund technischer und zeitlicher Einschränkungen von uns aber nicht umgesetzt werden (vgl. Kapitel 3.4.7).

Um die grosse Feature-Auswahl handhaben zu können, setzten sie auf Feature-Selection. Dieses Verfahren versucht, die Anzahl Features in einem Datenset so zu reduzieren, dass möglichst wenig Information verloren geht. Als Trainings- und Testdaten verwendeten Liljeson und Mohlin ein einziges internes Projekt ihres Industriepartners *Ericsson*. Um die Metriken zu extrahieren, nutzten sie grösstenteils ein selbst entwickeltes Toolset. Für die ML-Komponente wurden diverse Algorithmen verwendet, darunter Naive Bayes, AdaBoost, Bagging und Random Forest, wobei letzterer am Besten performte. Es sei aber darauf hingewiesen, dass Liljeson und Mohlin ein Klassifikationsproblem untersuchten. Unsere Arbeit beschränkt sich hingegen auf ein Regressionsproblem.

Ein Ziel unserer Arbeit war, typische Metriken aus der Textanalyse für die Fehlervorhersage einzusetzen. Insbesondere N-Grams bewerteten wir als vielversprechend. Tatsächlich fanden sich keine Arbeiten, welche dies bereits versucht hätten.

In seiner Masterarbeit implementierte Devin Chollack [29] eine Lösung, welche Softwarefehler mittels N-Grams lokalisieren soll. Dabei setzte er aber kein klassisches ML ein, sondern nutzte N-Grams, um automatisch eine Regelsammlung aus dem Kontrollfluss eines Programms zu generieren. Neuer Programmcode kann dann auf Verstösse dieser Regeln überprüft werden. Folgt z. B. in vielen Fällen auf einen Methodenaufruf A entweder ein Aufruf B oder C, dann wird dies als Regel $A(B|C)$ erkannt. Eine Abfolge ABD würde dann als Fehler gewertet werden. Wie effektiv dieser Ansatz ist, lässt sich anhand dieser Arbeit schlecht bewerten, da die verwendete Datenbasis nicht umfassend annotiert wurde.

Einen ähnlichen, regelbasierten Ansatz behandelten Nessa et al. [30]. Ausserdem finden sich einige Arbeiten wie die von Choi et al. [31] und Zhang et al. [32], welche N-Grams einsetzen, um Schadsoftware zu finden. Choi et al. bearbeiten aber nur kleinere Code-Abschnitte, wie sie z. B. in Injection- oder XSS-Angriffen eingesetzt werden. Diese sind schwierig mit Java-Klassen zu vergleichen. Zhang et al. analysieren kompilierte Executables, was ebenfalls nicht mit unserer Arbeit vergleichbar ist.

Auf der Suche nach geeigneten Trainings- und Testdaten stiessen wir auf die Mining Software Repositories (MSR)-Konferenz [33]. Diese befasst sich mit dem Analysieren der vielen Daten, die ein Softwarerepository bietet. Damit sollen interessante und nützliche Informationen über Softwaresysteme und -projekte gefunden werden. Das gesetzte Ziel der MSR für das Jahr 2016 befasst sich mit dem Repository Mining von grossen Versionsverwaltungssystemen wie GitHub [34] und SourceForge [35]. Dies scheint vielversprechend zu sein und die Resultate könnten in eine zukünftige Arbeit über Fehlervorhersage einfliessen.

Dyer et al. [36] beschreiben in ihrer Arbeit die Sprache *Boa* [37], welche als Abfragesprache für die gesammelten Repository-Daten der *MSR 2016* dient. Die Infrastruktur basiert auf *Hadoop*, Anfragen werden direkt in der Cloud bearbeitet. Der Einsatz von Hadoop erwies sich für unsere Arbeit aber als zu aufwändig. Des Weiteren schien die *Boa* Umgebung sehr komplex und Issue-Tracking-Daten (Enhancements oder Bugs) sind in der Datensammlung nicht vorhanden.

Andere Code Repositories, wie zum Beispiel das Software Artifact Infrastructure Repository [38], beinhalten eine Menge an kleinen Projekten mit verschiedenen Metriken. Für ML stellt allerdings eine grosse Datenmenge die Grundvoraussetzung dar. Auch sind die in diesem Repository enthaltenen Projekte

teilweise stark veraltet, weshalb sie für uns nicht in Frage gekommen sind.

1.4 Zielsetzung

Viele Tools zur Bug-Vorhersage verwenden ausgewählte und kategorisierte Lern- und Testdaten. Diese Arbeit soll ein Toolset bereitstellen, dass es erlaubt, Daten direkt aus Git-Repositories und Issue-Tracking-Systemen zu ziehen um anschliessend projektbezogen ein ML-Modell zu trainieren. Neben der offensichtlichen Komponente für Machine Learning muss ein weiteres Tool zum Sammeln und Strukturieren der Projektdaten entwickelt werden.

Die Literaturrecherche (Siehe Kapitel 1.3.2) lässt erkennen, dass es bereits viele Arbeiten über Fehlervorhersage mit Hilfe von statischen Metriken und ML gibt. Die gängigsten Metriken sind auch in dieser Arbeit wiederzufinden. Im Unterschied zu den bestehenden Arbeiten werden zusätzlich Textanalyse-Features wie N-Grams genutzt, welche aus einer Abstraktion des Java-Source-Codes generiert werden.

Der Ansatz, statische Code- und Textanalyse-Features zu kombinieren, wurde in anderen Arbeiten bisher nicht verfolgt. Deshalb soll geprüft werden, ob diese Kombination mit dem projektbezogenen Lernen eine hinreichende Fehlervorhersage gewährleistet.

Eine weitere Zielsetzung war es, den Grundstein für folgende, auf dieser Arbeit aufbauen Projekte zu legen. Dafür soll zwar kein schlüsselfertiges Tool entwickelt werden, sondern vielmehr Prototypen, auf denen aufgebaut werden kann. Deshalb muss die Wiederverwendbarkeit der Tools gewährleistet sein. Auch das Verfahren der Datensammlung soll für zukünftige Arbeiten nützlich sein. Die Architektur der entwickelten Tools muss erweiterbar sein, so dass neue Features einfach hinzugefügt werden können.

1.4.1 Terminologie

Der Terminus “Bug” oder “Softwarefehler” wird in der folgenden Arbeit mehrmals verwendet. Unter einem Bug wird ein vom Benutzer erstellter Issue im Issue-Tracking-System verstanden, der als Bug oder Defekt getaggt ist.

Wird ein Softwarefehler behoben, geschieht dies in der Regel durch Anpassung des Source Codes. Der Programmierer erstellt in der Versionsverwaltung einen Commit, welcher die Änderungen beinhaltet und pusht sie anschliessend auf den Server. Der erstellte Commit enthält in der Regel eine Nachricht, welche die Anpassungen möglichst genau beschreibt. Nach üblichen Konventionen verweist der Programmierer in der Nachricht auch auf den bearbeiteten (Bug-) Issue. Diese Assoziation machen sich die Tools dieser Arbeit zunutze, um Code (Inhalt des Commits) einem Bug zuordnen zu können. Dies hat zur Folge, dass unser Toolset nur bei fachgerechtem Gebrauch des Versionsverwaltungssystems funktioniert.

Ebenfalls anzumerken ist, dass als Bug getaggte Commits in der Regel keinen Bug-behafteten Code enthalten, sondern Code, welcher den Bug beheben soll. Deshalb kann aber davon ausgegangen werden, dass die Dateiversion *vor* dem Bugfix-Commit einen Bug enthalten muss.

Ein Vorteil dieser Art der Bug-Erfassung ist, dass Bugs nur gezählt werden, wenn auch tatsächlich eine Code-Anpassung stattgefunden hat. Damit werden

als Bug getaggte Issues, welche eigentlich bloss Scheinfehler waren, automatisch ignoriert.

1.4.2 Bewertung

Um die Qualität unseres Modells bewerten zu können, trainierten und testeten wir es mit Dateiversionen aus verschiedenen Zeitspannen eines Softwareprojekts. Das Ziel war, die Anzahl Bugfixes vorherzusagen, welche für eine Dateiversion in einer gewissen Zeitspanne (z. B. 6 Monate) auftreten.

Als Referenzwert (Baseline) dienten uns der Median und das arithmetische Mittel über die Bugfixes pro Zeitraum im Trainingsset (i. d. R. nahe bei 0), sowie einen entsprechend den Trainingsdaten gewichteten Zufallswert.

Das Ziel war es, zu überprüfen, ob mit unseren Methoden qualitativ bessere Aussagen als mit den Baselines gemacht werden können.

2 Theoretische Grundlagen

Machine Learning (ML) ist ein Gebiet der Informatik, welches sich mit selbstlernenden Programmen befasst. Anstatt einfach Beispiele auswendig zu lernen, erkennt ML Muster und Gesetzmässigkeiten in den Lerndaten und kann diese auf unbekannte Daten anwenden. Tom Mitchell beschreibt ML treffend:

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E" [39]

Die Erfahrung E ist im Falle unserer Arbeit eine Sammlung von Dateiversionen eines Java-Projektes. Diese werden, nach entsprechender Verarbeitung, in zwei Datensets unterteilt: Ein Trainingsset und ein Testset. Die Aufgabe T unseres Modells ist, die kommenden Bugfixes einer Dateiversion vorherzusagen. Um diese Aufgabe zu erfüllen, trainiert das Modell mit dem Trainingsset. Die anschliessende Messung der Performance erfolgt durch das Anwenden des Modells auf das Testset. Als Mass der Performance P dienen uns verschiedene Regressionsmetriken, wie sie im Kapitel 2.2 beschrieben werden.

Im Folgenden werden einige grundlegende Konzepte aus den Bereichen Machine Learning und Stochastik, welche in dieser Arbeit aufgegriffen werden, genauer erläutert. Dabei wird auch darauf eingegangen, inwiefern diese für unsere Arbeit relevant sind und worauf beim Einsatz mit dem gewählten Framework *Scikit-Learn* (siehe Kapitel 3.5.2) zu achten ist.

2.1 Regressions-Modelle

Die Regressionsanalyse ist ein in der Statistik sehr häufig eingesetztes Verfahren, welches versucht, die Abhängigkeit einer abhängigen Variable von einer oder mehreren unabhängigen Variablen zu untersuchen. Dabei lassen sich die Zusammenhänge nicht nur quantitativ beschreiben, sondern es lassen sich auch Prognosen für die abhängige Variable treffen [40].

Das Ziel dieser Arbeit ist, die Anzahl in einem bestimmten Zeitraum gefundene Bugs in einer Java-Datei vorherzusagen. Das heisst, die abhängige Variable ist die Anzahl Bugs. Die unabhängigen Variablen sind die verschiedenen Features, welche wir aus dem Source Code und der File-History extrahieren.

In den folgenden Kapiteln sollen die für diese Arbeit verwendeten Modelle der Regressionsanalyse erläutert werden.

2.1.1 Lineare Regression

Eines der einfachsten Modelle der Regressionsanalyse ist die lineare Regression. Dabei wird versucht, eine lineare Kurve so auf ein Trainings-Datenset von beobachteten Werten anzupassen, dass sie möglichst nahe an den Datenpunkten verläuft (siehe z. B. Abbildung 1).

Dies wird in der Regel mit der Methode der kleinsten Quadrate erreicht, einem Optimierungsproblem, bei dem die Quadrate der Residuen (Differenz zwischen Beobachtung und Modell) minimiert werden [42]. Das heisst, für ein Model mit den Koeffizienten $\omega = (\omega_1, \ldots, \omega_p)$ wird mathematisch folgendes

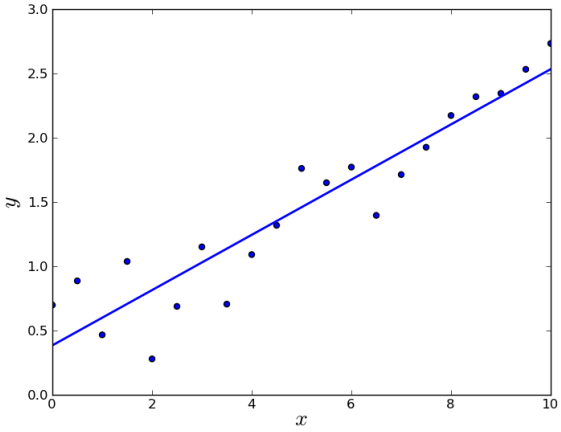

Abbildung 1: Beispiel einer Linearen Regressionskurve. x ist die unabhängige Variable, y die abhängige [41].

Problem gelöst:

$$\min_{\omega} = ||X\omega - y||_2^2$$

An dieser Stelle soll nicht darauf eingegangen werden, mit welchen Optimierungsverfahren die idealen Koeffizienten gefunden werden können, da dies den Umfang dieser Arbeit sprengen würde. Als Beispiel eines einfachen, numerischen Verfahrens sei das Gradientenverfahren genannt [43].

Probleme bei Linearer Regression können auftreten, wenn zwei oder mehr abhängige Variablen vollständig korrelieren. Das kann zum Beispiel auftreten, wenn ein Feature doppelt in das Datenset aufgenommen wird. Dies führt dazu, dass sich der Rang der Daten-Matrix verringert und sie nicht mehr regulär ist. In diesem Fall gibt es keine eindeutige optimale Lösung des Problems und das Modell wird sehr anfällig auf zufällige Fehler in den Trainingsdaten, was zu einer hohen Varianz führt [44].

In dieser Arbeit ist dies ein Problem, da zwischen unseren Features durchaus lineare Zusammenhänge bestehen könnten: Zum Beispiel ist es anzunehmen, dass die Anzahl hinzugefügter Dateien über verschiedene Zeiträume (siehe Kapitel 3.4.5) mehr oder weniger linear ist. Das ist einer der Gründe, warum lineare Regression für die Anwendung in dieser Arbeit ungeeignet ist. Das Modell wird aber trotzdem von der ML-Pipeline unterstützt, da es gerade für kleine Datensets vergleichsweise sehr schnell und dementsprechend nützlich für die Entwicklung der ML-Pipeline ist.

Scikit-learn bietet lineare Regression mit der Methode der kleinsten Quadrate (ordinary least squares) als *sklearn.linear_model.LinearRegression* an. Wenn die Daten-Matrix von der Grösse (n, p) ist, dann lernt das Modell mit einer Komplexität von $\mathcal{O}(np^2)$, angenommen dass $n \leq p$ [42].

2.1.2 Ridge Regression und polynomiale Features

Ridge Regression adressiert einige der Probleme von einfacher Linearer Regression indem ein Regularisierungsparameter eingeführt wird.

Lineare Regression kann ausschliesslich lineare Modelle abbilden. Falls der Zusammenhang zwischen abhängigen und unabhängigen Variablen also nicht linear ist, wird ein solches Modell entsprechend schlechte Resultate liefern. Eine Möglichkeit zur Abhilfe ist, die unabhängigen Variablen so zu transformieren, dass ein lineares Modell anwendbar ist. Ein sehr effektives Verfahren ist das Anwenden von *polynomialen Features*. Dabei wird der Zusammenhang zwischen abhängigen und unabhängigen Variablen als Polynom n-ten Grades dargestellt. So wird zum Beispiel das Modell

$$y = \omega_0 + \omega_1 x_1 + \omega_2 x_2$$

erweitert um polynomiale Features 2. Grades:

$$y = \omega_0 + \omega_1 x_1 + \omega_2 x_1^2 + \omega_3 x_1 x_2 + \omega_4 x_2^2 + \omega_5 x_2$$

Dieses Verfahren wird zum Teil auch als separate Form namens *polynomial regression* behandelt. In der Implementation der ML-Pipeline dieser Arbeit wird es aber als reine Transformation des Datensets angesehen.

Durch das Multiplizieren von Features mit sich selbst und Anderen können sehr komplexe Probleme gelöst werden. Tatsächlich ist dieses Verfahren so mächtig, dass es sehr anfällig auf Overfitting (Überanpassung) ist. Das bedeutet, dass das resultierende Modell zu stark auf die Trainingsdaten angepasst ist und als Konsequenz schlecht auf neue (Test-)Daten generalisiert (vgl. Abbildung 2).

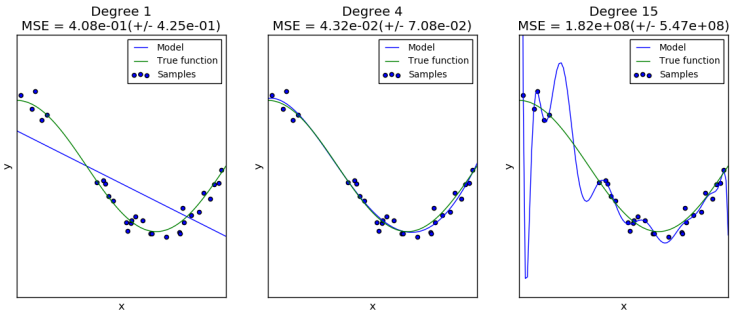

Abbildung 2: Links: Underfitting, das Modell bildet die Trainingsdaten schlecht ab. Mitte: Gutes Modell. Rechts: Overfitting, Modell bildet die Trainingsdaten sehr gut ab, generalisiert aber schlecht [45].

Aus diesem Grund ist eine Form von Regularisierung nötig. Regularisierung ist ein einfacher mathematischer Trick, welcher die Koeffizienten stabilisiert indem ein Regularisierungsterm eingeführt wird, welcher zu hohe Koeffizienten "bestraft". Mit diesem Term wird das Minimierungsproblem der Linearen Regression erweitert:

$$\min_{\omega} = ||X\omega - y||_2^2 + \alpha||\omega||_2^2$$

Gesteuert wird die Regularisierung über den Parameter $\alpha \geq 0$: Je höher α, desto grösser ist die "Bestrafung" und desto robuster werden die Koeffizienten gegenüber Kollinearität. Diese Form von Regularisierung wird in der Literatur z.T. auch *Tikhonov regularization* genannt. Der Regularisierungsparameter α wird oft auch mit dem Symbol λ dargestellt. [42], [46]

Wie ein sinnvoller Wert für α bestimmt werden kann, wird im Kapitel 2.1.4 beschrieben.

Ridge Regression wurde wie lineare Regression hauptsächlich in der Entwicklungsphase dieser Arbeit eingesetzt. In Kombination mit polynomialen Features wäre es zwar theoretisch geeignet, in der Praxis wird aber bei grossen Datensets die schlechte Skalierbarkeit ein Problem.

In *scikit-learn* lässt sich über *sklearn.linear_model.Ridge* ein Ridge Regression Modell instanzieren. Ridge Regression hat dieselbe Komplexitätsklasse wie lineare Regression.

Eine Anmerkung zu polynomialen Features: Werden die Trainingsdaten polynomial transformiert, müssen zwingend auch die Testdaten entsprechend transformiert werden, damit das Modell anwendbar ist. Ein wesentlicher Nachteil ist, dass die Anzahl der Output-Features (und damit die Anzahl zu optimierender Koeffizienten) polynomial mit der Anzahl Features und exponentiell mit dem Grad des Polynoms ansteigt und dementsprechend die Laufzeit und den Speicherbedarf massiv erhöht. Ausserdem wird es mit polynomialen Features schwieriger, Schlüsse aus dem Modell zu ziehen, da die Koeffizienten "verzerrt" werden.

2.1.3 Support Vector Regression

Support Vector Regression (SVR) ist ein auf Support Vector Machines (SVMs) basierendes Regressionsmodell, einem Lernalgorithmus welcher üblicherweise für Klassifizierungsprobleme eingesetzt wird.

Eine SVM versucht Datenpunkte zu klassifizieren, indem es den Vektorraum mit einer optimalen Hyperebene (engl. hyperplane) als Klassengrenze teilt. Dies bewerkstelligt sie so, dass ein möglichst breiter Bereich (engl. margin) um die Hyperebene frei von Objekten bleibt. So wird erreicht, dass die Klassifizierung möglichst robust gegenüber Rauschen und Ausreissern bleibt. Eine solche Hyperebene wird auch "maximum-margin hyperplane" genannt (siehe Abbildung 3).

Jeder Datenpunkt wird durch einen (Feature-)Vektor repräsentiert. Für das Finden der optimalen Hyperebene müssen nur die ihr am nächsten liegenden Datenpunkt-Vektoren, also die namensgebenden "Support Vektoren", berücksichtigt werden. Die restlichen Vektoren sind gewissermassen verdeckt und sind nicht weiter relevant, was einen wesentlichen Vorteil bezüglich Laufzeit und Speicherbedarf darstellt [47].

Da Hyperebenen definitionsgemäss linear sind, lassen sich mit einer solchen SVM nur linear trennbare Daten klassifizieren. Um SVMs auch auf linear unseparierbare Daten anzuwenden, wird der sogenannter Kernel-Trick angewendet. Dabei wird der Vektorraum in einen höherdimensionalen Raum überführt. Ziel ist es, eine Transformation zu finden, welche das Datenset linear separierbar macht. Im Falle des Beispiels in Abbildung 4 wurde die Transformation $T([x_1, x_2]) = [x_1, x_2, x_1^2 + x_2^2]$ angewandt.

Dies ist ein ähnlicher Ansatz wie polynomiale Features bei Ridge Regression und er führt zum selben Problem. Die Feature-Matrix wird bei hohen Dimensionen unbewältigbar gross. Es stellt sich aber heraus, dass zum Trainieren einer SVM lediglich das paarweise Skalarprodukt $\langle \vec{x_i}, \vec{x_j} \rangle$ der Trainingsdaten benötigt wird [49]. Nun existieren sogenannte Kernel-Funktionen, welche das Skalarprodukt zweier Vektoren in höherdimensionalen Räumen berechnen können, ohne

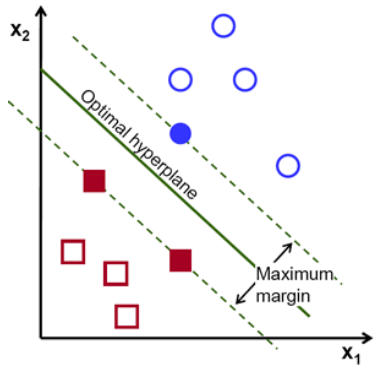

Abbildung 3: Optimale Hyperebene [47].

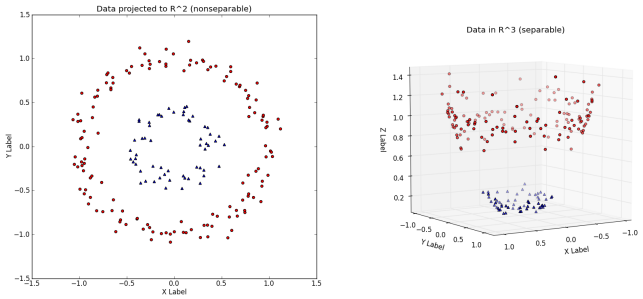

Abbildung 4: Links: ein linear nicht separierbares Datenset in $\mathbb{R}^2$. Rechts: Dasselbe Datenset, transformiert in einen höheren Raum ($\mathbb{R}^3$): Es wird linear separierbar [48].

die Vektoren explizit in diesen Raum zu transformieren. Das führt dazu, dass in der SVM einfach alle Skalarprodukte mit der entsprechenden Kernel-Funktion ersetzt werden können. Der Algorithmus benötigt damit keinen zusätzlichen Speicher und auch der zusätzliche Rechenaufwand hält sich in Grenzen. Dieser ist lediglich abhängig von der benutzten Funktion und kann dementsprechend sehr klein sein [48].

scikit-learn bietet folgende populäre Kernel-Funktionen an:

1. **Linear**: $\langle \vec{x_i}, \vec{x_j} \rangle$, entspricht dem normalen Skalarprodukt, also keiner speziellen Kernel-Funktion.

2. **Polynomial Kernel**: $(\gamma \cdot \langle \vec{x_i}, \vec{x_j} \rangle + r)^d \cdot d$, wobei d der Grad des Polynoms ist und r der Bias. Diese Werte sind über den Parameter *degree* respektive *coef0* konfigurierbar.

3. **Radial Basis Function (RBF) Kernel**: $\exp(-\gamma \cdot |\vec{x_i} - \vec{x_j}|^2)$, wobei der Parameter γ über das Keyword *gamma* konfiguriert wird.

4. **Sigmoid Kernel**: $\tanh(\langle \vec{x_i}, \vec{x_j} \rangle + r)$, bei dem ebenfalls die Parameter *gamma* und *coef0* relevant sind.

Ausserdem lassen sich in *scikit-learn* auch eigene Kernel-Funktionen definieren und einbinden. Das finden eines geeigneten Kernels für ein Problem und

der passenden Parameter ist keine triviale Aufgabe, kann aber massgeblich zur Performance des Modells beitragen [48], [50].

Neben den Parametern der Kernel-Funktionen muss auch immer noch der Regularisierungsparameter $C > 0$ spezifiziert werden. Ein hoher Wert für C erlaubt ein komplexeres Modell, das sich stark an die Trainingsdaten anpassen kann, aber auch mehr Rechenzeit benötigt. Ein niedriger Wert hingegen glättet die Entscheidungskante und verringert damit Overfitting. C entspricht damit dem Kehrwert des α-Parameters der Ridge Regression.

Da in dieser Arbeit kein Klassifizierungsproblem gelöst wird, ist SVM ungeeignet. Deshalb verwenden wir eine Variante namens Support Vector Regression, kurz SVR. Wie bei SVMs wird auch hier nur eine Teilmenge der Trainingsdaten verwendet. Alle beschriebenen Kernels können auch mit SVR genutzt werden [50].

2.1.4 Cross Validation

Parameter wie etwa C bei SVM oder α bei Ridge Regression sind massgeblich für die Performance eines Modells. Deshalb ist die Suche nach den bestmöglichen Parametern für ein Modell eine wichtige Aufgabe.

Eine Möglichkeit, diese Suche zu automatisieren, ist das Grid-Search-Verfahren. Dabei wird ein Raum (oder Grid) von Parametern aufgespannt, in dem alle Kombinationen durchprobiert werden. Am Ende werden die Parameter mit den besten Resultaten ausgewählt.

scikit-learn ermöglicht dies mit der Klasse *sklearn.grid_search.GridSearchCV*, welche mit dem Parameter *param_grid* eine Liste von Parameter-Grids entgegennimmt. Möchte man Beispielsweise ein SVM-Modell mit einem linearen und einem RBF-Kernel optimieren, könnte das Grid so aussehen [51]:

Listingsverzeichnis 1: Parameter Grid

```
param_grid = [
  {
    'C': [1, 10, 100, 1000],
        'kernel': ['linear']
  },
  {
    'C': [1, 10, 100, 1000],
    'gamma': [0.001, 0.0001],
    'kernel': ['rbf']
  },
]
```

Mit diesen Parameterkombinationen wird dann auf dem Trainingsset gelernt. Es wäre falsch, die Qualität der Parameter mit Hilfe des Testsets zu messen, da sie sich damit auf den Testdaten optimieren würden und anschliessend keine verlässlichen Angaben zur Qualität des Modells gemacht werden könnten. Aus diesem Grund sollte aus dem Trainingsset ein sogenanntes Validationsset abgespalten werden, um die Parameter damit zu testen.

Cross Validation (CV), auch Kreuzvalidierung, optimiert dieses Verfahren. In der einfachsten Ausführung, der sogenannten k-Fold CV wird das Trainingsset in k kleinere Sets aufgeteilt. Für jede der k "Folds" wird folgende Prozedur ausgeführt:

Symbol	Bezeichnung	Erklärung
$\mathbf{y}$	Ground Truth, beobachtete y-Werte	Der tatsächliche Messwert
$\hat{\mathbf{y}}$	Prediction, erklärte y-Werte	Werte, die durch das Regressions-Modell vorhergesagt ("erklärt") werden.
$\mathbf{e}$	Residuen, Fehler	Die Differenz zwischen beobachteten und erklärten Werten: $e = y - \hat{y}$
$\mathbf{Var}(\mathbf{X})$	Varianz	Kenngrösse der Wahrscheinlichkeitsverteilung einer Zufallsvariable. Entspricht der mittleren, quadrierten Abweichung der Werte von ihrem Durchschnitt: $Var(X) = \frac{1}{n}\sum(X - \bar{X})^2$

Tabelle 1: Relevante Begriffe der Statistik

1. Das Modell wird mit $k - 1$ der Folds trainiert;
2. das resultierende Modell wird dann mit dem verbleibenden Teil der Daten validiert (d. h. es wird als Testset zum Berechnen des Scores verwendet)

Das Performance-Mass, welches von der k-Fold CV zurückgegeben wird, ist dann der Mittelwert über alle Werte, welche in dieser Schleife berechnet wurden. Dieser Ansatz kann rechenintensiv sein, verschwendet aber nicht so viele Daten wie wenn ein separates Validationsset verwendet würde. Speziell wenn kleinere Datensets verwendet werden, ist dies ein grosser Vorteil [52].

In dieser Arbeit wird ausschliesslich k-Fold CV verwendet, insbesondere auch weil die Anwendung mit *scikit-learn* sehr einfach ist und ein separates Validationsset zusätzlichen Konfigurationsaufwand nach sich ziehen würde.

2.2 Regressions-Metriken

Um die Qualität eines Modells zu bewerten, können sich verschiedene Metriken aus der Statistik zunutze gemacht werden. Die für dieses Projekt verwendeten Metriken werden im Folgenden kurz beschrieben.

2.2.1 Allgemeine Begriffe der Statistik

In dieser Arbeit werden einige allgemeine Begriffe der Statistik mehrmals verwendet, die in Tabelle 1 kurz erläutert werden.

2.2.2 Varianzaufklärung

Die Varianzaufklärung (Explained Variance) ist ein Mass dafür, wie gut ein Regressionsmodell die Streuung, also die Varianz, eines Datensatzes erklären kann. Ist y die Ground Truth und $\hat{y}$ die Prediction, dann berechnet sich die

Varianzaufklärung aus dem Verhältnis der Varianz Var der Ground Truth und der Varianz der Residuen:

$$\text{explained\_variance}(y, \hat{y}) = 1 - \frac{Var\{y - \hat{y}\}}{Var\{y\}}$$

Das bestmögliche Resultat ist 1, es tritt genau dann ein, wenn die Varianz der Prediction gleichwertig zur Varianz der Ground Truth ist. Je tiefer der Wert, desto schlechter ist das Resultat bezüglich Varianz. Dabei kann die Varianzaufklärung bei beliebig grossem Unterschied auch negativ werden.

2.2.3 Mittlerer absoluter Fehler

Der mittlere absolute Fehler (Mean absolute Error, MAE) zeigt an, wie nahe die erklärten y-Werte an den Beobachteten liegen. Er entspricht dem Mittelwert der absoluten Residuen. Ist n die Anzahl Datensätze in y, berechnet er sich folgendermassen:

$$\text{MAE}(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} |y_i - \hat{y}_i|$$

Im Idealfall ist der mittlere absolute Fehler 0. Das bedeutet, dass die Prediction exakt der Ground Truth entspricht. Je höher das Resultat ausfällt, desto schlechter war die Vorhersage.

2.2.4 Mittlerer quadrierter Fehler

Der mittlere quadrierte Fehler (Mean squared Error, MSE) wird sehr ähnlich berechnet, wie der mittlere absolute Fehler. Anstatt des Betrags der Residuen, wird allerdings das Quadrat genommen:

$$\text{MSE}(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \hat{y}_i)^2$$

Wie beim mittleren absoluten Fehler ist das bestmögliche Resultat 0. Höhere Fehler fallen durch die Potenzierung schwerer ins Gewicht. Für uns ist der mittlere quadrierte Fehler nicht sehr relevant, da der mittlere absolute Fehler eine nachvollziehbarere, lineare Bewertung eines Modells ermöglicht. Dieser Score wird aber bei linearer Regression im Optimierungsterm verwendet, da die Quadrierung es dort erlaubt, einfacher abzuleiten (siehe Kapitel 2.1.1).

2.2.5 Median des absoluten Fehlers

Der Median des absoluten Fehlers (Median absolute Error, MDE) ist ähnlich wie der mittlere absolute Fehler, ist aber robuster gegen Ausreisser. Er entspricht dem Median aller absoluten Residuen:

$$\text{MDE}(y, \hat{y}) = median(|y_1 - \hat{y}_1|, \ldots, |y_n - \hat{y}_n|)$$

Hier ist ebenfalls der bestmögliche Wert 0. Zu beachten ist allerdings, dass ein Resultat von 0 nicht bedeutet, dass die Prediction exakt der Ground Truth entspricht, sondern dass mehr als die Hälfte der erklärten y-Werte der Ground Truth entsprechen. Wie beim mittleren absoluten Fehler kann das Resultat ebenfalls beliebig hoch und damit schlecht ausfallen.

2.2.6 Bestimmtheitsmass R^2

Das Bestimmtheitsmass (auch Determinationskoeffizient) ist der Anteil der Variation, welcher durch die lineare Regression erklärt wird. Damit dient es als wichtiges Mass der Güte der Anpassung und gibt an, "wie gut" ein statistisches Modell eine Menge von Beobachtungen erklären kann. Berechnet wird das Bestimmtheitsmass aus dem Verhältnis der Variation der Residuen und der Variation von y (n ist dabei die Anzahl Datensätze in y):

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n-1}(y_i - \bar{y})^2} = 1 - \frac{\text{unerklärte Variation}}{\text{Gesamtvariation}}$$

Ein perfekter linearer Zusammenhang ergibt ein Bestimmtheitsmass von 1. Auf dem Datenset, auf das die Regressionskurve angepasst wurde, kann das Bestimmtheitsmass minimal 0 werden, was keinem linearen Zusammenhang entspricht. Auf dem Testset kann das R^2 aber beliebig schlecht werden und damit auch negative Werte annehmen.

Nach Cohen [53] lässt sich aus dem Bestimmtheitsmass zusätzlich noch die Effektstärke f^2 berechnen:

$$f^2 = \frac{R^2}{1 - R^2}$$

Dabei gelten nach Cohen [53] folgende Werte, um eine Aussage über die Stärke des Effekts zu machen:

$f^2 = 0.02$	kleine Effektstärke
$f^2 = 0.15$	mittlere Effektstärke
$f^2 = 0.35$	grosse Effektstärke

Tabelle 2: Effektstärken in f^2

Wenn man dies auf das Bestimmtheitsmass R^2 zurückrechnet, ergeben sich folgende Werte [40]:

$R^2 = 0.0196$	kleine Effektstärke
$R^2 = 0.1300$	mittlere Effektstärke
$R^2 = 0.2600$	grosse Effektstärke

Tabelle 3: Effektstärken in R^2

Das heisst bei $R^2 > 0.26$ lässt sich bereits von einer grossen Effektstärke sprechen.

2.3 N-Grams

N-Grams sind das Ergebnis der Zerlegung eines Textes in Fragmente. Dabei ist N ein Platzhalter für die Anzahl zusammengehängter Fragmente. In der Textanalyse ist ein 1-Gram typischerweise ein Wort, ein 3-Gram entspricht einer Wortkette von 3 Wörtern.

N-Grams spielen in der Textanalyse eine entscheidende Rolle. Das Listing 2 soll ein Beispielsatz und dessen Unterteilung in N-Grams zeigen.

Listingsverzeichnis 2: N-Grams Beispiel

```
Text: Ich freue mich.

1-Grams: ich, freue, mich
2-Grams: ich_freue, freue_mich
3-Grams: ich_freue_mich
```

Eine häufige Anwendung ist die Sentimentanalyse von Texten. Dabei wird mit Hilfe von ML z. B. ein Satz entweder positiv oder negativ gedeutet. Ist im Trainingsset nun der im Listing 2 dargestellte Text mit positiver Deutung vorhanden, werden die einzelnen N-Grams als positiv interpretiert. Ein Text im Testset, welcher einige oder alle diese N-Grams enthält, wird somit auch als positiv gewertet. Anhand dieser Tatsache können N-Grams Ähnlichkeiten erkennen.

Im Paper von Mohammad et al. [54] werden Textanalyse-Features zur Sentimentanalyse von Twitter Nachrichten verwendet. Die N-Gram Features haben bei der Arbeit von Mohammad neben den Lexikon-Features den grössten Einfluss auf das Resultat. Dies unterstreicht das Potenzial von N-Grams.

3 Vorgehen

Dieses Kapitel beschreibt, wie wir vorgegangen sind um unser Ziel zu erreichen. Als erstes wird darauf eingegangen, wie das Gesamtsystem konzipiert wurde. Danach besprechen wir die Entwicklung der einzelnen Komponenten im Detail. Zum Schluss wird das Vorgehen beim Testen unserer Lösung beschrieben.

3.1 Grobkonzept und Anforderungen

Um unser Ziel zu erreichen, benötigten wir als erstes einen genügend grossen und aussagekräftigen Lerndatensatz. Ein solcher soll einerseits eine grosse Menge an Java Source Code Files umfassen, andererseits muss erkennbar sein, ob ein File von einem (oder mehreren) Bug(s) betroffen ist.

Der in der Arbeit verfolgte ML-Ansatz erforderte das Konzipieren und Implementieren von diversen Features. Dabei wurde auf einen Mix von Features gesetzt, der sich sowohl aus klassischen Code-Metriken wie auch aus Ansätzen der Textanalyse, insbesondere N-Grams, zusammensetze.

Ausserdem musste eine sinnvolle ML-Pipeline entwickelt werden. Diese sollte primär gute Ergebnisse liefern und sekundär in akzeptabler Laufzeit terminieren. Sinnvolle Analyse- und Konfigurationstools sollten sicherstellen, dass die Qualität der Resultate zielgerichtet optimiert und angepasst werden kann.

Aufgrund dessen wurde die Arbeit wie folgt unterteilt:

1. Erstellen eines Lerndatensatzes
2. Feature Design
3. Machine Learning

Diese Unterteilung war bestimmend für das Design der Lösung.

3.1.1 Lerndatensatz

Die erste Herausforderung war die Beschaffung von Lerndaten. Bestehende Datensammlungen (siehe Kapitel 1.3.2) wurden gefunden, jedoch erfüllte keine davon unsere Ansprüche. Die von der *MSR* ins Leben gerufene *Boa*-Projekt enthält leider keine Issue-Tracking-Informationen und erwies sich aufgrund seiner Komplexität als zu umfangreich.

Der Ansatz von *Boa*, sich nicht auf ein Zielprojekt zu beschränken, schien uns allerdings sinnvoll. Dies führte dazu, dass wir Tool benötigten, welches aus einem beliebigen Java-Projekt Lerndaten generieren kann.

Als Datenquelle bot sich Open-Source-Software an. Heutzutage gibt es nicht nur Unmengen von Open-Source-Projekten, darunter finden sich auch viele genügend grosse und ausgereifte Projekte. Dadurch, dass sie nicht nur frei verfügbar sind, sondern auch teils hunderte Mitautoren haben, sind solche Projekte auf mächtige Collaboration-Tools angewiesen. Allem voran sind effiziente Versionsverwaltungssysteme zentral, wie etwa *Git* [55]), *Subversion (SVN)* [56] oder *Concurrent Versions System (CVS)* [57].

Ebenfalls essentiell sind Issue-Tracking-Systeme, wie etwa *Bugzilla* [58] oder *JIRA* [59], welche Pendenzen und Bugs verwalten. Einige Plattformen vereinen

dabei gleich beides und sind dementsprechend beliebt und effizient für das Hosting von Open-Source-Projekten, als Beispiele seien *GitHub* [34], *SourceForge* [35], *CodePlex* [60] und *BitBucket* [61] genannt.

In dieser Arbeit liegt der Fokus auf der Versionsverwaltungssoftware *Git*. Den grössten Vorteil von *Git* sahen wir darin, dass es uns als dezentralisierte Versionsverwaltung erlaubte, einfach ein Repository mit kompletter History lokal zu speichern. Zentralisierte Versionsverwaltungen wie *CVS* oder *SVN* sind hingegen als Client-Server-System konzipiert und erfordern eine Serververbindung für den Zugriff auf die History, was für eine schlechte Performance beim Data Mining hervorgerufen und mehr Abhängigkeiten in der Implementation verursacht hätte.

Andere verteilte Versionsverwaltungen, wie etwa *Bazaar* [62] oder *Mercurial* [63] hätten diese Anforderung natürlich auch erfüllt, sind aber wesentlich weniger verbreitet [64]. *Git* bietet zudem offene und gut dokumentierte Schnittstellen und ist äusserst effizient und schnell. Ausserdem wird es auf allen modernen Unix-artigen Betriebssystemen sowie auf Windows gut unterstützt.

Als sinnvollste Quelle von *Git*-Repositories erachteten wir *GitHub*, da dort eine enorm grosse Anzahl an Open-Source-Projekten gehostet werden, darunter auch solche mit genügend grossem Umfang für unsere Anforderungen. Von *SourceForge* haben sich nach einigen Kontroversen [65], [66] viele Projekte zurückgezogen, *CodePlex* und *BitBucket* konnten uns hingegen keine genügend breite Auswahl an Projekten bieten (vgl. [67]).

Anhand eines Issue-Tracking-Systems, bei dem Bug-Reports als Issues (auch Tickets) erfasst werden, können fehlerbehaftete Dateien bestimmt werden. Hier fokussierten wir uns in erster Linie auf das integrierte Issue-Tracking von *GitHub* und auf *JIRA* von *Atlassian*.

Das Issue-Tracking von *GitHub* bot sich an, da die verwendeten Testprojekte sowieso primär von dieser Plattform bezogen wurden und die meisten der dort gehosteten Projekte auch dessen Issue-Tracking-System benutzen. Es hat ausserdem eine sehr einfache Issue-Struktur, was für uns das Handling einfach machte. *GitHub* bietet eine umfangreiche, aber gut dokumentierte REST-API an [68].

Mit *JIRA* unterstützten die Tools dieser Arbeit neben *GitHub* ein weiteres mächtiges Issue-Tracking-System mit grosser Verbreitung. Wie *GitHub* wartet auch *JIRA* mit einer mächtigen und bestens dokumentierten REST-API auf [69], was die Einbindung vereinfachte.

Alle diese Erkenntnisse und Entscheidungen flossen in die Implementierung eines Repository Mining Tools ein, mit dem wir für unsere Zwecke passende Daten sammeln konnten. Dieses Tool wird im Kapitel 3.2 genauer beschrieben.

3.1.2 Systemarchitektur

Beim Design der Systemarchitektur war insbesondere die Modularität wichtig. Obschon das Ziel dieser Bachelorarbeit nicht das Implementieren eines releasewürdigen Tools war, sollte dennoch eine solide Grundlage entstehen, auf dem spätere Projekte aufbauen können. Das erschien uns bei einem monolithischen Projekt wesentlich schwieriger als bei einer Sammlung von separaten Tools, welche zwar kombiniert werden können, aber auch alleinstehend nützlich sind.

Da sich aus unseren Anforderungen drei Hauptaufgaben herauskristallisiert haben, lag es nahe, unser System in drei einzelne Tools aufzuteilen:

1. Einen *Repository Miner*
2. Ein *Feature Extraction Tool*
3. Eine *Machine Learning Pipeline*

Diese Reihenfolge entspricht auch dem angedachten Arbeitsablauf:

Der *Repository Miner* ist dafür verantwortlich, die Rohdaten zu sammeln. Er durchforstet *Git*-Repositories und Issue-Tracking-Systeme um Statistiken und relevante Informationen zu sammeln, welche dann in der Datenbank gespeichert werden. Die Architektur des Repository Miners ist im Kapitel 3.2 genauer beschrieben.

In einem zweiten Schritt verarbeitet das *Feature Extraction Tool* die Rohdaten weiter und berechnet mit der Datengrundlage der Versionsverwaltung *Git* und dem Datenbankerzeugnis des Repository Miners diverse Features bzw. Metriken, welche ebenfalls in der Datenbank abgespeichert werden. Es wird im Kapitel 3.3 beschrieben.

Zum Schluss liest die *Machine Learning Pipeline* die benötigten Trainings- und Testdaten aus der Datenbank, wendet auf sie einen ML-Algorithmus an und bildet daraus ein Modell, auf dessen Basis dann Vorhersagen für neue Files getroffen werden können. Das Kapitel 3.5 widmet sich diesem Tool.

Die Kommunikation der einzelnen Tools erfolgt dabei über eine zentrale Datenbank. So werden die Ergebnisse jedes Schrittes strukturiert gespeichert und können zur Analyse einfach abgerufen werden. Dabei ist das Database Management System (DBMS) austauschbar, was das Design generell flexibler macht.

Die Einteilung in einzelne Tools und das Persistieren der Zwischenergebnisse in einer Datenbank brachte auch eine wesentliche Zeiteinsparung während der Entwicklung. Würden die Daten alle in einem Schritt und ohne Zwischenspeichern verarbeitet, würde jeder Durchlauf sehr viel Zeit in Anspruch nehmen. Mit dem Einsatz einer DB konnten während der Entwicklung der Tools auf bereits bestehende, konsistente Daten des jeweils vorangehenden Schrittes zugegriffen werden.

In Abbildung 5 wird die resultierende Systemarchitektur grob dargestellt.

3.1.3 Infrastruktur

ML mit einer grossen Anzahl an Features und Datensätzen ist sehr rechenintensiv. Die Berechnung auf einem der verwendeten Entwicklungscomputer durchzuführen war mangels genügend RAM und unzureichender CPU-Leistung nur begrenzt möglich. Uns war von Anfang an klar, dass die Projekte mit denen wir uns beschäftigen eine sehr grosse Anzahl Features generieren und deswegen auf einem dedizierten System ausgeführt werden müssen.

Die dafür genutzte Infrastruktur basiert auf einem *Ubuntu Server 15.10 64 Bit*. Das System wird auf einer virtuellen *VMWare vSphere 6.0* Infrastruktur ausgeführt und verfügt über 190 GB SSD Speicher (*CRUCIAL SSD 480 GB M500*). Dem System stehen 8 Kerne einer *Intel Xeon X5660 CPU* und 38 GB RAM zur Verfügung. Der Datenbankserver und die in dieser Arbeit entwickelten Tools sind auf dem selben System installiert. Eine Separierung wäre natürlich möglich, hätte aber aufgrund von Ressourcenknappheit auf unserem System keinen grossen Vorteil gebracht.

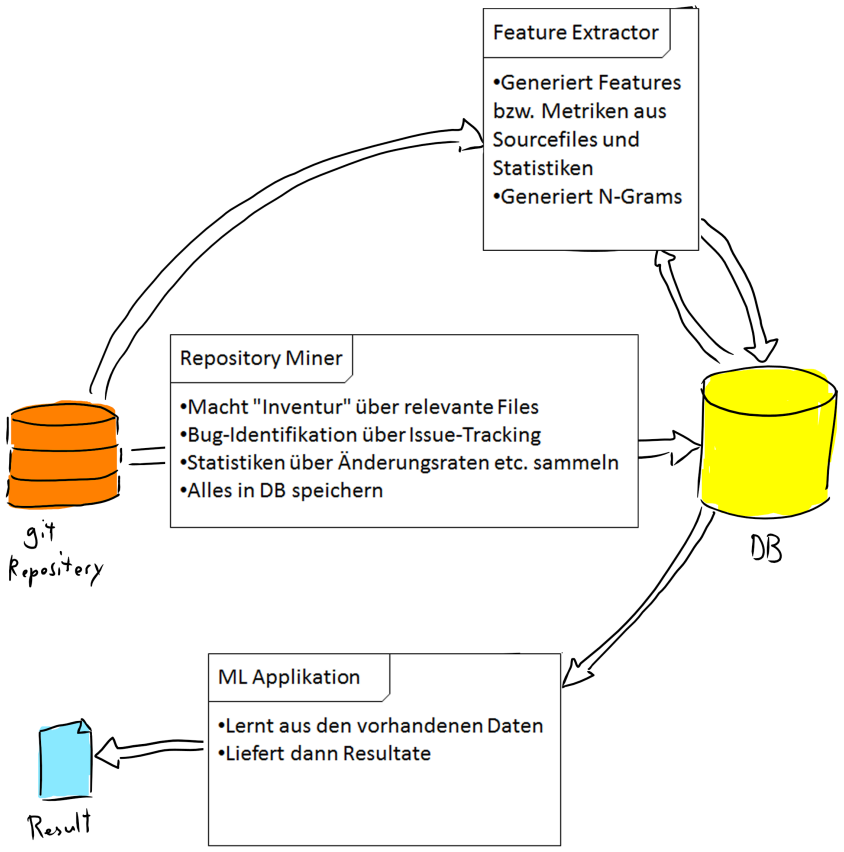

Abbildung 5: Grobübersicht des Systems

3.2 Repository Mining

Unser Repository Mining Tool erhielt das Akronym *GtSooG* als Namen, was für *Get the Stuff out of Git* steht. Die Hauptaufgabe von GtSooG besteht aus der Datenaufbereitung von *Git*-Repositories und Issue-Tracking-Systemen. Die gesammelten Daten werden in einer SQL Datenbank gespeichert und dienen später als Grundlage für den Feature Extractor.

Für diesen und das ML-Tool ist performanter Zugriff auf die Datenbasis ein wichtiger Faktor. Ein direkter Zugriff auf Online-Ressourcen wäre für eine schnelle Verarbeitung undenkbar.

Die Versionsverwaltung *Git* hat den Vorteil, das ein lokaler Klon eines Repositories dessen komplette Revisionsgeschichte beinhaltet. Damit kann dieser unabhängig vom Server betrieben werden. Die Speicherverwaltung von *Git* ist sehr effizient [70]. Mit Hilfe von Deduplizierungsverfahren werden redundante Code-Objekte nur einmal gespeichert und untereinander verlinkt. Da wir nicht davon ausgehen, dies ansatzweise so effizient lösen zu können, verzichten wir auf die Speicherung des kompletten Codes in der Datenbank. Die Filter- und Suchmechanismen von *Git* bieten allerdings nur eingeschränkte Möglichkeiten.

Eine Suche resultiert meistens im Absuchen des ganzen Repositories und ist dementsprechend langsam.

Informationen über Issues werden separat in einem Issue-Tracking-System gespeichert. Diese Systeme werden meistens auf einem Server betrieben und können nicht lokal kopiert werden. Aufgrund dieser Tatsache ist das Abrufen von Issues eine schleppende Angelegenheit. Für das Issue-Tracking-System *GitHub* kommt hinzu, dass der Zugriff auf die *GitHub*-API limitiert ist [71].

Die Aufgabe von GtSooG besteht darin, die oben erwähnten Limitierungen aufzuheben. Dies soll durch die strukturierte Speicherung der Metadaten von *Git* und des Issue-Trackings in einer relationalen Datenbank bewerkstelligt werden. Damit ist nicht nur ein effizienter Zugriff gewährleistet, mit der mächtigen und weitverbeiteten Structured Query Language (SQL) lassen sich auch beliebig komplexe Abfragen erstellen.

3.2.1 Anforderungen

Die oben erwähnten Überlegungen führen zu folgenden Anforderungen:

Funktional:

- Zugriff auf lokal gespeicherte *Git* Repositories
- Direkter Zugriff auf die Issue-Tracking-Systeme *GitHub* und *JIRA* über deren REST-API
- Datenspeicherung in einer SQL Datenbank
- Speichern aller Commits, inkl. Autor und Commit-Datum
- Speichern aller Dateien, welche in einem Commit enthalten sind
- Speichern der verschiedenen Dateiversionen
- Speichern der geänderten Codezeilen pro Dateiversion
- Speichern aller Issues und die Zuordnung ob es sich um einen Bug handelt.

Nichtfunktional:

- Modulare und erweiterbare Architektur
- Kommandozeilenbasiert
- Adäquates Errorhandling, so dass die Software bei einem nicht lesbaren Commit oder Issue die Arbeit fortsetzt (non-blocking)
- Die Ausführung von GtSooG soll jederzeit unterbrochen werden können, die gespeicherten Daten sollen trotzdem stets konsistent sein.
- Vermeiden redundanter Speicherung des Codes

Abgrenzung:

- Es ist eine singlethreaded Architektur angedacht, Multithreading ist keine Anforderung

3.2.2 Design

Da sich die Schnittstellen von *Git* und den verschiedenen Issue-Tracking-Systemen fundamental unterscheiden ist GtSooG in folgende Komponenten unterteilt:

- Der Repository Miner liest Daten direkt vom Dateisystem aus einem *Git*-Repository aus und vertraut dabei auf die Funktionalität der *gitpython* Library
- Der Issue Scanner liest Daten aus einem Issue-Tracking-System per REST-API aus

Die beiden Komponenten extrahieren folgende Informationen aus *Git* und den Issue-Tracking-Systemen:

- Name und Pfad des Repositories
- Alle Commits mit Mitteilung, Autor und Commit Datum
- Alle dem Commit angehängten Dateien mit Erstellungsdatum (entspricht Datum des Commits, mit dem sie erstellt wurden)
- Die verschiedenen Versionen der Dateien und die Anzahl von hinzugefügten und gelöschten Codezeilen
- Die hinzugefügten oder gelöschten Codezeilen pro Dateiversion
- Name und URL des Issue-Tracking-Systems
- Alle Issues mit Titel und der Information ob es sich dabei um einen Bug handelt
- Die Issues sollen den Commits zugeordnet werden können

Daraus resultierte das Datenbank Design in Abbildung 6. Die Felder *files_added* und *files_deleted* in der Tabelle *version* sind redundant und könnten über eine SQL-Join-Query abgefragt werden. Da die Performance beim Feature Extractor eine grosse Rolle spielt, werden diese Informationen aber als zusätzliches Feld in der Datenbank gespeichert. Dasselbe gilt für die Felder *added_files_count*, *deleted_files_count*, *renamed_files_count*, *changed_files_count* in der Tabelle *commit*.

3.2.3 Implementierung

Der Repository Miner ist in Python geschrieben. Die Software macht Gebrauch von den Python-Bibliotheken *request* [72], *GitPython* [73] und *SQLAlchemy* [74]. *GitPython* fungiert als Schnittstelle zum *Git*-Repository. Ursprünglich sollte der Repository Miner mit Multithreading den Datamining-Prozess beschleunigen. Nach einigen Versuchen und Online-Recherche stellte sich dieser Prozess jedoch als sehr schwierig bis unmöglich heraus. Einerseits sind die einzelnen Commits voneinander abhängig was eine parallele Verarbeitung sehr komplex macht. Andererseits stellte sich heraus, dass das Unterfangen bereits vom Projekt *gitpython/async* [75] erfolglos versucht wurde. Es war somit nicht realistisch, im Rahmen dieses Projekts eine sinnvolle Lösung zu finden. Deshalb wurde der Repository Miner singlethreaded implementiert. GtSooG liest die

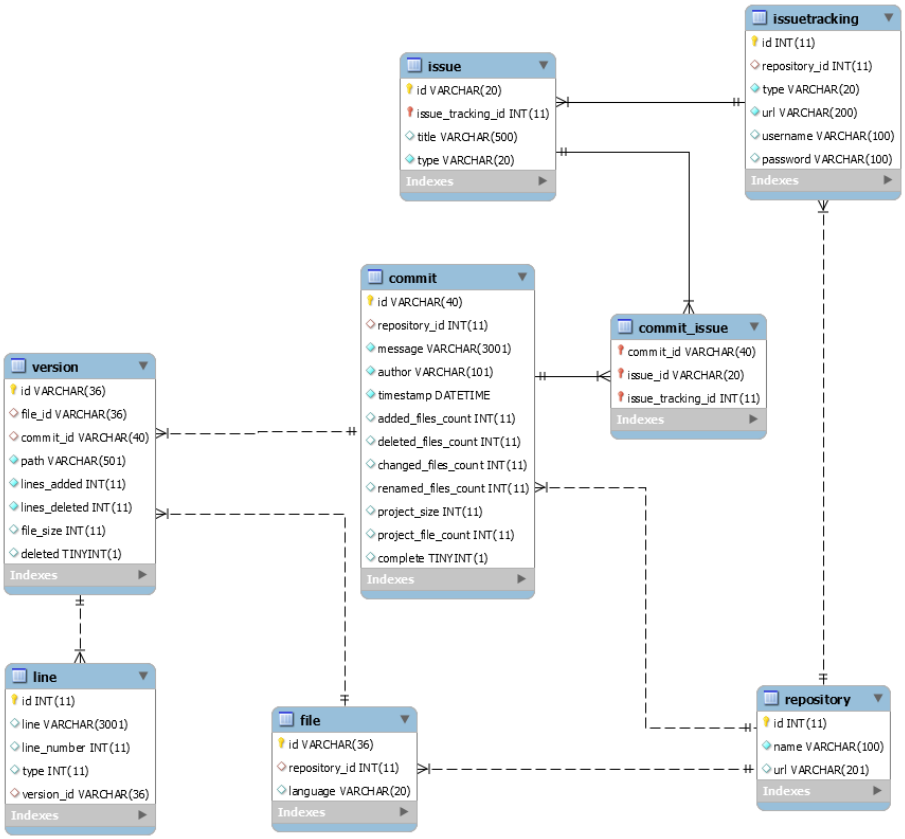

Abbildung 6: ERM GtSooG Datenbank

Commits sequentiell aus dem *Git*-Repository. Pro Commit werden die Code-Änderungen verarbeitet. Eine Änderung erzeugt eine neue Version einer Datei. Die Zuordnung von Version zu Datei erfolgt über den Pfad der Datei. Pro Version analysiert GtSooG die geänderten Code Zeilen. Der Issue Scanner ruft Issues vom Issue-Tracking-System via REST-Anfragen ab. Die Kadenz der Anfragen ist serverseitig von *GitHub* und *JIRA* begrenzt. Aufgrund dessen hätte ein Ansatz mit Multithreading keinen grossen Mehrwert gehabt. Die Abbildung 7 visualisiert den Ablauf von GtSooG. Den Datenbankzugriff übernimmt das Object Relational Mapping (ORM) Framework *SQLAlchemy*. Das DBMS im Hintergrund bleibt dadurch austauschbar. Anfangs wurde aufgrund der einfachen Handhabung mit *SQLite* gearbeitet. Schnell wurde aber klar, dass dieses triviale DBMS mit der anfallenden Datenmenge hoffnungslos überfordert ist. Das leistungsfähigere *MySQL* löste *SQLite* daher ab.

GtSooG soll die Daten stets in einem konsistenten Zustand hinterlassen. Da Commits voneinander abhängig sind, müssen sie entweder komplett (d. h. mit allen dazugehörigen Dateiversionen und geänderten Zeilen) oder gar nicht in der Datenbank gespeichert werden. Um dies auch bei einem Systemabsturz zu gewährleisten, entschieden wir uns, das Datenbankfeld *completed* in der Tabelle *commit* einzuführen. Erst nach der erfolgreichen Verarbeitung eines Commits

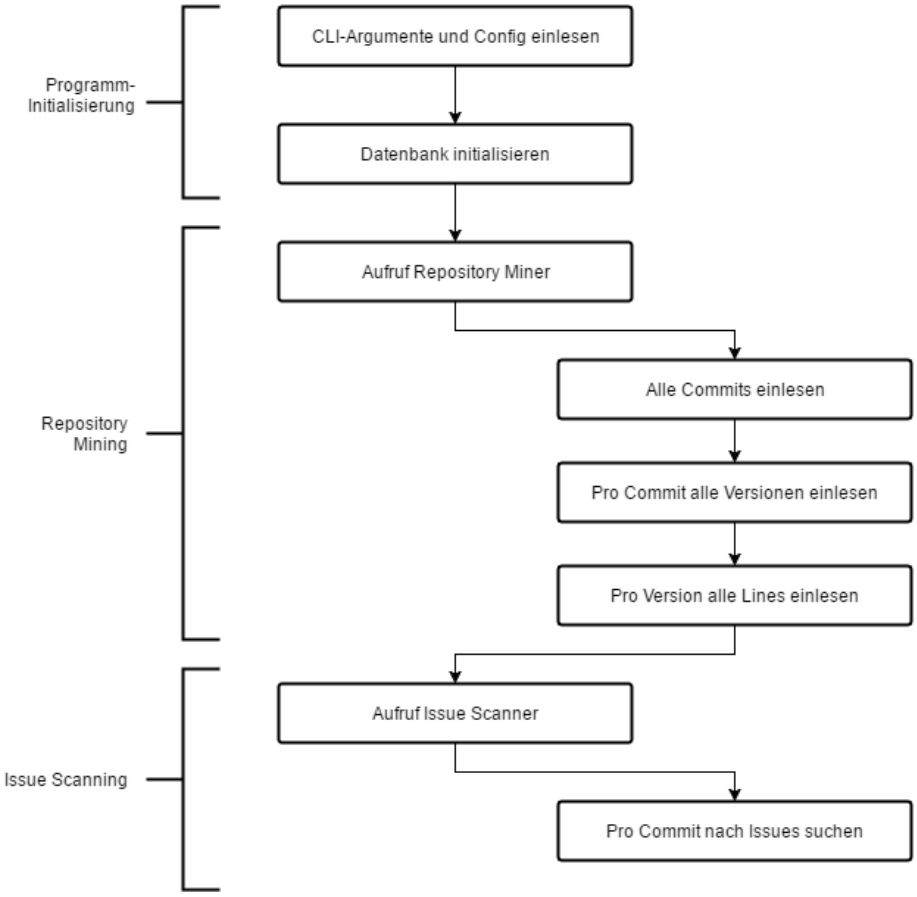

Abbildung 7: Der grobe Programmablauf von GtSooG

wird dieses Feld auf True gesetzt. Da beim Scannen von Issues kein Subprozess anfällt, waren dafür keine besonderen Massnahmen notwendig.

3.2.4 Erkenntnisse

Die Arbeit mit der *GitPython* Bibliothek stellte sich als schwierig heraus. Teile der von uns benötigten Funktionalität waren leider nicht vorhanden. Zum Beispiel existierte keine Funktion um die geänderten Zeilen pro Datei herauszufinden. *GitPython* bot lediglich den String-Output des *diff* Programms an. Dieser musste manuell geparst werden. Die Entwicklung und das Testing stellte sich als schwierig heraus. Die immense Datenmengen von unterschiedlichen Repositories erzeugten immer wieder Spezialfälle in den *diff*-Strings, was zu Fehlern in GtSooG führte. Da es für Python keine anderen, nennenswerten *Git*-Bibliotheken gab, hätten wir uns bei der Wahl der Programmiersprache im Nachhinein wohl besser für Java entschieden. Damit ist mit *Eclipse JGit* [76] eine etablierte *Git*-Bibliothek verfügbar.

3.3 Feature Extractor

Die Aufgabe des Feature Extractors besteht darin, die für das ML benötigten Features (siehe Kapitel 3.4), aus den gesammelten Daten zu generieren. Die Datenbasis besteht aus der von GtsooG erstellten Datenbank und dem entsprechenden *Git*-Repository im Dateisystem. Um die von der Gesamtarchitektur angestrebte Modularität zu gewährleisten, soll der Feature Extractor die generierten Features anschliessend in der DB speichern. Dafür entschieden wir uns trotz anfänglicher Bedenken bezüglich der grossen Datenmenge. Der Entscheidung liegen folgende Überlegungen zu Grunde:

- Die Verwendung der identischen Technologie wie GtSooG vereinfacht die Architektur.
- Es kann ein schneller Zugriff durch die strukturierte Speicherung der Daten in der Datenbank sichergestellt werden.
- Durch strikte Verwendung eines ORMs ist das DBMS austauschbar und somit nicht an Speicherlimits von Herstellern gebunden
- Das DBMS kann einfach von den restlichen Komponenten entkoppelt werden

3.3.1 Anforderungen

Der Feature Extractor soll folgende Anforderungen erfüllen:

Funktional:

- Zugriff auf die von GtSooG in der Datenbank gespeicherte Datenbasis
- Zugriff auf die zugehörigen *Git*-Repositories
- Generieren eines Java Abstract Syntax Tree (AST) aus Java Code
- Speichern der generierten Features in der SQL-Datenbank
- Korrekte Berechnung der im Kapitel 3.4 beschrieben Features

Nichtfunktional:

- Modulare und erweiterbare Architektur
- Kommandozeilenbasiert
- Verarbeitung mit Multithreading
- Einzelne Features sollen testbar sein
- Soll mit wenig Aufwand um zusätzliche Features erweitert werden können

3.3.2 Design

Der Feature Extractor wurde in drei Komponenten unterteilt:

- Die Feature Berechnung
- Die Datenbankanbindung per ORM
- Die Datenanbindung für die *Git*-Repository

Die extrahierten Features werden in der von GtSooG angelegten Datenbank gespeichert. Um dies zu ermöglichen, musste das Datenmodell um die Tabellen *feature_value* und *ngram_vector* erweitert werden. In der Abbildung 8 sind die neuen Tabellen der Datenbank farblich gekennzeichnet.

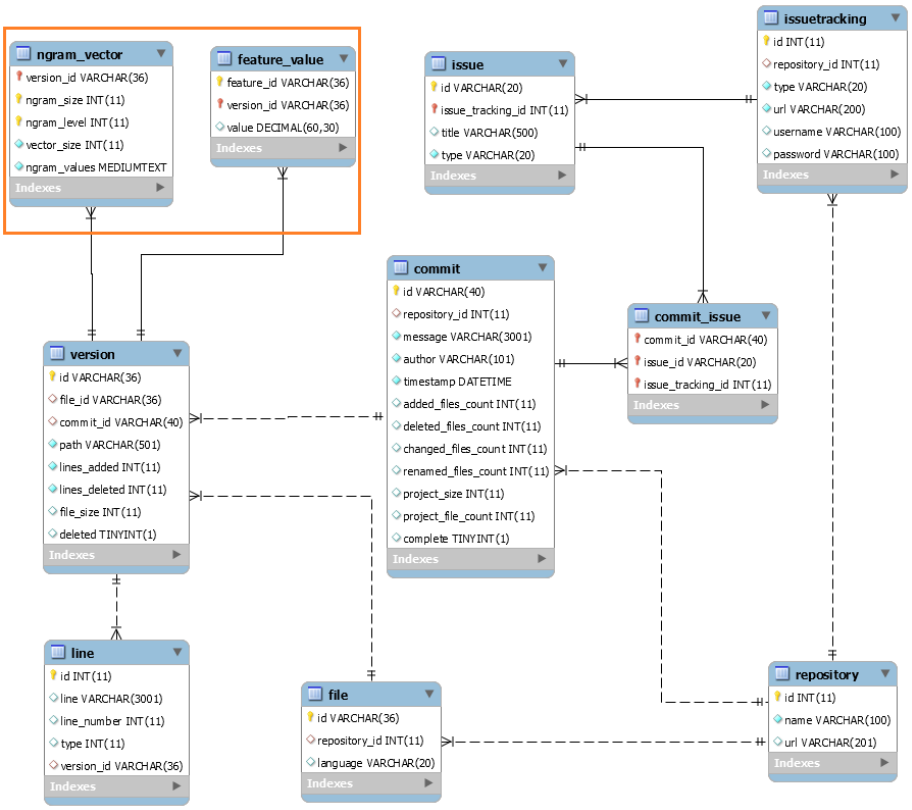

Abbildung 8: ERM der GtSooG Datenbank mit Feature Extractor Erweiterungen

Alle Features ausser N-Grams werden in der Tabelle *feature_value* gespeichert. Sie beinhaltet die Kennzeichnung des Features (*feature_id*), eine Referenz zur dazugehörigen Version (*version_id*) und den berechneten Wert (*value*). In der *feature_value*-Tabelle werden pro Version alle Features gespeichert. Mit einer grossen Anzahl Dateiversionen wächst die Grösse der Tabelle rapide an. Dieses Design bietet jedoch die Möglichkeit, einfach nach einzelnen Features filtern zu können.

Da wir bei den N-Grams verschiedene Längen und verschiedene Levels haben (siehe Kapitel 3.4.6) und auch danach filtern wollen, ist eine separate Tabelle (*ngram_vector*) notwendig. Der berechnete Feature-Wert der N-Grams (entspricht der Anzahl Vorkommnisse) wird als Komma-separierter String im Feld *ngram_values* hinterlegt. Auf diese Weise kommen auf jede Version nur wenige N-Gram-Datensätze, was die Tabelle klein hält. Trotzdem ist es noch möglich, auf bestimmte N-Gram-Kategorien zu selektieren. Die Selektion einzelner N-Grams ist keine Anforderung.

3.3.3 Implementierung

Der Feature Extractor ist in Java implementiert. Diese Entscheidung ist auf den Umstand zurückzuführen, dass für einige Features das Parsen und Analysieren von Java-Code nötig ist. *Eclipse JDT AST* [77] ist eine einfach zu handhabende Java-Bibliothek, welche diese Aufgabe erfüllen kann. Sie bietet die Funktionalität, Java-Source-Code als Abstract Syntax Tree (AST) darzustellen. Ausserdem befreit *Eclipse JDT AST* den Source Code automatisch von Namen und ermöglicht somit einen Vergleich mit anderen Source-Code-Dateien. Unter den Java-Syntaxparsern stellte sich *Eclipse JDT AST* gegenüber anderen Implementationen wie *javaparser* [78] als die mächtigste und weitverbreitetste Bibliothek heraus.

Ablauf und Erweiterbarkeit Der Feature Extractor liest zuerst die Versionen und Commits aus der Datenbank in den Arbeitsspeicher. Dann iteriert er durch die verschiedenen Versionen. Pro Version werden die konfigurierten Features berechnet. Um dabei eine bessere Leistung zu erzielen, werden ähnliche Features in Featuregruppen zusammengefasst. Zum Beispiel werden die Lines-of-Code-Features PLOC, SLOC, BLOC und CLOC in der Featuregruppe *LinesOfCodeFeatureGroup* zusammengefasst.

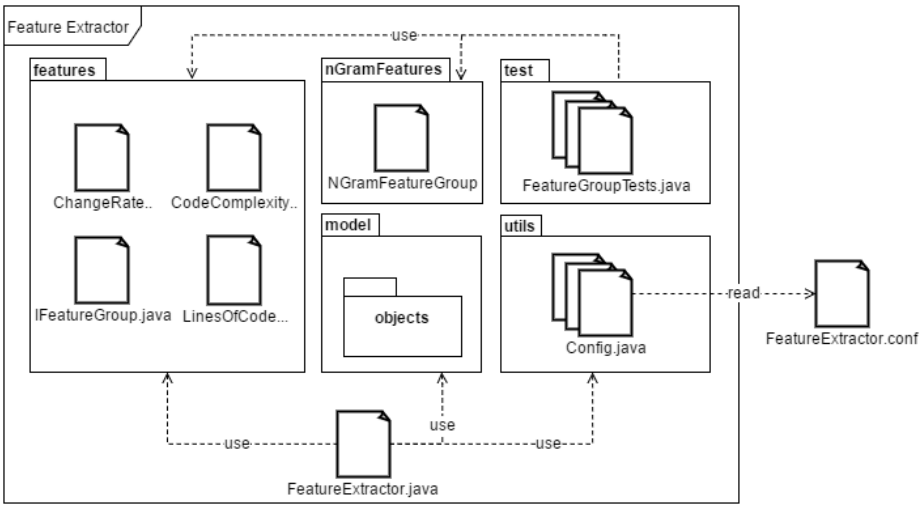

Abbildung 9: Die Package-Struktur des Feature Extractors (grob vereinfacht)

Das Hinzufügen von Features ist sehr einfach: Es muss eine neue Klasse

erstellt werden, die das *IFeatureGroup* Interface und deren Methode *extract* implementieren. In dieser Methode ist die Berechnung des oder der Features implementiert. Zusätzlich muss sich die Klasse im Package *features* befinden (siehe Abbildung 9). Der Feature Extractor iteriert automatisch mittels Reflection durch alle Klassen in diesem Package und ruft deren Methode *extract* auf. Als Rückgabewert erhält er eine HashMap. Diese Map hat als Key die Feature-Bezeichnung und als Value den dazugehörigen Wert. Die Implementation der *extract*-Methode erhält folgende Argumente:

- Den Java Source Code
- Den AST-Code
- Die aktuelle Version
- Eine Liste aller Commits

Der Java-Source-Code wird direkt aus einem *Git*-Repository ausgelesen. Dies übernimmt die *Git*-Komponente mit Hilfe der *JGit*-Bibliothek. Die Datenbankanbindung wurde mit dem weitverbreiteten ORM *HibernateORM* realisiert.

Datenzugriff Der Feature Extractor generiert eine grosse Anzahl von Daten. Diese Aufgabe ist weitaus anspruchsvoller als diejenige von GtSooG. Das Repository von *Elasticsearch* mit 10 Jahren erfasster Entwicklung umfasst 21'740 Commits und über 100'000 Dateiversionen. Daraus generiert der Feature Extractor 10 GB herkömmliche Features und 120 GB N-Grams. Bei der ersten Version des Feature Extractors stellte sich das Speichern der Daten in der DB als Flaschenhals heraus. Der DB-Server verzeichnete trotz Verwendung von SSDs eine Harddisk-Auslastung von 100% über den gesamten Durchlauf. Die effektive Schreibrate betrug aufgrund der Datenbank und der Implementierung lediglich ca. 0.5 MByte/s.

Folgende Optimierungsmassnahmen sollten den Datenbankzugriff beschleunigen:

- Es werden alle benötigten Daten (Versionen, Commits, Issues) beim Programmstart eingelesen um die Anzahl Lesezugriffe auf die Datenbank zu reduzieren.
- Bulk-Writing: Anstatt alle Features für jede Version einzeln zu schreiben, werden Schreiboperationen zusammenfassen.

Die Massnahmen zeigten Wirkung und die durchschnittliche Schreibrate stieg auf ca. 5 MByte/s. Weiteres Optimierungspotential wurde beim DBMS, in diesem Fall *MySQL* festgestellt. Normalerweise hat die Beständigkeit und Konsistenz der Daten in einer Datenbank höchste Priorität. Für die ML-Pipeline ist die Geschwindigkeit allerdings wichtiger. Deswegen haben wir uns entschieden, bei einer Schreiboperation nicht auf die definitive Schreibbestätigung (fsync) zu warten und die Daten asynchron zu schreiben. Daraus resultierten folgende Einstellungen für den SQL Server:

Listingsverzeichnis 3: MySQL Konfiguration

```
innodb_log_buffer_size=2G
innodb_flush_log_at_trx_commit=0
innodb_flush_method=nosync
skip-innodb_doublewrite
```

Es ist uns bewusst, dass bei einem Rechnerabsturz die Wahrscheinlichkeit eines Datenverlusts dadurch steigt. Diese Einstellungen führten aber zu einer immens höheren Schreibrate von ungefähr 25 MByte/s.

3.3.4 Erkenntnisse

Der Umgang mit dem Eclipse AST stellte sich teilweise als gewöhnungsbedürftig heraus. Features wie N-Grams und CodeComplexity sind besonders stark auf die Funktionalitäten von AST angewiesen. Da die verschiedenen Node-Typen (Klassen, Methoden, Variablen) teilweise für ähnliche Eigenschaften (zum Beispiel um alle Childnodes abzurufen) verschiedene Methoden verwenden, ist eine aufwendige Node Unterscheidung im Code notwendig. Dies führte zu viel und kompliziertem Code. Beim Vergleich mit anderen Projekten schien es dafür allerdings keine bessere Lösung zu geben.

3.4 Features

Features sind die messbaren Eigenschaften einer Beobachtung, auf deren Basis ein ML-Algorithmus trainiert wird, um dann anschliessend Resultate vorherzusagen. Ein Beispiel soll die Wichtigkeit von gut ausgewählten Features erläutern.

Eine Immobilienfirma versucht, Grundstückspreise anhand der Grundstücksfläche vorherzusagen. Dies ist eine typische ML-Aufgabe und kann in diesem Fall mit Hilfe von Linearer Regression gelöst werden. Das dabei eingesetzte Feature ist die Grundstücksfläche. Die Abbildung 10 zeigt einen Lerndatensatz mit der Grundstücksfläche als Feature auf der X-Achse und die dafür bekannten Grundstückspreise auf der Y-Achse. Der Zusammenhang ist trivial und die resultierende Regressionsgerade kann eine nahezu perfekte Vorhersage machen.

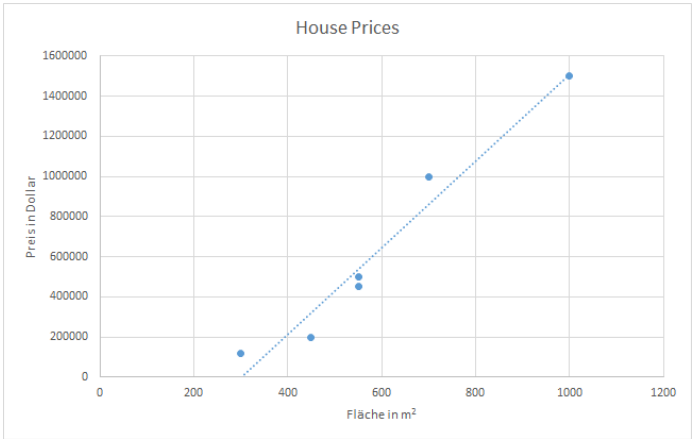

Abbildung 10: Vorhersage Grundstückspreise

Das Beispiel zeigt, dass die richtige Auswahl der Features grossen Einfluss auf das Resultat der ML-Applikation hat. Anzumerken ist, dass in realen Szenarien die Anzahl Features meist ein Vielfaches beträgt. Auch besteht in der Regel kein trivialer linearer Zusammenhang.

Eine der Herausforderungen dieser Arbeit war es, aus Java-Source-Code-Dateien sinnvolle Features zu bestimmen, welche für die Problemstellung, also der Fehlervorhersage, nützlich waren. Feature Design (auch Feature Engineering) ist oft ein schwieriger und zeitaufwändiger Prozess. Deshalb wurde sich hier zu einem grossen Teil auf bestehende Arbeiten gestützt.

Das folgende Kapitel soll eine Übersicht und detaillierte Beschreibung über die implementierten Features geben und schlägt auch einige Features vor, mit denen das Ergebnis in Zukunft noch weiter verbessert werden könnte.

3.4.1 Lines-of-Code-Features

Die Lines-of-Code-Features werden pro Dateiversion über den Source Code berechnet, unabhängig davon wie viele Klassen, Enums und Interfaces sich in einer Java-Source-Code-Datei befinden.

Abkürzung	**Name**	**Beschreibung**
PLOC	Physical lines of code	Zeilenanzahl einer Dateiversion im Total
SLOC	Total lines of code	Zeilen, die Java Code enthalten
CLOC	Comment lines of code	Anzahl Zeilen, die einen Kommentar enthalten
BLOC	Blank lines of code	Leere Zeilen
MINLINE	minimal line length	Minimale Zeilenlänge
MAXLINE	maximal line length	Maximale Zeilenlänge
MEDLINE	median line length	Median der Zeilenlängen

Tabelle 4: Lines-of-Code-Features

Da wir erwarteten, dass die Länge der einzelnen Codezeilen wahrscheinlich von Zeile zu Zeile sehr unterschiedlich sein wird, machte es keinen Sinn, diese einzeln zu erfassen. Stattdessen wurde der Mittelwert der Zeilenlänge verwendet. Als Mittelwert wurde bewusst der Median gewählt, da er robust gegenüber Ausreissern ist. Die Extreme wurden durch die maximale und minimale Zeilenlänge aufgezeichnet.

Die ausgewählten Features stützen sich auf mehrere Quellen. Sowohl Liljeson und Mohlin [27] über statische Code-Analyse und ML als auch Moriggl et al. [79] und Nagappan et al. [80] verwenden in ihren Arbeiten Lines-of-Code-Features.

Ein grosser Effekt wurde von den Lines-of-Code-Features nicht erwartet. Allerdings waren sie dank geringem Implementationsaufwand schnell einsatzfähig und boten eine gute Grundlage für die ersten Gehversuche mit ML. Hinzu kam, dass diese Features in diversen Arbeiten erwähnt wurden.

3.4.2 Objektorientierte Features

Objektorientierte Metriken wurden bereits öfters zur Fehlervorhersage verwendet. So zum Beispiel von Gyimothy et al. [81] und Siket et al. [82] auf verschiedenen Open-Source-Projekten. Es sind zudem die einzigen Features, welche die

Eigenschaften von objektorientiertem Programmieren erfassen sollen. Deshalb war anzunehmen, dass diese gerade bei Java einen signifikanten Einfluss auf das Resultat haben könnten.

Wie die Lines-of-Code-Features berechnet der Feature Extractor die objektorientierten Features pro Java-Source-Code-Datei. Befinden sich mehrere Klassen, Enums oder Interfaces in einer Java-Source-Code-Datei werden die Features pro Klasse erfasst. Falls pro Datei mehrere Klassen vorhanden sind, werden die Feature-Werte pro Dateiversion zusammengezählt.

Abkürzung	**Name**	**Beschreibung**
WMC	Weighted methods per class	Anzahl Methoden pro Klasse (inkl. Konstruktoren)
CBO	Coupling between objects	Die Anzahl der nicht zur Klasse gehörenden Methodenaufrufe
RFC	Response for class	WMC und CBO zusammengezählt.
LCOM	Lack of cohesion in methods	Anzahl Klassenmethoden die nicht auf eine Klassenvariable zugreifen
NPM	Number of public methods	Anzahl Klassenmethoden mit dem Modifier public
NPV	Number of public variables	Anzahl Klassenvariablen mit dem Modifier public

Tabelle 5: Objektorientierte Features

Die objektorientierten Features wurden weitgehend aus der Arbeit von Liljeson und Mohlin [27] übernommen. Die objektorientierten Features Depth of Inheritance Tree (DIT), Number of Children (NOC) und Afferent coulings (AC) benötigen zur Berechnung den Source Code des gesamten Projekts. Da der Feature Extractor Dateiversionen isoliert behandelt, ist die Betrachtung des ganzen Source Codes sehr aufwendig. Aufgrund dessen und weil wir den Fokus auf Textanalyse-Features gelegt haben, wurden diese Features nicht implementiert. Eine spätere Erweiterung des Feature Extractors ist möglich.

3.4.3 Code-Complexity-Features

Die Literaturrecherche hat gezeigt, dass bei statischer Code-Analyse Komplexitätsmetriken schon seit langem miteinbezogen werden. Es erschien uns deshalb als sinnvoll, eine Auswahl der bekanntesten Code-Komplexitätsmetriken als Features zu implementieren. Wir erhofften uns im besten Fall eine Kausalität zwischen Software-Komplexität und Softwarefehlern. Die Entscheidung fiel auf Metriken nach Halstead [20] und McCabe [19]. Begründen lässt sich die Auswahl aufgrund der Bekanntheit und Etabliertheit der beiden Metriken.

Das Herzstück der Halstead-Metrik sind folgende aus dem Source Code extrahierten Werte:

Abkürzung	Beschreibung
n1	Anzahl eindeutiger Operatoren
n2	Anzahl eindeutiger Operanden
N1	Vorkommnisse der Operatoren im Code
N2	Vorkommnisse der Operanden im Code

Tabelle 6: Healstead-Operanden und -Operatoren

Daraus werden anschliessend diverse Kennwerte wie zum Beispiel die Anzahl zu erwartende Bugs (nach Halstead) oder der Programmieraufwand berechnet. Da die Halstead-Metrik lange vor Java definiert und entwickelt worden ist, war die grösste Herausforderung das Einordnen der Java-Objekte in die beiden Klassen Operatoren und Operanden. Die Implementierung dieser Arbeit lehnt sich an die Arbeit von Dinari et al. [83] an.

Die McCabe Metrik ist weiter verbreitet. Es existieren in Java geschriebene Eclipse-Plugins zur automatischen Berechnung. Unsere Implementation lehnt sich stark an das Eclipse-Plugin *metrics2* [84] an.

Aus den beiden Metriken entstehen folgende Features:

Abkürzung	Name	Beschreibung
n1	n1	Anzahl eindeutiger Operatoren
n2	n2	Anzahl eindeutiger Operanden
N1	N1	Vorkommnisse der Operatoren im Code
N2	N2	Vorkommnisse der Operanden im Code
n	Vocabulary length	n1 + n2
N	Program length	N1 + N2
$\hat{N}$	Calculated program length	$n1 * \log_2 n1 + n2 * \log_2 n2$
V	Volume	$N * \log_2 n$
E	Effort	$\frac{n1}{2} * \frac{N2}{n2}$
T	Time required to program	$\frac{effort}{18}$
B	number of bugs	$\frac{effort^{\frac{2}{3}}}{3000}$
mccabeTotal	McCabe-Total	McCabe Metrik für die ganze Version
mccabeMethod	McCabe per Method	McCabe Metrik pro Methode $\frac{mccabeTotal}{MethodCount}$
mccabeClass	McCabe per Class	McCabe Metrik pro Klasse $\frac{mccabeTotal}{ClassCount}$
mccabeClassMethod	McCabe per Method and Class	McCabe Metrik pro Klasse und Methoden $\frac{mccabeTotal}{ClassCount*MethodCount}$

Tabelle 7: Code-Complexity-Features

3.4.4 Anzahl-und-Typen-Features

Die Auswirkung von Anzahl-und-Typen-Features ist schwer vorherzusagen. Vielleicht können triviale Java Klassen, welche bloss ein Datenmodell abbilden, von fehleranfälligen Klassen mit vielen komplexen Methoden unterschieden werden. Da diese Features einfach zu implementieren sind, wurden sie integriert.

Die Anzahl-und-Typen-Features konzentrieren sich auf die Vorkommnisse von High-Level-Elementen in einer Java-Version. Zu den High-Level-Elementen zählen:

- Imports
- Klassen
- Interfaces
- Enums
- Methoden
- Klassenvariablen
- Klassenkonstanten
- lokale Variablen

Konstanten werden durch in Java über final static Modifiers identifiziert. Für Klassen, Interfaces, Enums, Methoden, Klassenvariablen und Klassenkonstanten wird die Anzahl Vorkommnisse ausserdem pro Modifier (public, private und protected) gezählt.

3.4.5 Temporale Features

Die temporalen Features machen sich dem Namen entsprechend zeitabhängige Informationen der Versionsverwaltung zunutze. Die einzelnen Features sollen vom Zeitpunkt der aktuellen Dateiversion über eine Zeitspanne berechnet werden.

Die von GtSooG gesammelten Metadaten über das *Elasticsearch* Projekt werden im Kapitel 4.2.2 analysiert und mit Hilfe von Statistiken grafisch dargestellt. Die Auswertung offenbarte interessante Details über den Zusammenhang von Enhancements und Bugs. Werden viele Erweiterungen erzeugt, folgen mit einer zeitlichen Verzögerung eine erhöhte Anzahl von Bugs. Diese Informationen erschienen uns für die Vorhersage von Bugs als sehr wichtig und wurden deswegen in Form temporaler Features aufgenommen. Da viele Änderungen am Source Code auch die Bug-Wahrscheinlichkeit erhöhen, sind als weitere Features die Anzahl geänderter Dateien, sowie die Anzahl hinzugefügter und gelöschter Zeilen als Features erfasst.

Wir entschieden uns diese Werte über mehrere Zeitspannen zu speichern. Die Abstände der Zeitspannen wurden bewusst nicht linear gewählt, sodass mit nur sieben Zeitspannen einen möglichst grosser Zeitraum von 2 Jahren abgedeckt werden kann.

- 1 Tag

- 1 Woche
- 1 Monat
- 3 Monate
- 6 Monate
- 1 Jahr
- 2 Jahre

Die folgenden Features werden pro Zeitspanne berechnet:

Abkürzung	**Name**	**Beschreibung**
NOAL	Number of added lines	Anzahl hinzugefügter Zeilen über einen Zeitraum
NODL	Number of deleted lines	Anzahl gelöschter Zeilen über einen Zeitraum
NOA	Number of authors	Anzahl verschiedener Autoren über einen Zeitraum
NOB	Number of bugs	Anzahl Bugfixes über einen Zeitraum
NOE	Number of enhancements	Anzahl Enhancements über einen Zeitraum

Tabelle 8: Temporale Features

Aus der Kombination der Zeitspannen und den Features ergeben sich insgesamt 30 Features. Das Beispiel "number of added lines per month" wird in der Datenbank als NOAL30D gespeichert. Der Feature Extractor sucht vom Zeitpunkt der Dateiversion 30 Tage in die Vergangenheit nach Commits, welche eine ältere Version derselben Datei enthalten. Für jeden Commit werden die Anzahl hinzugefügter Zeilen aufsummiert.

Die Anzahl geänderter Dateien pro Commit werden für alle oben aufgeführten Zeitspannen berechnet. In einer der oben aufgelisteten Zeitspannen können mehrere Commits sein. Da die Anzahl der Commits in der Zeitspanne von Version zu Version unterschiedlich ist, erfassen wir von den folgenden Veränderungstypen den Median, den Mittelwert, das Maximum und das Minimum.

Abkürzung	**Name**	**Beschreibung**
NOAF	Number of added files	Anzahl hinzugefügter Dateien über einen Zeitraum
NODF	Number of deleted files	Anzahl gelöschter Dateien über einen Zeitraum
NOCF	Number of changed files	Anzahl geänderter Dateien über einen Zeitraum
NORF	Number of renamed files	Anzahl umbenannter Dateien über einen Zeitraum

Tabelle 9: Temporale Features - geänderte Dateien

Ein weiteres Feature, DBLC (Days between last commit), berechnet die Anzahl Tage zwischen dem Commit mit der aktuellen Version und dem vorherigem Commit, welcher die Dateiversion enthält. Damit soll die Zeitspanne bis zur letzten Änderung der Datei erfasst werden.

3.4.6 Textanalyse-Features

In der Text- und Sprachanalyse werden verschiedene Analyseverfahren zur Detektion von Bedeutungsstrukturen eingesetzt. In Programmiersprachen ist die Bedeutung von Wörtern normalerweise eindeutig. Durch Kombination der Wörter entsteht die Programmlogik, welche man im entferntesten Sinne mit der Semantik einer natürlichen Sprache vergleichen kann. Daraus entstand die Idee, Textanalyse-Features auf eine Programmiersprache anzuwenden. Eine Programmiersprache weist grundsätzlich eine einfachere Grammatik auf. Zum Beispiel entfallen Konjugationen von Wörtern. Dadurch gibt es weniger Interpretationsspielraum. Auch erhoffen wir uns daraus bei den N-Gram Features eine geringe Anzahl von Daten zu erhalten.

Einen guten Überblick über verschiedene Textanalyse-Features liefert Mohammad et al. mit ihrer Arbeit über Twitter-Sentiment-Analyse [54]. Aus dieser Arbeit erschienen uns N-Grams und die Länge von Namen als vielversprechend. Die N-Grams könnten verschiedene Anordnungen von Java-Code-Wörtern, welche einem Bug zugeordnet sind, identifizieren und diese in einer zukünftigen fehlerbehafteten Versionen wiedererkennen. Die Länge von Namen soll die Einhaltung von Code Style (Naming-Konventionen) prüfen. Vielleicht ist ein Nichteinhalten ein Indiz auf einen Softwarefehler.

Abkürzung	**Name**	**Beschreibung**
MINVAR	Minimal variable name length	Der kürzeste Variablenname der Klasse und deren Methoden
MAXVAR	Maximal variable name length	Der längste Variablenname der Klasse und deren Methoden
MEDVAR	Median of variable name length	Der Median über alle Variablennamen
MINMET	Minimal method name length	Der kürzeste Methodenname der Klasse
MAXMET	Maximal method name length	Der längste Methodenname der Klasse
MEDMET	Median of method name length	Der Median über alle Methodennamen
MINCLS	Minimal class name length	Der kürzeste Klassenname
MAXCLS	Maximal class name length	Der längste Klassenname
MEDCLS	Median of class name length	Der Median der Klassennamen

Tabelle 10: Textanalyse-Features

Länge von Namen Die Länge der Namen wird nicht über Java-Schlüsselwörter, sondern über die Länge von Variablen-, Klassen- und weiteren Namen be-

rechnet. Es werden die in der Tabelle 10 aufgeführten Längenangaben gesammelt.

Der Wert für den kürzesten und längsten Namen, sowie der Median einer Klasse wird in den meisten Fällen identisch sein, da in der Regel pro Java-Datei nur eine Klasse vorhanden ist.

N-Grams Den Ansatz, N-Grams auf Source Code anzuwenden wurde bis dato nur selten verfolgt. Die Arbeit von Chollak et al. [85] versucht mit N-Grams den Kontrollfluss und die Reihenfolge von Methodenaufrufen zu analysieren.

In dieser Arbeit werden die N-Grams nicht direkt auf den Source Code angewendet sondern auf dem Abstract Syntax Tree (AST). Da im Source Code viele individuelle Namen (Klassenname, Methodenname) vorkommen, ist eine Analyse direkt auf dem Code nicht sinnvoll. Mit Hilfe des AST können die Namen durch die Knoten-Bezeichnung ersetzt werden. Am Beispiel dieses Java Codes:

Listingsverzeichnis 4: Java Source Code

```
public class MyClass {
        private String name;
        private int id;
        ...
```

ergeben sich folgenden AST-Knoten:

Listingsverzeichnis 5: AST Elemente mit Levels

```
(55, TypeDeclaration, level 1-4)
 (83, Modifier, level 1)
 (42, SimpleName, level 1)
 (43, SimpleType, level 1)
  (42, SimpleName, level 1)
 (23, FieldDeclaration, level 1-2)
  (83, Modifier, level 1)
   (43, SimpleType, level 1)
    (42, SimpleName, level 1)
  (59, VariableDeclarationFragment, level 1)
   (42, SimpleName, level 1)
 (23, FieldDeclaration, level 1-2)
  (83, Modifier, level 1)
  (39, PrimitiveType, level 1)
  (59, VariableDeclarationFragment, level 1)
   (42, SimpleName, level 1)
```

Jedes Code Fragment kann einem AST-Type zugeordnet werden. Der AST-Type hat eine Bezeichnung und eine Nummer. Um Speicherplatz zu sparen verwenden wir nur die Nummer. Ein 3-Gram hat dann beispielsweise folgende Form: *83_43_42*.

Gut zu erkennen ist, dass aus kurzen Code Fragmenten ein grosser AST resultiert. Um Kontrollflüsse im Programm festzustellen, wären sehr grosse N-Grams nötig. Je grösser die N-Grams, desto mehr Speicherplatz und Rechenleistung werden benötigt. Damit wir den Kontrollfluss trotzdem erfassen können, haben wir uns entschieden, verschiedene Abstraktionslevels des ASTs zu erfassen.

Level Nummer	Beschreibung
1	Gesamter AST ohne Ausnahmen
2	Auf Statement Level. Somit fallen alle Subtypen wie zum Beispiel SimpleName, VariableDeclarationFrament weg.
3	Alle Klassen, Interfaces, Methoden, Kontrollstrukturen und Schleifen
4	Alle Klassen, Interfaces und Methoden

Tabelle 11: AST Abstraktions-Level

Im Listing 5 sind die Levels zu welchen das entsprechende Statement gehört gekennzeichnet. Die maximale Länge der N-Grams ist in der Konfiguration des Feature Extractors hinterlegt und kann frei gewählt werden. Bei einem grossen Projekt wie *Elasticsearch* mit über 100'000 Dateiversionen, benötigt das Berechnen der Features ca. 28 GB RAM. Deswegen setzen wie die maximale N-Gram Grösse standardmässig auf 5.

Die N-Grams werden als String in der Datenbank in der Tabelle *ngram_vector* (siehe Kapitel 3.3.2) gespeichert. Die ML-Applikation benötigt pro Version die Anzahl Vorkommnisse aller N-Grams. Um dies zu erreichen werden in einem ersten Schritt alle N-Grams über alle Versionen gesammelt. Im zweiten Durchlauf wird pro Version die Anzahl Vorkommnisse der N-Grams als Komma-separierter String gespeichert. Ein Datensatz aus *ngram_vector* könnte wie in der Tabelle 12 beschrieben aussehen.

Feld	Wert
version_id	'0001beff499e4302af1121f0fe6279e8'
ngram_size	1
ngram_level	1
vector_size	68
ngram_values	'2,7,11,3,0,0,13,1,0,1,2,2,0,2,1,0,9,4,3,0,3,1,1,1,1,2,2,10,41,1, 113,2,18,2,3,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0'

Tabelle 12: Beispieldatensatz aus Tabelle *ngram_vector*

Die *version_id* stellt die Relation zur Version her. Die *ngram_size* gibt an, um welchen Grad N es sich bei den N-Grams handelt, in diesem Fall 1-Grams. Der *ngram_level* beschreibt den Abstraktions-Level des ASTs. Die *vector_size* gibt die Anzahl N-Grams an. Die *ngram_values* geben die Vorkommnisse der N-Grams an. Der eigentliche N-Gram-Typ ist nicht in der Datenbank gespeichert. Damit die N-Grams beim späteren ML-Prozess verglichen werden können, muss die Reihenfolge im Feld *ngram_values* stets dieselbe sein. Pro Version, N-Gram-Grösse und N-Gram-Level wird ein Eintrag in der Tabelle erstellt.

Eine weitere Überlegung wäre das Anwenden der N-Grams auf den Java Byte Code. Da die Effektivität dieses Verfahrens schwer abzuschätzen war, verzichteten wir allerdings vorerst darauf.

3.4.7 Ideen für weitere Features

Neben den implementierten Features sind noch Ideen zu weiteren Features entstanden, welche aufgrund der Projektdimension nicht implementiert werden konnten. Im Folgenden sollen diese kurz beschrieben werden, da sie vielleicht zukünftigen Arbeiten dienlich sein könnten.

Textanalyse auf Kommentare Kommentare im Code könnten Rückschlüsse auf die Qualität des dazugehörigen Source Codes ermöglichen. Zum Beispiel weist 'quick fix', 'fix me' oder 'TODO' auf unvollständigen evtl. Bug-anfälligen Code hin.

Clean Code Code Konventionen für Programmiersprachen haben sich über die Zeit anhand von Praxiserfahrung entwickelt. Die Einhaltung von Code Styles und Clean Code könnte ein Indiz auf qualitativ guten, fehlerfreie Software sein.

Kommentarlänge im Verhältnis zur Methodenlänge Das Verhältnis der Länge von JavaDoc-Kommentaren zur Anzahl Statements und Kontrollstrukturen in einer Java-Methode könnte interessant sein. Zum Beispiel könnte ein sehr langer Kommentar für eine Methode mit einer sehr kleinen Anzahl Statements auf ein komplexes, fehleranfälliges Konstrukt hinweisen.

Fehleranfällige Imports Schröter et al. [28] messen in seiner Arbeit die Fehleranfälligkeit von häufig importierten Java-Paketen und überprüfen ob die konsumierenden Java-Klassen dann auch von Fehlern betroffen sind. Die von Schröter aufgestellte Hypothese erwies sich als wahr. Die Datengrundlage, welche GtSooG schafft, beinhaltet nur den Projektcode. Es ist also nicht möglich die Imports nach Fehlern abzusuchen. Es besteht jedoch die Möglichkeit die Imports der Projektklassen zu erfassen. Der Gebrauch eines Imports in einer Java-Klasse mit vielen Fehlern weist vielleicht auf eine hohe Fehleranfälligkeit beim Gebrauch des Imports hin und somit auch auf eine erhöhte Fehleranfälligkeit der restlichen konsumierenden Klassen.

Test-Metriken Ein interessanter Ansatz ist das Berechnen von Metriken über den Test Code. In der Regel werden dieselben Features wie zum Beispiel Lines-of-Code auf den Source Code angewendet. Nagappan [86] entwickelte in seiner Doktorarbeit eine Test-Metrik-Suite *STRAW*, welche die Metriken des Test Codes ins Verhältnis mit dem Source Code stellt. Eine Implementation von *STRAW* für Java hat Nagappan [87] in einer weiteren Arbeit nachgereicht.

3.5 ML-Pipeline

Die ML-Pipeline ist die dritte und letzte Softwarekomponente dieser Arbeit. Sie vereint die beim Repository Mining gesammelten und vom Feature Extractor verarbeiteten Daten, um dann auf deren Basis mittels Machine Learning ein Modell zu trainieren. Mit diesem sollen kommende Bugfixes vorhergesagt werden können.

In diesem Kapitel werden die Anforderungen an dieses Tool, eine Übersicht über das Softwaredesign und relevante Entscheidungen sowie Schlüsselerkenntnisse aus der Entwicklung beleuchtet.

3.5.1 Anforderungen

Folgende Anforderungen sollen mit der ML-Pipeline erfüllt werden:

Funktional:

- Das Tool benötigt lesenden Zugriff auf die Datenbank
- Die Steuerung über eine Konfigurationsdatei
- Ein sinnvolles Mass an Logging soll vorhanden sein, damit Programmverlauf und -zustand während und nach der Ausführung nachvollziehbar sind
- Es sollen verschiedene Regressionsmodelle unterstützt werden, mindestens lineare Regression, Ridge Regression und SVR
- Cross Validation soll für relevante Parameter unterstützt werden
- Alle relevanten Parameter der Modelle müssen einfach konfigurierbar sein
- Die zu verwendenden Trainings- und Testsets müssen einfach auswählbar sein
- Die in den Datensets zu verwendenden Features müssen konfigurierbar sein
- Neben normalen Features müssen auch N-Grams unterstützt werden
- Es müssen sinnvolle Reports und Auswertungen ausgegeben werden können

Nichtfunktional:

- Es soll eine modulare, erweiterbare Architektur implementiert werden
- Die Software soll auf Kommandozeile ausgeführt werden können und ohne GUI funktionieren
- Wo sinnvoll, soll Multithreading zur Erhöhung der Performance eingesetzt werden
- Der Code soll übersichtlich dokumentiert sein, so dass er auch von Dritten benutzt und erweitert werden kann

3.5.2 Design

Wahl eines ML-Frameworks Da die ML-Pipeline primär ein Wrapper um ML-Funktionen ist, war die zentralste Designentscheidung das Machine Learning Framework zu bestimmen. Diese Entscheidung war nicht nur massgeblich für die Architektur, sondern auch mitbestimmend für die Wahl der Plattform.

Nachdem diverse Frameworks in Erwägung gezogen wurden, fiel die Entscheidung auf die bekannte und weitverbreitete Python ML-Bibliothek *scikit-learn* [88], [89]. Sie bietet nicht nur eine breite Auswahl an ML-Algorithmen

(von Klassifikation über Regression bis Clustering), sondern auch diverse andere Tools wie Dimensionality Reduction, Model Selection und diverses Preprocessing. Dabei baut *scikit-learn* auf die bewährten Python-Bibliotheken *Numpy*, *SciPy* und *matplotlib*. Alle Funktionen sind sinnvoll gekapselt und können einfach und schnell verwendet werden. Trotzdem bietet *scikit-learn* auch eine Menge Konfigurationsparameter und lässt im Detailgrad der Konfiguration so gut wie keine Wünsche offen. Ebenfalls hervorzuheben ist die sehr gute und vollständige Dokumentation aller Komponenten und Funktionen. *scikit-learn* ist Open Source, steht unter der BSD Lizenz und kann damit auch kommerziell verwendet werden. Es ist seit 2007 in aktiver Entwicklung.

Der Umstand, dass es sich bei *scikit-learn* um eine Python-Bibliothek handelt kam uns entgegen, da der Repository Miner bereits in Python implementiert wurde. So können Erkenntnisse, wie z. B. die Benutzung des *SQLAlchemy* ORMs, direkt in die ML-Pipeline einfliessen.

Alternativen, welche in der Entscheidungsphase evaluiert wurden, waren vor allem folgende Softwareprodukte:

- **Apache Spark MLlib** [90]: Basierend auf dem verbreiteten Open Source Cluster Computing Framework *Apache Spark* bietet die MLlib eine umfangreiche Bibliothek von ML-Algorithmen. Durch die distributed memory-based Spark-Architektur und gut darauf angepasste Algorithmen ist Spark MLlib äusserst performant und skalierbar. Es wird sowohl Python als auch Java zur Entwicklung unterstützt. Für ein grösseres Projekt wäre Apache Spark sicher eine mächtige Lösung. Für diese Arbeit war aber weder die nötige Infrastruktur noch genug Wissen über verteilte Storage Solutions (wie z. B. Hadoop) vorhanden.

- **LibSVM** [91] und **LibLinear** [92]: Low-Level C Bibliotheken für Support Vector Machines und lineare Modelle. Ein direkter Einsatz wäre aufgrund fehlenden C-Know-Hows schwierig gewesen. Ausserdem hätte es dem Ziel einer übersichtlichen und einfach erweiterbaren Lösung widersprochen. Beide Bibliotheken können aber über Bindings in *scikit-learn* verwendet werden [93].

- **Cloud Lösungen**: Sowohl Microsoft mit *Azure Machine Learning* [94] als auch Amazon mit *Amazon Machine Learning* [95] bieten ML-Funktionalität in der Cloud an, beide mit einem pay-as-you-go Monetarisierungsmodell. Neben den Kosten kamen diese Lösungen insbesondere wegen der Plattformabhängigkeit und anderen Limitationen nicht in Frage.

- **Pylearn2** [96]: Ein Python-Toolset für ML, welches auf auf *Theano* [97] basiert, einer Python-Bibliothek für das Definieren, Optimieren und Evaluieren von mathematischen Ausdrücken. Pylearn2 hat eine vielversprechende Vision, befindet sich aber noch in einer zu frühen Entwicklungsphase.

- **PlebML** [98]: Ein Jahr vor dieser Arbeit wurde am Institut für angewandte Informationstechnologie der ZHAW im Rahmen einer Bachelorarbeit PlebML entwickelt, ein modulares ML-Framework in Java. Von Vorteil wäre der einfache Kontakt zu den Entwicklern gewesen. Da PlebML aber vorläufig nur Textklassifikation unterstützt wäre viel Erweiterungsarbeit nötig gewesen, um das Framework für unsere Zwecke benutzen zu können.

- **mlpack** [99]: Eine C++ Bibliothek mit einer Vielzahl an ML-Funktionen. Mlpack wurde mangels C++ Know-How nicht verwendet, Bindings zu Java oder Python existieren nicht.

Weiterhin wurde in der ML-Pipeline *NumPy* [100] für einen Grossteil der Berechnungen und das Handling von Vektoren und Matrizen benutzt. Für Sparse-Matrizen wurde das *scipy.sparse* Package [101] genutzt. Wie beim Repository Miner wurde als ORM *SQLAlchemy* [74] eingesetzt und *pymysql* [102] als MySQL-Connector. Um Grafiken zu erzeugen wurde auf *matplotlib* [103] gesetzt, eine mächtigen Programmbibliothek für mathematische Darstellungen, die eng mit *NumPy* zusammen arbeitet und sich sehr ähnlich wie *MATLAB* bedienen lässt. Um textuelle Auswertungen übersichtlich darzustellen, wurde ausserdem *terminaltables* [104] verwendet, eine kompakte Bibliothek die es erlaubt, Daten einfach in ASCII-Tabellen darzustellen.

Architektur Wie es der Name "ML-Pipeline" bereits antönt, ist der Programmablauf sehr geradlinig und lässt sich einfach in aufeinanderfolgende Phasen resp. Schritte einteilen:

1. **Programminitialisierung**: Als erstes soll die Konfiguration eingelesen werden. Sie enthält alle Einstellungen, die während dem Rest des Programmablaufs relevant sind. Optional wird zuerst noch das Kommandozeilen-Argument '-c' eingelesen, mit welchem der Pfad zum Config-File spezifiziert werden kann. Anschliessend wird die Datenbankanbindung mit den Einstellungen aus dem Config-File initialisiert.

2. **Lernphase**: Zu Beginn der Lernphase werden zuerst die Trainings- und Testsets aus der Datenbank oder aus dem Dateisystem in den Arbeitsspeicher geladen. Dann wird das ML-Modell entsprechend der Konfiguration erstellt. Dieses Modell lernt dann auf dem Trainingsset. Die Lernphase ist die komplizierteste und aufwändigste der drei Phasen.

3. **Auswertung**: In der letzten Phase werden verschiedene Predictions erstellt. Einerseits wird das gelernte Modell auf das Testset angewendet, andererseits werden zum Vergleich verschiedene, naive Baseline-Predictions erstellt. Basierend auf diesen Predictions und dem Modell werden zum Schluss diverse Auswertungen generiert und angezeigt.

Dieser Ablauf wird in Abbildung 11 visualisiert.

Die Projektstruktur ist ähnlich simpel aufgebaut, wie die Abbildung 12 zeigt. Es wurde insbesondere darauf geachtet, die Abhängigkeiten klein zu halten. Der Einstiegspunkt ist das Python-File *ml_pipeline.py*, welches auch für den Ablauf der Pipeline zuständig ist. Im Package *utils* ist vor allem das Modul *Config* relevant, welches das externe Konfigurationsfile liest und auch den Zustand der Konfiguration speichert. Dieser Zustand wird ausschliesslich von *ml_pipeline.py* gelesen und dann über Parameter an die Logik in den anderen Packages weitergegeben. Dadurch wird eine zu grosse Abhängigkeit zu *Config* verhindert. Das Package *model* enthält die ORM-Objekte und die DB-Anbindungslogik. Letztere ist mit der Implementation des Repository Miners grösstenteils identisch. Bei den ORM-Objekten wurden für die ML-Pipeline nicht relevante Attribute aus Effizienzgründen entfernt. Der grösste Teil der Logik befindet sich im Package

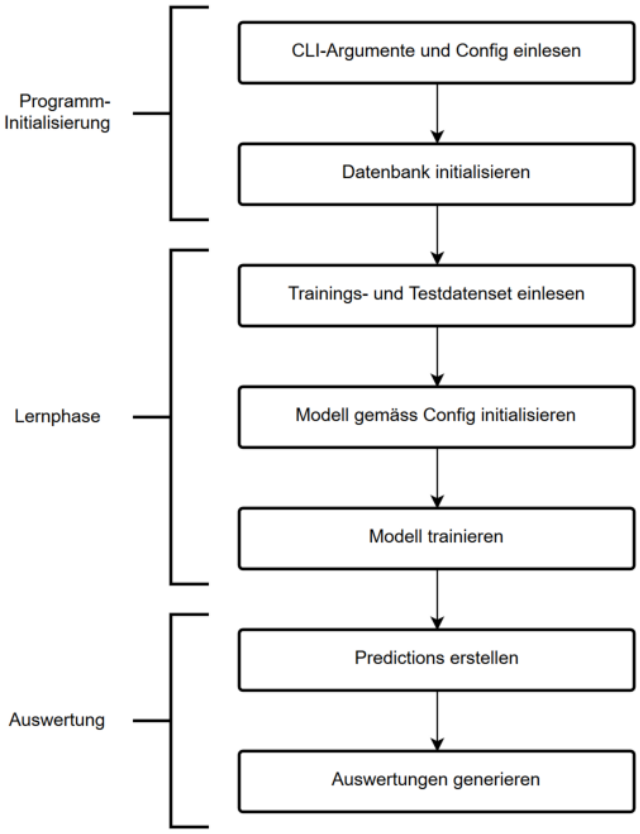

Abbildung 11: Der grobe Programmablauf der ML-Pipeline

ml. Dataset ist für das Laden und Verarbeiten von Datensets zuständig, *Model* für das Initialisieren und Trainiern der Regressionsmodelle, *Predict* erstellt Model- und Baseline-Predictions und *Reporting* enthält alle Auswertungsfunktionen.

3.5.3 Implementierung

An dieser Stelle wird auf einige hervorzuhebende Probleme und Lösungen in der Implementierung der ML-Pipeline eingegangen.

Datasets Der Umgang mit den grossen Datenmengen der Trainings- und Testsets war eine der grösseren Herausforderungen bei der Implementation der ML-Pipeline und erforderte diverse Optimierungen. Datensets sind Objekte, die aus 2 Komponenten bestehen, der Data-Matrix (Attribut *data*) und dem Target-Vektor (Attribut *target*). Erstere enthält sämtliche Features aller Versionen im Datenset, wobei jede Zeile der Matrix den Feature-Vektor einer Version darstellt. Das heisst, die Data-Matrix ist immer von der Form $(n_{versions} \times n_{features})$. Der Target-Vektor enthält die Zielwerte, also z. B. die Anzahl Bugfixes pro Monat für jede Version. Er hat die Länge $(n_{versions})$.

Anfangs war das Hauptproblem, dass sich das Einlesen der Features aus der

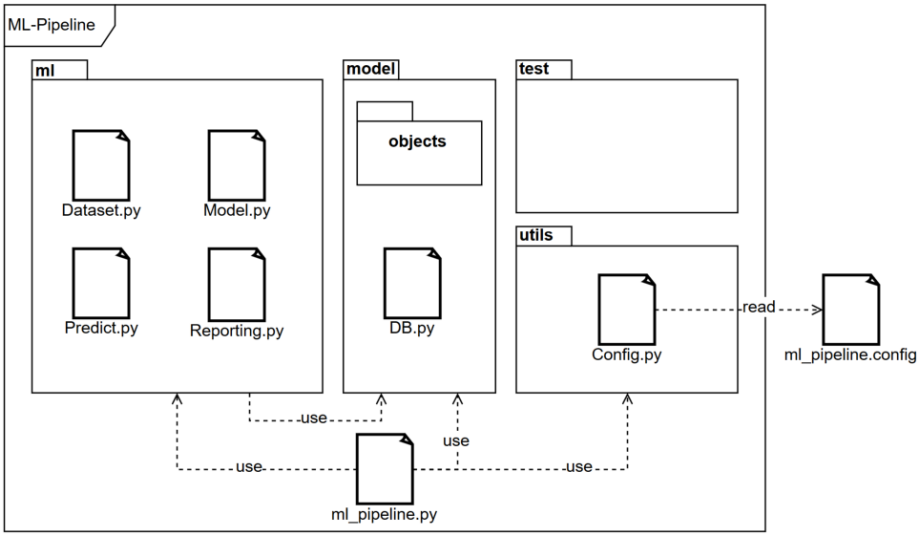

Abbildung 12: Die Package-Struktur der ML-Pipeline (grob vereinfacht)

Datenbank als sehr langsam gestaltete. Der Grund dafür war, dass alle Daten über das ORM mit Lazy Fetching geladen wurden, was zu einer immens hohen Anzahl von SQL-Queries führte. Der dadurch entstandene Mehraufwand führte zu sehr langen Ladezeiten der Datensets. Als erster Lösungsansatz wurde alles auf Eager Loading umgestellt, was bedeutete, dass alle Daten vom ORM über eine grosse Join Query geladen werden. Für kleine Datenmengen war dies sehr effizient, doch für eine grössere Anzahl von Versionen und Features bewährte sich dieser Ansatz nicht. Durch die grossen Joins wurde das Resultat der SQL-Query zu gross für den Arbeitsspeicher. Als Konsequenz wurde Eager Loading für die Features bzw. N-Grams wieder deaktiviert, für die restlichen Tabellen (*Commit*, *Version*, *UpcomingBugsForVersionMv*) wurde es beibehalten. Dies stellte sich als sinnvolle Balance zwischen Rechenzeit und Speicherverbrauch heraus und ermöglichte das Laden sowohl von grossen als auch kleinen Datensets in nützlicher Frist. Bei Bedarf lässt sich Eager Loading aber in der Konfiguration mit dem Parameter *eager_load* in der Sektion *[DATABASE]* für alle Tabellen aktivieren.

Trotz Optimierungen war die Ladezeit von Datensets hinderlich für die Entwicklung. Aus diesem Grund wurde eine Cache-Funktion für Datensets implementiert. Wenn diese aktiviert ist (über die Einstellung *cache = True* in der Sektion *[DATASET]*) werden Datensets nach dem Einlesen aus der Datenbank in einer Datei gespeichert. Sofern möglich, wird ein menschenlesbares Format gewählt. So muss ein Datenset im Idealfall nur einmal aus der Datenbank gelesen werden und kann ab diesem Zeitpunkt äusserst effizient vom Dateisystem geladen werden. Unterschieden werden die Datensets anhand ihres Dateinamen. Zum Beispiel:

Listingsverzeichnis 6: Datenset Dateiname

```
Test_2015_01_01_2015_06_30_sixmonths_ngrams_1-2-3_1-2-3-4
_sparse_a773ef13d1f7cd7710dfc2f7efe5532b.datenset
```

Diese Datei enthält ein Datenset mit dem Label "Test", umfasst den Zeitraum vom 01.01.2015 bis 30.06.2015 und hat als Target die Anzahl Bugfixes in den nächsten 6 Monaten. Sie enthält N-Grams mit der Grösse 1-3 und den Leveln 1-4. Die N-Grams/Features sind als Sparse-Matrizen gespeichert. Der letzte Teil des Dateinamens ist ein MD5-Hash über die Feature-Liste. Eine komplette Auflistung der Features hätte den Dateinamen zu lang gemacht, mit dem Hash können unterschiedliche Feature-Kombinationen dennoch unterschieden werden.

Ein weiteres Problem mit den Datensets trat auf, sobald N-Grams dazugeschaltet wurden und die Anzahl Feature-Werte damit dramatisch stieg. Bei grossen Datensets wurden bis zu 40 GB Arbeitsspeicher benötigt, was zumindest mit unserer Infrastruktur (siehe Kapitel 3.1.3) nicht mehr bewältigbar war. Da die meisten N-Grams einer Version eine grosse Anzahl von 0 aufweisen, ist die entstehende Data-Matrix eine sogenannte Sparse-Matrix (auch dünnbesetzte oder schwachbesetzte Matrix). *Scipy* bietet im Package *scipy.sparse* eine Auswahl von verschiedenen Speicherformaten für Sparse-Matrizen an, welche diesen Umstand ausnutzen.

Wenn in der Konfiguration der Parameter *sparse* in der Sektion *[DATASET]*) auf True geschaltet ist, wird die Data-Matrix im CSR-Format (Compressed Sparse Row, *scipy.sparse.dok.csr_matrix*) gespeichert. Dieses Format lässt effizientes Row Slicing sowie arithmetische Operationen zu, wie sie *scikit-learn* benötigt. Der Speicherverbrauch reduziert sich damit immens, nachteilig ist aber, dass gewisse Limitationen beim Preprocessing auftreten. So kann der *StandardScaler* von *scikit-learn*, welcher für das Feature Scaling verantwortlich ist, die Daten einer Sparse-Matrix nicht zentrieren. Auch polynomiale Features sind damit nicht mehr möglich.

Ein weiteres Problem war Anfangs der Aufbau der CSR-Matrix. Strukturänderungen stellten sich als sehr ineffizient heraus, so dass das Einlesen von N-Grams mehrere Stunden dauerte. Als Lösung wurde das Datenset zuerst mit einer DOK-Matrix (Dictionary Of Keys, *scipy.sparse.dok.dok_matrix*) initialisiert. Diese Datenstruktur erlaubte es die Matrix inkrementell aufzubauen. Wenn alle Daten eingelesen sind, wird die Matrix ins CSR-Format konvertiert. Eine weitere Optimierung war es, 0-Werte nicht in die DOK-Matrix zu schreiben. Dadurch wurde die Datenstruktur zwar nicht grösser (da 0-Werte nicht gespeichert werden), trotzdem benötigte es aber Rechenzeit, vermutlich für den Look-Up in der Matrix. Mit diesen beiden Anpassungen wurde das Einlesen mit N-Grams auch bei grossen Datensets auf wenige Minuten reduziert.

Preprocessing und Pipelines Preprocessing umfasst Transformationen der Data-Matrix, bevor das Modell darauf angepasst wird. Die ML-Pipeline unterstützt Feature Scaling und polynomiale Features (siehe Kapitel 2.1.2).

Am Anfang der Entwicklung führten wir das Preprocessing manuell auf den Datensets aus, was den Vorteil hatte, dass wir das transformierte Datenset cachen konnten. Es machte aber auch die Logik komplizierter und wir begingen Anfangs den Fehler, dass wir Trainings- und Testset separat skaliert haben, was zu sehr schlechten Ergebnissen führte.

Es stellte sich heraus, dass *scikit-learn* mit Pipelines eine simple und effiziente Lösung bietet. Pipelines können benutzt werden, um mehrere Estimators aneinanderzuketten und diese dann wie ein normales Modell benutzen zu können. [105] Als Estimator gelten sowohl Preprocessing-Komponenten als

auchML-Modelle. Allerdings müssen zwingend alle Schritte bis auf den letzten Transformatoren sein, d. h. sie müssen über eine transform()-Methode verfügen.

Pipelines lassen sich sehr einfach konstruieren:

Listingsverzeichnis 7: Python Pipeline Code

```
from sklearn import svm
from sklearn.pipeline import Pipeline
from sklearn.preprocessing.data import PolynomialFeatures
from sklearn.preprocessing.data import StandardScaler

scaler = StandardScaler()
poly = PolynomialFeatures(degree=2)
svm = svm.SVR(kernel='linear', C=0.1)

steps = [('scale', scaler), ('poly', poly), ('svr', svm)]

model = Pipeline(steps=steps)
```

Der Vorteil ist, dass auf dem konstruierten Pipeline-Modell nur noch einmal fit() und predict() aufgerufen werden muss. Der Scaling-Faktor des StandardScalers wird automatisch gespeichert und beim Aufruf von predict() auf das Input-Datenset angewendet. Ausserdem ermöglichen es Pipelines auch, Grid Search über alle Parameter der Komponenten in einem Lauf durchzuführen.

Predicting Neben der Vorhersage mit Hilfe eines trainierten Modells kann die ML-Pipeline noch drei verschiedene Baseline-Predictions erstellen. Diese sollen als naive Vergleichswerte für die Performance des Modells dienen:

1. **Mean Prediction**: Ein konstantes Modell, welches unabhängig vom Inputwert den arithmetischen Mittelwert des Target-Vektors des Trainingssets zurückgibt.
2. **Median Prediction**: Ein konstantes Modell, welches unabhängig vom Inputwert den Median des Target-Vektors des Trainingssets zurückgibt.
3. **Weighted Random Prediction**: Dieses Modell bezieht aus dem Trainingsset die Verteilung der Werte des Target-Vektors. Es wählt die Rückgabewerte zufällig, wobei häufige Werte aus dem Trainingsset-Target mit einer entsprechend höheren Wahrscheinlichkeit gewählt werden.

Reporting Das Reporting-Modul (*ml.Reporting*) bietet verschiedene Möglichkeiten zur Auswertung eines Modells an. Folgende Reports werden, in ASCII-Tabellen formatiert und in schriftlicher Form ausgegeben respektive in einer Textdatei gespeichert:

- **Config**: Alle relevanten Parameter aus der Konfiguration werden aufgelistet. Das ist wichtig um die Übersicht zu behalten und Resultate den jeweiligen Einstellungen zuordnen zu können.
- **Regressionsmetriken**: Über die Klasse *ml.Reporting.Report* können alle im Kapitel 2.2 beschriebenen Metriken erstellt werden. Diese können anschliessend nicht nur in Tabellenform ausgegeben, sondern mit der Funktion *get_report_comparisation_table* auch einfach miteinander verglichen

werden. Das ist insbesondere sinnvoll um die Scores der Baseline, des Trainingsets und des Testsets gegenüberzustellen.

- **Top-Features**: Die Funktion *get_top_features_table* listet die am Höchsten gewichteten Features des trainierten Modells auf. Das lässt sich nur auf lineare Regression, Ridge Regression und SVR mit linearem Kernel anwenden, da die Gewichtung von anderen SVR Kernelfunktionen verzerrt wird. Auch mit polynomialen Features macht dieser Report keinen Sinn, da die entstehenden Polynome nicht mehr den ursprünglichen Features zugeordnet werden können. Am Anfang der Entwicklung der ML-Pipeline war dieser Report aber sehr praktisch, da auffällige Feature-Gewichtungen auf Fehler hindeuteten.

- **Kategorische Auswertung**: In der angedachten Anwendung eines Fehlervorhersage-Tools ist ein kontinuierlicher Wert für den Benutzer vermutlich uninteressant, da es schliesslich keine "halben" Bugs gibt. Vielmehr würde er eine diskrete Auswertung erwarten. Aus diesem Grund wurde die Funktion *get_category_table* implementiert. Sie bildet die kontinuierlichen Vorhersagen eines Modells auf beliebige Kategorien ab, z. B. 0 Bugs, 1 Bug, 2-3 Bugs und 4+ Bugs. Für jede Kategorie werden die Anzahl richtiger und falscher Aussagen der Vorhersage gezählt. Das erlaubt eine realitätsnähere Auswertung des Modells.

- **Confusion-Matrize**: Da sich die kategorische Auswertung bewährte, wurde für eine detailliertere Auswertung eine Confusion-Matrix (auch *Error Matrix* [106]) implementiert. Sie wird oft bei (Multi-) Klassifikationsproblemen verwendet. Jede Spalte der Matrix repräsentiert eine Vorhersageklasse, die Zeilen sind die Klassen der Ground Truth. Damit stellt die Diagonale der Matrix die Anzahl korrekter Vorhersagen dar. Im Gegensatz zur kategorischen Auswertung lässt sich zusätzlich auch ablesen, in welchen Klassen die falschen Aussagen gefallen sind. Sie lässt sich mit der Funktion *get_confusion_matrix* generieren.

- **Classification Report**: Die Funktion *get_confusion,_matrix* gibt neben der Confusion-Matrix auch einen Classification Report aus. Denn dank der kategorischen Auswertung ist es auch möglich, die Precision und den Recall pro Kategorie zu bestimmen und damit auch einen F1-Score zu berechnen. Dieser Score wird oft bei Klassifikationsproblemen benutzt, da er es erlaubt, die Qualität einer Vorhersage an einem einzigen Wert abzulesen.

- **Ranking**: Um verschiedene Reports zu vergleichen, wurde ausserdem das Modul *ml.Scoreboard* implementiert. Dieses speichert die Resultate eines ML-Pipeline-Durchlaufs zusammen mit dessen Konfiguration in einer Datei. Wird diese später wieder ausgelesen, kann der aktuelle Lauf mit den vergangenen Läufen anhand einer beliebigen Metrik verglichen werden. Da diese Auswertung nicht idempotent ist und ohne Kontext der vorherigen Durchläufe nicht aussagekräfig ist, wird in dieser Arbeit nicht weiter darauf eingegangen. Sie stellte sich aber während dem Testen und Entwickeln einige Male als nützlich heraus.

3.5.4 Erkenntnisse

Die ML-Pipeline entwickelte sich zu einer leistungsfähigen Software, welche vergleichsweise einfach zu bedienen ist. Durch die einfache aber umfangreiche Konfigurationsmöglichkeit und die Performance-Features ermöglicht sie, Experimente vielseitig zu gestalten.

Als nachteilig erwies sich, dass neue Funktionen nicht ganz so einfach ergänzt werden können, wie wir es uns gewünscht haben, da sie zuerst in den bestehenden Aufbau eingegliedert werden müssen. Oft verlangte dies Anpassungen beim Einlesen der Konfiguration oder auch beim Datenset-Caching (da neue Attribute in die Logik aufgenommen werden müssen). Wirkliches Rapid Prototyping hätte man erreicht, indem man direkt mit *scikit-learn*-Skripts gearbeitet hätte. Parameteranpassungen bei Experimenten wären dadurch allerdings wesentlich aufwändiger geworden.

Weitere Probleme betreffen Performance und Skalierbarkeit. Beides ist mit dem jetzigen Design und der Python-Plattform limitiert. Für den Rahmen dieser Arbeit sind aber sowohl die aktuelle Performance als auch die Skalierbarkeit hinreichend.

3.6 Machine Learning

Ob der von uns verfolgte Ansatz funktioniert, zeigte sich in diversen ML-Experimenten. In diesem Kapitel sollen die Vorgehensweise und die Überlegungen zu diesen Versuchen beschrieben werden.

Da reale Projekte als Datenbasis dienten, stellte sich die Frage, wie die Fehlervorhersage validiert werden soll. Eine Möglichkeit wäre gewesen, ein Trainings- und Testset aus einer zufälligen Auswahl von Dateiversionen zusammenzustellen. Am Sinnvollsten erschien es uns aber, das Trainings- und Testset zeitlich aufzuteilen. Dies richtet sich an den angedachten Anwendungsfall, in dem ein Tool aus einer bestimmten Zeitspanne der Vergangenheit lernt und dann Voraussagen für die Zukunft machen wird. Ein Beispiel soll dies verdeutlichen:

Als Trainingszeitraum wird der 01.08.2014 bis 31.12.2014 verwendet. Auf dem Trainingsset gewichtet der ML-Algorithmus die einzelnen Features um dann anschliessend Bug-Vorkommnisse für das nächste halbe Jahr zu prognostizieren. Eine zweite, etwas kürzere Zeitspanne, zum Beispiel 01.01.2015 bis 31.01.2015, wird als Testset verwendet. Da auch das Testset in der Vergangenheit liegt, ist die Anzahl Bugs bereits bekannt. Dieser Wert dient als Ground Truth, mit welcher die Vorhersage des ML-Modells verglichen wird. Die Abbildung 13 soll das Beschriebene verdeutlichen.

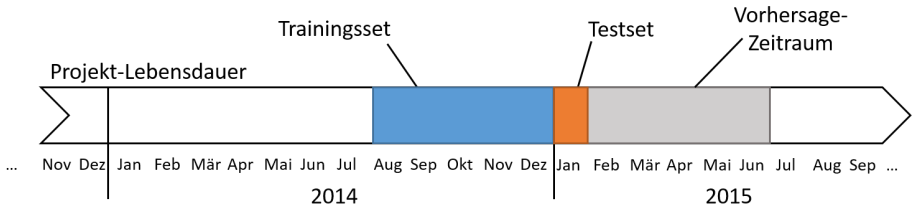

Abbildung 13: Phasen der ML-Experimente

Um eine repräsentative Aussage über die Leistung des Toolsets und der verwendeten Features machen zu können, sind eine grosse Anzahl an Experimenten notwendig. Aufgrund des Zeitplans war es uns nicht möglich, Experimente in diesem Umfang durchzuführen. Stattdessen verglichen wir mithilfe einer kleine Auswahl an Experimenten die Performance der verschiedenen ML-Modelle miteinander. Insbesondere sollte geprüft werden, ob die von uns definierten Baselines (siehe Kapitel 3.5) übertroffen werden können. Damit wollten wir untersuchen ob das in dieser Arbeit beschriebene Verfahren ansatzweise funktioniert und welche Algorithmen die besten Ergebnisse liefern.

Ausserdem wurden die Effektivität der einzelnen Feature-Gruppen analysiert. Dazu wurden bei identischer Konfiguration einzelne Feature-Gruppen deaktiviert und geprüft, wie sich dadurch die Vorhersagen ändert (sog. Ablation-Tests).

Gemeinsame Parameter Die gemeinsamen Parameter der Versuche werden in Tabelle 13 aufgeführt.

Zeitraum Traingsset:	01.10.2014 - 31.12.2014
Zeitraum Testset:	01.01.2015 - 31.01.2015
Repository:	*Elasticsearch*
Cross Validation:	Ja
Ridge Regression α:	1, 0, 0.01, 0.1, 10, 100, 1000
SVR C:	1, 0.001, 0.01, 0.1, 5, 10, 50, 100
SVR ϵ:	0.1
SVR γ:	auto
SVR *degree*:	3
SVR *coef0*:	0

Tabelle 13: Parameter der ML-Experimente

Die Trainingszeiträume von uns sind so ausgewählt worden, dass sie in vernünftiger Zeit terminieren. Die Laufzeit hängt natürlich von den gewählten Parametern (vor allem dem ML-Algorithmus, mit/ohne N-Grams und CV) ab.

4 Resultate

In diesem Kapitel werden die Resultate aller relevanten Experimente und deren Auswertungen aufgelistet und beschrieben. Auf die beim Repository Mining gewonnenen Erkenntnisse wird im Kapitel 4.2 eingegangen, das Kapitel 4.3 behandelt die Resultate der Versuche mit der ML-Pipeline.

4.1 Testdaten

Als Basis der nachfolgenden Experimente und Auswertungen diente das Softwareprojekt *Elasticsearch* [107], welches auch anderweitig als primäre Datenquelle für diese Arbeit diente.

Elasticsearch ist ein auf *GitHub* gehosteter open-source Search-Engine-Server [108] auf Basis von *Apache Lucene*. Das Java-Projekt wird seit Februar 2010 [109] von Shay Banon als Hauptentwickler gepflegt und steht unter der Apache License. Es ermöglicht das Durchsuchen von beliebigen Dokumenten (fast) in Echtzeit über ein verteiltes System von Indizes und unterstützt Features wie Multitenancy und eine extensive REST- und Java-API. *Elasticsearch* ist die momentan meistverbreitetste Enterprise Search Engine [110] und wird unter anderem von *Wikimedia* [111], *GitHub* [112], *Stack Exchange* [113], *Netflix* [114] und *SoundCloud* [115] genutzt.

Damit ist *Elasticsearch* nicht nur allgemein ein spannendes Projekt, es hat sich auch wegen seines Alters (über 6 Jahre), seiner Projektgrösse (ca. 20'000 Commits) und seinem gut gepflegten Issue-Tracking (ca. 10'000 Issues auf *GitHub*) ideal als Testprojekt für diese Arbeit angeboten.

4.2 Repository Mining

Im folgenden Kapitel versuchten wir, mit statistischen Mitteln erste Erkenntnisse zu gewinnen und ein Gefühl für die gesammelten Daten zu erhalten. Dabei eröffneten sich nicht nur interessante Einblicke in die Projektstruktur von *Elasticsearch*, es gelang auch bereits erste Korrelationen zum Auftreten von Bugs zu erkennen.

4.2.1 Projektstatistiken

Projektwachstum Abbildung 14 zeigt, wie das *Elasticsearch*-Projekt seit seiner Entstehung gewachsen ist. Zu erkennen ist hier, dass der initiale Commit am 01.02.2010 bereits ca. 1400 Files hunzugefügt hat. Dieser Initial-Commit fand gerade mal eine Woche vor dem offiziellen Release [109] statt, deshalb ist anzunehmen, dass Shay Banon eine bereits lauffähige Version in das heute benutzte *Git*-Repository lud. Solch grosse Initial-Commits lassen sich auch in anderen Projekten beobachten.

Ebenfalls auffällig ist, dass Anfangs die Projektgrösse, also der benötigte Speicherplatz, ungefähr proportional zur Anzahl Files wächst. Ab Anfang 2015 steigt die Projektgrösse allerdings deutlich schneller an, als die Anzahl Files. Vermutlich heisst das, dass ab dieser Zeit vermehrt bestehende Klassen erweitert anstatt Neue erstellt wurden. Die Analyse von File-Stichproben im Kapitel 4.2.2 unterstützt diese Vermutung.

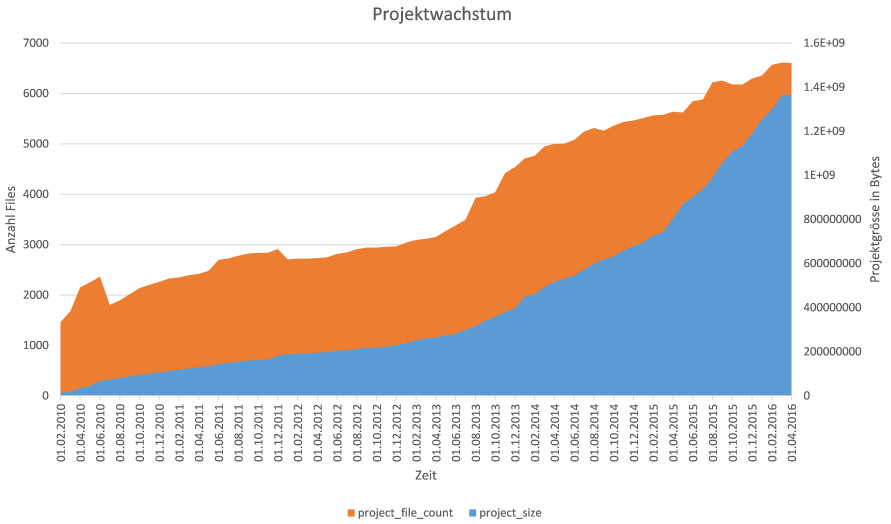

Abbildung 14: Projektgrösse von *Elasticsearch* über die Zeit

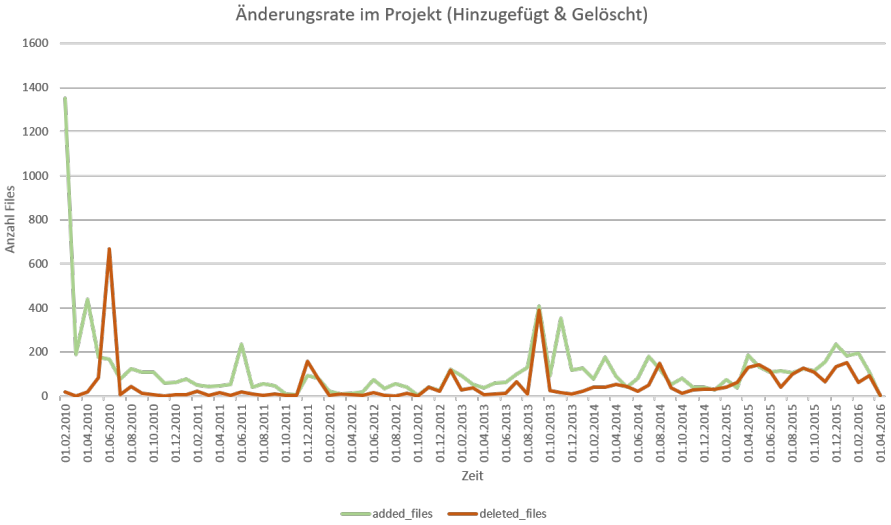

Abbildung 15: Anzahl hinzugefügter/gelöschter Files im Projekt *Elasticsearch* über die Zeit

Zuletzt lassen sich in den Abbildungen 15 und 16 auffällige Spitzen feststellen. Diese zeigen grössere Refactorings und Umstellungen an Core-Komponenten an. Ebenfalls interessant ist, dass sich die Anzahl der geänderten Files mit der Zeit erhöht. Das deutet darauf hin, dass Änderungen mit der Zeit immer mehr Files beeinflussen, möglicherweise durch mehr Abhängigkeiten. Die Anzahl geänderter Files steigt signifikant Anfang 2015, also zeitgleich wie der Anstieg des Projektgrössen-Wachstums.

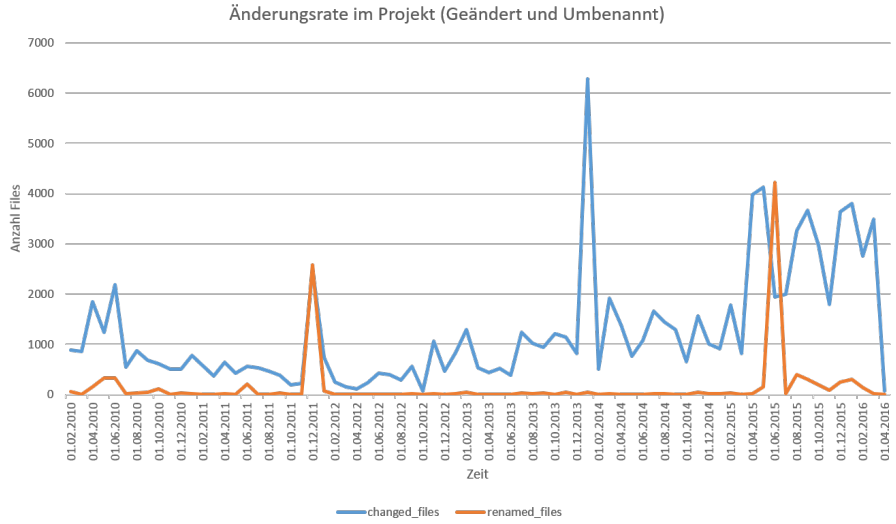

Abbildung 16: Anzahl geänderter/umbenannter Files im Projekt *Elasticsearch* über die Zeit

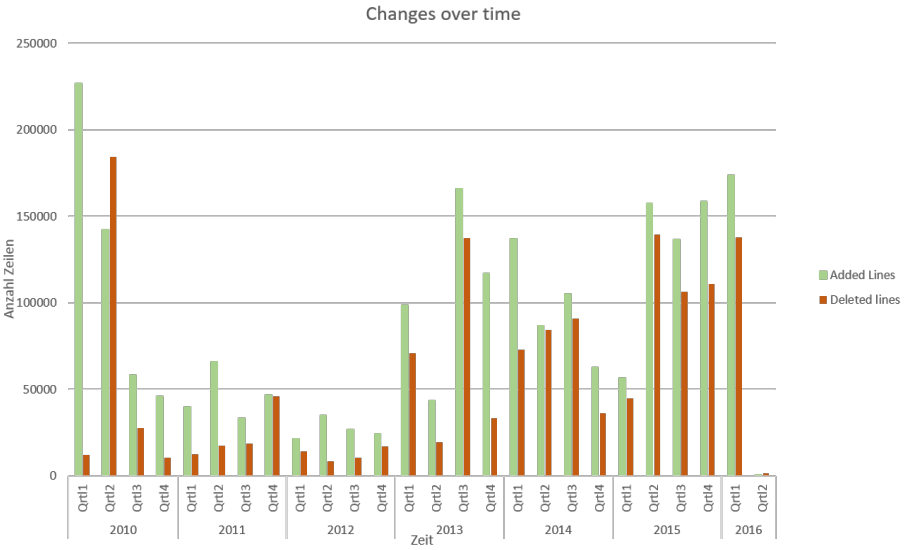

Abbildung 17: Anzahl geänderter Zeilen im Projekt *Elasticsearch* über die Zeit

Zeilenänderungen In Abbildung 17 wird die Menge an total hinzugefügten und gelöschten Zeilen dargestellt. Gut erkennbar ist, dass beide Werte einigermassen proportional zueinander sind. Das war auch zu erwarten, insbesondere da in *Git* eine Zeilenänderung sowohl als gelöschte Zeile (die Zeile im alten Zustand) sowie auch als hinzugefügte Zeile (die Zeile im neuen Zustand) zählt. Ebenfalls zu erkennen ist, dass durchgehend mehr Zeilen hinzugefügt als gelöscht werden, was zum bereits beschriebenen Projektwachstum führt.

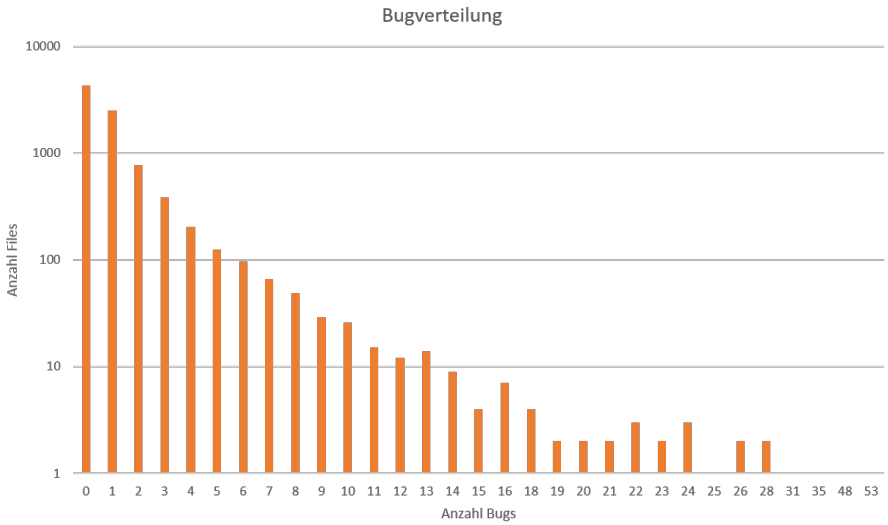

Abbildung 18: Anzahl Files pro Anzahl Bugs

Anzahl Bugs	**Anzahl Files**
0	4265
1	2487
2	774
3	384
⋮	⋮
53	1
Total	8593

Tabelle 14: Auszug von Anzahl Files pro Anzahl Bugs

Total Files	8593
Total Bugs	10043
Durchschnitt	1.17 Bugs/File
Median	1 Bug

Tabelle 15: Anzahl Bugs zusammengefasst

Bugverteilung über Files Abbildung 18 stellt die Verteilung von Bugs über die gesamte Lebenszeit über alle Java Files dar. Zu beachten ist, dass die Y-Achse (Anzahl Files) logarithmisch skaliert ist. Als Bug gilt jede Änderung, die ein File erfahren hat, welche über das Issue-Tracking mit einem Bug-Issue in Verbindung gebracht wurde.

Tabelle 14 zeigt einen Auszug der gesammelten Daten. Insgesamt umfasst die Statistik 8593 Java-Files aus dem *Elasticsearch*-Projekt. Davon werden 4265 Files, also 49.6%, nie mit einem Bug in Verbindung gebracht. Von den restlichen 50.4% aller Files sind wiederum 58.3% im Zuge eines Bugfixes geändert worden. Files mit über 10 Bugs in ihrer Lebenszeit sind in diesem Projekt äusserst selten, sie machen gerade mal 1.33% aller Java-Files aus. Im Durchschnitt fallen 1.17

Bugs auf ein File, der Median liegt bei einem Bug.

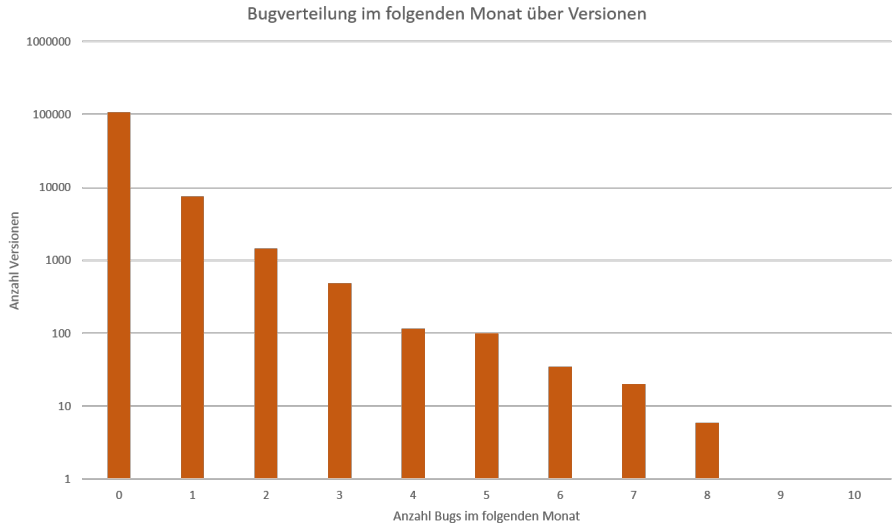

Abbildung 19: Anzahl Versionen pro Anzahl Bugs im folgenden Monat

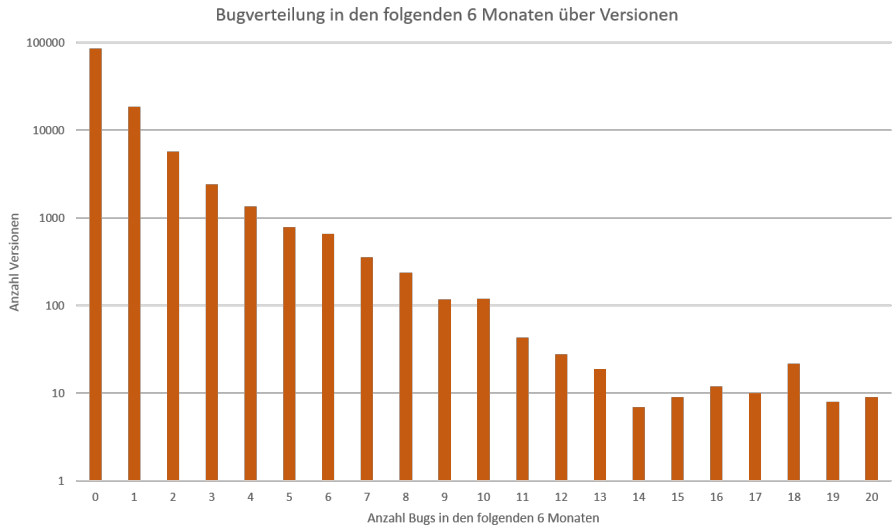

Abbildung 20: Anzahl Versionen pro Anzahl Bugs in den folgenden 6 Monaten

Bugverteilung über Versionen Die Bugverteilung über die gesamte Lebenszeit von Files ist ein interessantes Mass, welches einiges über die Qualität eines Projekts verrät. Für den Zweck dieser Arbeit viel interessanter ist jedoch die Verteilung von *aufkommenden* Bugs pro *Version*.

So zeigen die Abbildungen 19, 20 und 21 die Verteilung von kommenden Bugs pro Version für den folgenden Monat resp. die folgenden 6 Monate und das

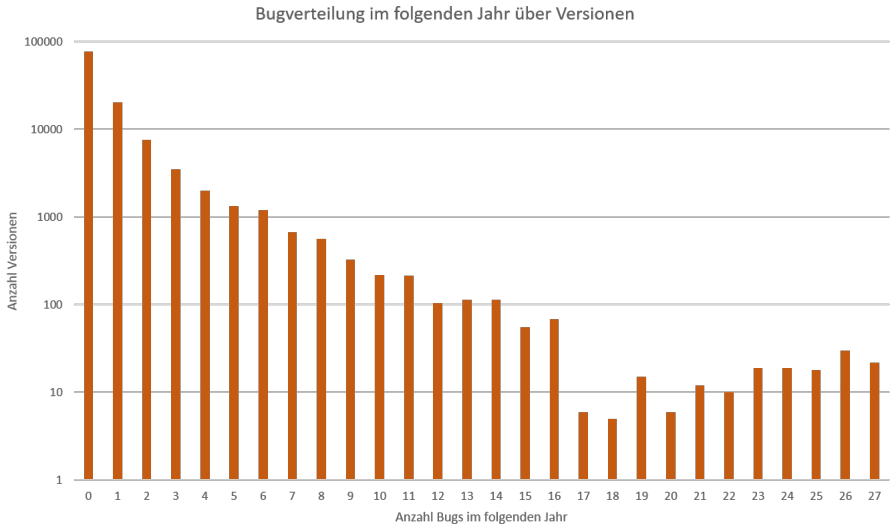

Abbildung 21: Anzahl Versionen pro Anzahl Bugs im folgenden Jahr

folgende Jahr. Als aufkommende Bugs für eine Version gelten alle zukünftigen Commits in der jeweiligen Zeitspanne, welche auf dasselbe File verweisen und über das Issue-Tracking als Bugfix markiert sind.

Es ist auch hier wichtig zu beachten, dass die Y-Achse dieser drei Grafiken logarithmisch skaliert ist. Es ist auffallend, dass die Verteilung so annähernd linear wird, was auf eine Poisson-Verteilung hindeutet. Die Ungenauigkeiten könnten einfach von Ausreissern stammen, es wäre aber möglich, dass der Grund dafür eine Nicht-Unabhängigkeit der Ereignisse ist. Wir vermuten, dass ein Bugfix die Wahrscheinlichkeit für weitere Bugfixes wesentlich beeinflusst, was der Grund für das Implementieren der temporalen Features war (siehe Kapitel 3.4.5). Die Beobachtung dieser annähernden Poisson-Verteilung führte zur Idee, einen Log-Transform auf dem Target-Vektor auszuführen. Dies wird im Kapitel 4.3.4 genauer beschrieben.

Ein weiterer interessanter Punkt ist, dass sich die Verteilung über die verschiedenen Zeitspannen nicht massgeblich unterscheidet. Erwartungsgemäss gibt es bei grösseren Zeitspannen aber mehr Ausreisser die eine sehr hohe Buganzahl aufweisen.

Verteilung von Changes Mit den Abbildungen 22 und 23 wird die Verteilung von Zeilenänderungen dargestellt. Die Grafiken zeigen, dass bei den meisten Files über ihre gesamte Lebensdauer zwischen 30 und 100 Zeilen hinzugefügt werden. Das impliziert auch, dass die meisten Files nicht grösser als 100 Zeilen sind. Die Verteilung der gelöschten Zeilen zeigen, dass bei vielen Files nur wenig gelöscht wird. An dieser Stelle sei nochmals darauf hingewiesen, dass auch eine Zeilenänderung als gelöschte Zeile gilt.

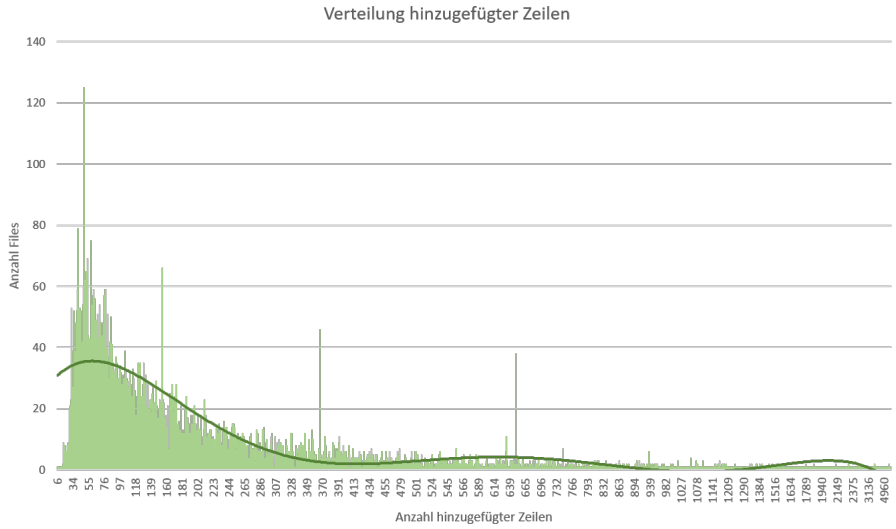

Abbildung 22: Anzahl Files pro Anzahl hinzugefügter Zeilen

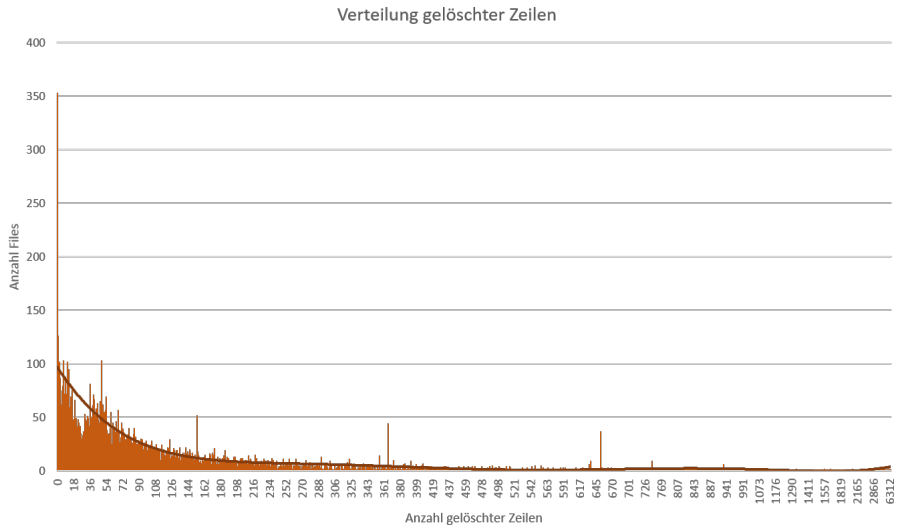

Abbildung 23: Anzahl Files pro Anzahl gelöschter Zeilen

4.2.2 Stichproben von Java Files

Im Folgenden werden einige ausgewählte Java-Dateien des Projekts *Elasticsearch* genauer analysiert.

Die "Issues über Zeit"-Statistiken zeigen, wie die verschiedenen Issues über die Zeitachse der Datei-Lebenszeit verteilt sind.

InternalEngine.java Diese Java-Klasse wurde ausgewählt, weil sie die höchste Bug-Anzahl (53) aufweist. Da diese Datei eine Kernkomponente ist, welche

seit dem Initial-Commit im Projekt vorhanden ist, bestehen vermutlich auch viele Abhängigkeiten, was dazu führte, dass sie mit eben so vielen Bugs in Verbindung gebracht wird. Gerade deswegen bietet sie aber die beste Grundlage für eine genauere Analyse.

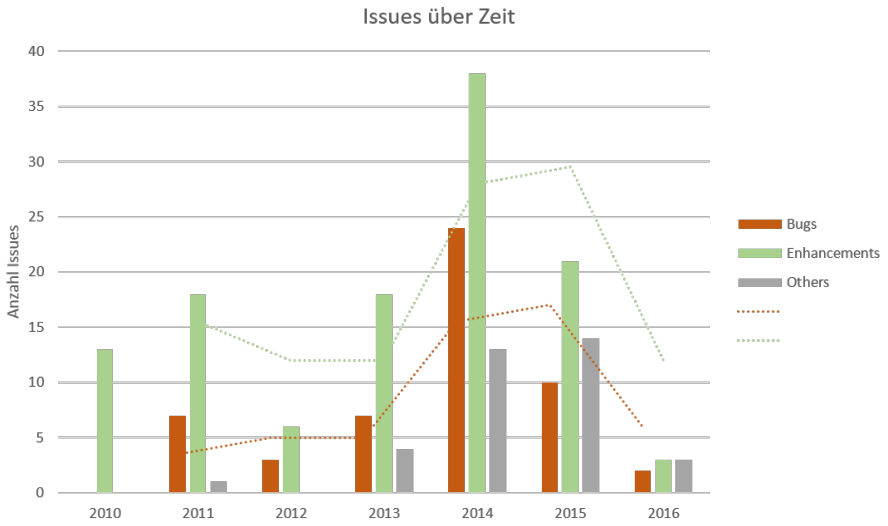

Abbildung 24: Anzahl Enhancements und Bugs, welche auf InternalEngine.java referenzieren, über die Zeit

Stellt man die Anzahl Issues, gruppiert nach ihrem Typ (Enhancement und Bug), auf einer Zeitachse dar (siehe Abbildung 24), lässt sich feststellen, dass die Anzahl Bugs pro Jahr proportional zur Anzahl Enhancements im selben Jahr ist. Dies bestätigt die Vermutung, das Enhancements neue Programmfeatures hinzufügen, in welchen Bugs vorhanden sind.

Sieht man sich die Verteilung in einem Jahr genauer an (in Abbildung 25 das Jahr 2014) lässt sich das sogar noch besser aufzeigen. Auf eine grosse Anzahl von Enhancements folgt, um ca. 3 Monate versetzt, eine ebenso grosse Anzahl von Bugs. Dasselbe Phänomen kann auch in anderen Jahren beobachtet werden. Auf jede “Welle” von Enhancements folgt kurze Zeit später eine ähnliche “Welle” von Bugs.

ChildQuerySearchIT.java Dieses File weist mit 31 Bugs ebenfalls eine sehr hohe Anzahl von Bugs auf und ist deshalb für eine Analyse gut geeignet.

In der Abbildung 26 sind die zwei Spitzen der Bug-Anzahl Ende 2013 und Mitte 2014 auffällig. Ausserdem ist hier gut ersichtlich, dass ein Source File tendenziell am Anfang seiner Lebenszeit mehr Bugs aufweist, was darauf zurückzuführen ist, dass dann auch am meisten neuer Code geschrieben wird. Hier sind die meisten Bugs, inklusive den beiden Ausreissern im ersten Drittel der Lebenszeit des Files zu verzeichnen. Danach pendelt es sich ein und zeigt das bereits erwähnte Muster von Enhancement-Wellen mit kurz darauffolgenden Wellen von Bugs.

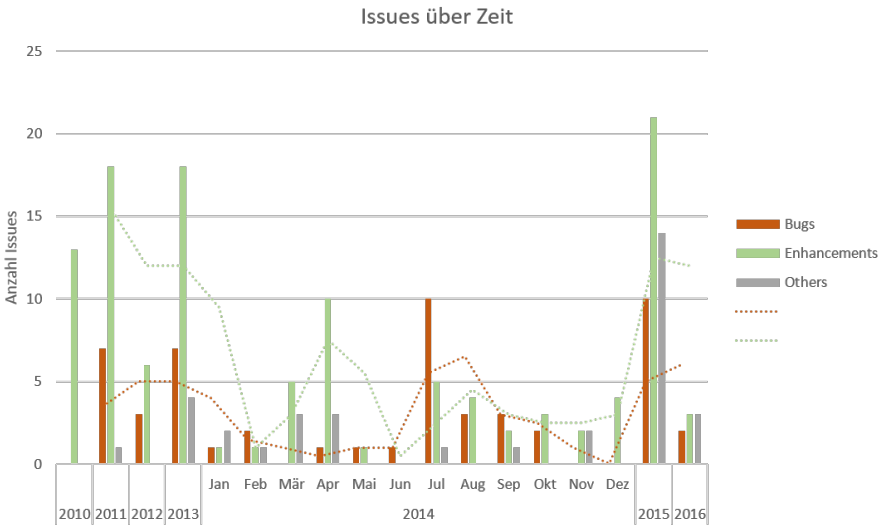

Abbildung 25: Anzahl Enhancements und Bugs, welche auf InternalEngine.java referenzieren, über die Zeit, Fokus auf 2014

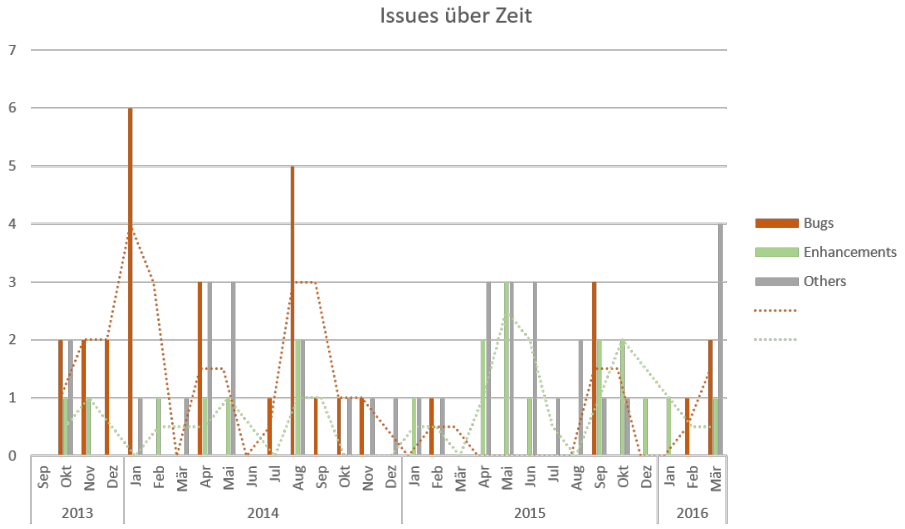

Abbildung 26: Anzahl Enhancements und Bugs, welche auf ChildQuerySearchIT.java referenzieren, über die Zeit

4.3 Machine Learning

In diesem Kapitel werden die Resultate der im Kapitel 3.6 beschriebenen Experimente aufgeführt und besprochen. Als erstes wurden verschiedene ML-Modelle miteinander und gegenüber einer Baseline verglichen, anschliessend wurde die Signifikanz der verschiedenen Feature-Gruppen zu bestimmen versucht.

Als Metriken zur Bewertung der ML-Modelle wurden der Mean Absolute

Error (MAE), Median Absolute Error (MDE) und der R^2-Score verwendet. Für die Bedeutung dieser Werte verweisen wir auf das Kapitel 2.2, wo sie kurz beschrieben werden.

Aufgrund von zeitlichen Einschränkungen wurden alle der folgenden Experimente auf nur einer Trainings- und Testset Kombination durchgeführt (siehe Tabelle 16). Dies schränkt die Aussagekraft der erhaltenen Resultate natürlich ein. Bei einer weiterführenden Arbeit müsste noch mit vielen anderen Daten getestet werden. Vorschläge, wie man diese Aufgabe angehen könnte, besprechen wir im Kapitel 5.2.5.

Projekt:	*Elasticsearch*
Target:	Bugs in den folgenden 6 Monaten
Trainingsset:	01.10.2014 - 31.12.2014 (620 Commits mit 3414 Dateiversionen)
Testset:	01.01.2015 - 31.01.2015 (165 Commits mit 950 Dateiversionen)

Tabelle 16: Verwendete Testdaten für Resultate

4.3.1 Baseline

Um die Performance der erstellten ML-Modelle überprüfen zu können, wurden deren Resultate mit drei verschiedenen Baseline-Predictions verglichen. Die Implementation dieser Baselines wurde im Kapitel 3.5.3 beschrieben.

Zu beachten ist, dass sich diese Baselines auf die Ergebnisse des Testsets beziehen, da mit diesem die Qualität eines ML-Modells gemessen wird. Das heisst, dass der Mean, der Median und die Target-Gewichte über die Ground Truth des Trainingsset berechnet wurden. Dann wurde die Vorhersage dieser Baseline-Modelle mit der Ground Truth des Testsets verglichen und daraus die Metriken in Tabelle 17 berechnet.

Baseline	**MAE**	**MDE**	$\mathbf{R^2}$
Mean	1.4145	0.9060	-0.0131
Median	1.1895	0.0000	-0.2314
Weighted Random	1.6411	1.0000	-0.6142

Tabelle 17: Ergebnisse der Baselines

Keine der Baselines erzielt ein besonders gutes Ergebnis. So sind alle R^2-Scores negativ. Das heisst, sie bilden die Varianz der Ground Truth sehr schlecht ab, was aber gerade bei den statischen Modellen Mean und Median zu erwarten war. Wir hätten ein besseres Resultat der Weighted Random Baseline erwartet, da sie eigentlich mehr Informationen in ihr Modell einfliessen lässt. Auffallend ist weiterhin, dass die Median-Baseline einen MDE von 0 hat. Der Grund dafür ist, dass der Median des Target-Vektors der Ground Truth in der Regel auch 0 ist, da die meisten Dateiversionen keinen Bug in den folgenden 6 Monaten erwarten. Somit ist die Vorhersage der Median-Baseline für die meisten Datensätze korrekt. Der MAE ist aber trotzdem eher hoch. Zuletzt ist noch anzumerken, dass die Weighted Random Baseline naturgemäss variiert. Dies allerdings nicht so stark, dass es für die Anwendung als Baseline signifikant wäre.

4.3.2 Vergleich der ML-Modelle

In einer ersten Phase der Experimente wurden verschiedene Konfigurationen der möglichen ML-Modelle getestet und miteinander verglichen.

Resultate ohne N-Grams Die Tabelle 18 zeigt die erhaltenen Resultate, welche *ohne* die Einbeziehung von N-Grams gewonnen wurden.

Erwartungsgemäss hat lineare Regression mit Abstand am schlechtesten abgeschnitten. Es gelingt zwar, das Modell den Trainingsdaten anzupassen, das Modell generalisiert aber sehr schlecht. Wieso die Ergebnisse auf dem Testset dermassen schlecht sind, ist uns nicht ganz klar. Wir vermuten, dass das Modell aufgrund linear voneinander abhängigen Features stark verzerrt ist und deshalb auf unbekannten Daten unsinnige Ergebnisse produziert.

Ridge Regression wurde in zwei Konfigurationen getestet. Einmal ohne polynomiale Features (Ridge Regression PD1) und einmal mit polynomialen Features 2. Grades (Ridge Regression PD2). Bereits ohne polynomiale Features zeigt sich, dass die Regularisierung die Resultate auf dem Testset gegenüber Linearer Regression massiv zu verbessern vermag. Polynomiale Features 2. Grades verbessern die Performance von Ridge Regression noch mehr und können bereits brauchbare Resultate liefern. Ein Nachteil ist hier der hohe Speicherbedarf. Es wurde versucht, polynomiale Features 3. Grades einzusetzen, dies führte aber bereits zu einem Speicherbedarf von über 40 GB, womit unsere Infrastruktur überfordert war. Es ist anzunehmen, dass sich bei grösseren Trainingssets auch schon der 2. Grad als zu speicherhungrig herausstellt.

SVR benötigt zwar nicht so viel Speicher, dafür aber bedeutend mehr Rechenzeit. SVR linear, also ohne Kernelfunktion, schneidet bedeutend besser ab als lineare Regression. Grund dafür ist wie bei Ridge Regression mehrheitlich die eingesetzte Regularisierung. Die Performance von SVR polynomial liegt etwas unter Ridge Regression mit polynomialen Features 2. Grades. Dafür war hier auch der Einsatz von Grad 3 möglich, ohne dass dies zu einem Memory Error geführt hat. Die besten Resultate erzielte eindeutig SVR mit RBF-Kernel. Dieses Modell führte sowohl auf dem Trainings- als auch auf dem Testset zu akzeptablen Ergebnissen mit einem sehr hohen Bestimmtheitsmass und einem mittleren Fehler < 1. Der Sigmoid-Kernel konnte hingegen nicht überzeugen und produzierte das zweitschlechteste Resultat nach Linearer Regression.

Resultate mit N-Grams Nachdem mit den herkömmlichen Features getestet wurde, wurden zusätzlich noch N-Grams aktiviert. Das erfordete den Einsatz von Sparse-Matrizen, ansonsten wurde die Konfiguration der jeweiligen Modelle identisch belassen. Die Resultate dieser Durchläufe sind in Tabelle 19 zusammengestellt.

Lineare Regression führte auch hier zu äussert schlechten Ergebnissen. Im Unterschied zu den Experimenten ohne N-Grams reichte hier die Regularisierung von Ridge Regression (PD1) aber scheinbar nicht, um das Resultat zu verbessern. Da sich das Modell aber scheinbar sehr gut auf das Trainingsset angepasst hat, ist hier von sehr starkem Overfitting auszugehen. Bei einem weiteren Versuch müsste man versuchen, den Regularisierungsparameter C noch weiter zu verringern.

Das Erzeugen von polynomialen Features (für Ridge Regression PD2) war mit N-Grams nicht möglich, da dies aus der Sparse-Matrix eine Dense-Matrix

Modelltyp	Set	MAE	MDE	$\mathbf{R^2}$
Lineare Regression	Training	0.7929	0.5200	0.5937
	Test	$4.035 \cdot 10^9$	5.9635	$-1.744 \cdot 10^{20}$
Ridge Regression PD1	Training	0.7858	0.4980	0.5600
	Test	1.1120	0.6866	0.3960
Ridge Regression PD2	Training	0.3702	0.2127	0.8914
	Test	1.0142	0.6385	0.6103
SVR Linear	Training	0.3126	0.1000	0.8387
	Test	0.9532	0.5141	0.5284
SVR Polynomial	Training	0.4267	0.1000	0.7170
	Test	0.8795	0.3658	0.5639
SVR RBF	Training	0.3084	0.1001	0.8439
	Test	0.8343	0.4732	0.7004
SVR Sigmoid	Training	0.7174	0.2351	0.3861
	Test	0.9669	0.2951	0.2697

Tabelle 18: Performance der verschiedenen ML-Modelle *ohne* Einsatz von N-Grams

erzeugt hätte. Wenn dann daraus noch Polynome erstellt würden, stiege der Speicherbedarf enorm.

Wie auch bei Ridge Regression vermochte die Regularisierung von SVR Linear hier nicht, das Resultat auf dem Testset im Gegensatz zu Linearer Regression signifikant zu verbessern. Wir erhofften uns, dass SVR Polynomial sinnvolle Resultate erzeugen würde, da wie bereits erwähnt polynomiale Features mit N-Grams nicht möglich waren. Leider wies dieses Modell eine enorm lange Laufzeit auf, so dass es nach über 27 Stunden terminiert werden musste, ohne dass es Resultate produziert hatte.

Genau wie bei den Experimenten ohne N-Grams erwies sich auch hier der RBF-Kernel als der Effektivste. Er terminierte nicht nur in sinnvoller Zeit, sondern produzierte auch ein sinnvolles Ergebnis. Der Sigmoid-Kernel erwies sich abermals als nicht nützlich für diesen Anwendungsfall.

Leider können diese Resultate nur beschränkt einen positiven Nutzen von N-Grams zeigen. Einzig der MDE erfährt eine Verbesserung. Der R^2-Score ist auf dem Testset deutlich schlechter als beim selben Experiment ohne N-Grams. Hingegen ist der R^2-Score auf dem Trainingsset sehr hoch. Dies deutet eigentlich auf Overfitting hin, die Validationskurve (siehe Abbildung 28) spricht da aber eher dagegen. Die Lernkurve (siehe Abbildung 27) unterstützt die Vermutung, dass für N-Grams schlicht mehr Trainingsdaten benötigt werden. Ebenfalls anzumerken ist, dass aufgrund der langen Laufzeiten bei diesen Experimenten auf das Einbeziehen von weiteren Parametern wie *gamma* und *epsilon* in die Cross Validation verzichtet wurde. Mit dem Finden von besseren Parametern könnte das Ergebnis vermutlich verbessert werden.

Laufzeiten Um die Modelle zu vergleichen, wurden auch die Laufzeiten für das Trainieren, resp. das Predicten erfasst. Denn für einen sinnvollen Einsatz einer Fehlervorhersage muss auch gewährleistet sein, dass die Lösung in nützlicher Zeit terminiert und Vorhersagen machen kann.

Die Tabelle 20 zeigt die Laufzeit der verschiedenen ML-Konfigurationen der

Modelltyp	Set	MAE	MDE	R^2
Lineare Regression	Training	0.0722	0.0354	0.9939
	Test	$5.112 \cdot 10^{10}$	$1.185 \cdot 10^{10}$	$-2.088 \cdot 10^{21}$
Ridge Regression PD1	Training	0.1148	0.0422	0.9838
	Test	1134.684	0.5785	$-9.413 \cdot 10^{7}$
Ridge Regression PD2	Training	N/A	N/A	N/A
	Test	N/A	N/A	N/A
SVR Linear	Training	0.1310	0.1000	0.9749
	Test	916.5596	0.5371	$5.387 \cdot 10^{7}$
SVR Polynomial	Training	N/A	N/A	N/A
	Test	N/A	N/A	N/A
SVR RBF	Training	0.1320	0.0999	0.9774
	Test	0.8688	0.3723	0.4794
SVR Sigmoid	Training	0.9404	0.1000	-0.1846
	Test	1.2143	0.1000	-0.1942

Tabelle 19: Performance der verschiedenen ML-Modelle *mit* Einsatz von N-Grams

Modelltyp	Training	Prediction
Lineare Regression	1 s	<1 s
Ridge Regression PD1	5 s	<1 s
Ridge Regression PD2	3 s	<1 s
SVR Linear	1 h 3 min 25 s	2 min 3 s
SVR Polynomial	54 s	4 s
SVR RBF	2 min 30 s	3 s
SVR Sigmoid	2 min 40 s	3 s

Tabelle 20: Laufzeit der verschiedenen ML-Modelle *ohne* Einsatz von N-Grams

ersten Experimentserie ohne N-Grams. Lineare Regression und Ridge Regression sind erwartungsgemäss sehr schnell. Mit SVR performt der polynomial-Kernel am Besten, RBF und Sigmoid benötigen vergleichsweise eher lange. Überraschend ist hier die unverhältnismässig lange Laufzeit von SVR Linear. Da hier keine Kernel-Funktion eingesetzt wird, wäre eigentlich eine kürzere Laufzeit als bei den restlichen SVR-Konfigurationen zu erwarten gewesen. Wir haben keine gute Erklärung für dieses Verhalten. Es ist nicht auszuschliessen, dass es sich hierbei um eine Eigenheit der Implementation von *scikit-learn* handelt.

Werden N-Grams miteinbezogen steigt die Laufzeit mit allen ML-Modellen signifikant an. Ridge Regression mit polynomialen Features (PD2) konnten wie bereits erklärt aufgrund von Speichermangel nicht eingesetzt werden. Auch hier benötigte SVR Linear äusserst lange. Mit N-Grams hat aber die mit Abstand längste Laufzeit SVR mit einem polynomial-Kernel. Wir mussten den Lernvorgang nach über 27 Stunden abbrechen. Vermutlich liegt das an der viel höheren Zahl an Features, welche mit N-Grams entstehen. SVR mit RBF-Kernel hat zwar mit fast einer Stunde auch eher lange, damit lässt sich aber arbeiten. Mit N-Grams wird ausserdem deutlich, dass SVR-Modelle im Vergleich zu Linearer und Ridge Regression deutlich länger für Vorhersagen benötigen.

Modelltyp	Training	Prediction
Lineare Regression	54 s	2s
Ridge Regression PD1	14 min 29 s	21 s
Ridge Regression PD2	N/A	N/A
SVR Linear	13 h 33 min 3 s	12 min 42 s
SVR Polynomial	>27 h	N/A
SVR RBF	55 min 8 s	13 min 26 s
SVR Sigmoid	47 min 9 s	10 min 35 s

Tabelle 21: Laufzeit der verschiedenen ML-Modelle *mit* Einsatz von N-Grams

Details zu SVR mit RBF-Kernel Da SVR in Kombination mit einem RBF-Kernel zu den besten Resultaten führte, soll im Folgenden noch etwas detaillierter auf diese Ergebnisse eingegangen werden.

Abbildung 27 zeigt die Lernkurve der Durchläufe ohne und mit N-Grams. Sie zeigt den erreichten R^2-Score des Modells in Bezug auf die Grösse des Trainingssets. Hier fällt auf, dass sowohl der Score auf dem Trainingsset als auch der CV-Score fast linear steigt. Dies deutet darauf hin, das mit noch grösseren Datensets bessere Ergebnisse zu erwarten sind.

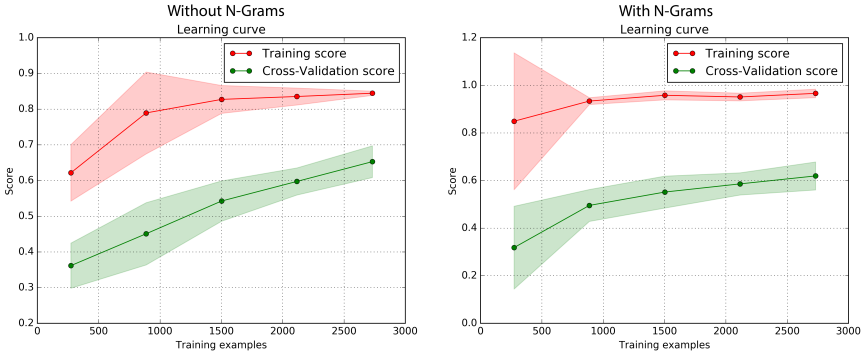

Abbildung 27: Lernkurve: (R^2)-Score nach Grösse des Trainingssets, SVR-Modell mit RBF-Kernel. Links: ohne N-Grams. Rechts: mit N-Grams

Abbildung 28 stellt die Validationskurve ohne und mit N-Grams dar. Sie zeigt den erreichten R^2-Score des Modells in Bezug auf den Regularisierungsparameter C. Beide Kurven erreichen den optimalen Parameter zwischen 1 und 50. Sie flachen sowohl links als auch rechts ab, was darauf hindeutet, dass eine breitere Auswahl an Parametern keinen grossen Nutzen hätte. Auffallend ist ausserdem, dass ohne N-Grams bei grösseren Cs Overfitting auftritt. Dieses kann alleine durch Regularisierung scheinbar nicht verhindert werden, da bei kleineren Cs sowohl der Test- als auch der CV-Score schlechter wird.

Die beiden Tabellen 22 und 23 zeigen die Confusion-Matrix für die Test-Prediction ohne resp. mit N-Grams. Daran lässt sich die Verteilung der Vorhersagen ablesen. Um dies zu ermöglichen, wurden die vorhergesagten Werte in Kategorien von 0 Bugs, 1 Bug, 2-3 Bugs und >4 Bugs aufgeteilt. Dabei wurde natürlich gerundet. In der Diagonale, wo die Prediction der Ground Truth entspricht, befinden sich die korrekten Vorhersagen. In fast allen Kategorien werden

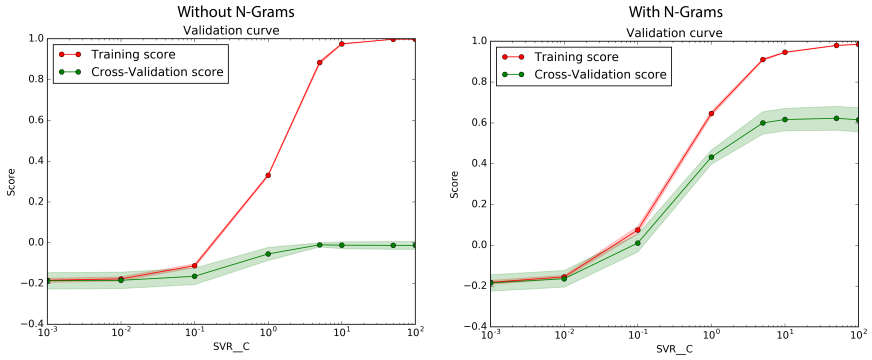

Abbildung 28: Validationskurve: (R^2)-Score nach C-Parameter des SVR-Modells mit RBF-Kernel. Links: ohne N-Grams. Rechts: mit N-Grams

die meisten Werte korrekt vorhergesagt. Auffällige Ausreisser gibt es nicht, aber scheinbar gibt es eine gewisse Tendenz zur Unterbewertung. So gibt es z. B. keine Datei mit >4 Bugs, welche als 0 Bugs gewertet wird. Hingegen gibt es aber einige Dateien mit 0 Bugs, die als >4 Bugs gewertet werden.

Prediction → Ground Truth ↓	0 Bugs	1 Bug	2-3 Bugs	>4 Bugs
0 Bugs	401	172	27	7
1 Bug	172	62	49	12
2-3 Bugs	20	14	37	31
>4 Bugs	0	3	15	52

Tabelle 22: Confusion-Matrix für Prediction auf dem Testset *ohne* Einsatz von N-Grams

Prediction → Ground Truth ↓	0 Bugs	1 Bug	2-3 Bugs	>4 Bugs
0 Bugs	387	21	29	1
1 Bug	193	80	34	39
2-3 Bugs	1	23	52	14
>4 Bugs	0	3	13	48

Tabelle 23: Confusion-Matrix für Prediction auf dem Testset *mit* Einsatz von N-Grams

Zusammenfassung Die Resultate zeigen, dass eindeutig ein statistischer Zusammenhang zwischen den von uns gesammelten Features und der Anzahl Bugs in den kommenden 6 Monaten besteht. Ob die N-Grams wirklich einen positiven Effekt haben, wird dadurch aber noch nicht klar beantwortet. Dazu müssten zuerst noch weitere Experimente mit grösseren Datensets durchgeführt werden.

Zu bemerken ist, dass sowohl mit als auch ohne N-Grams ein MDE von < 0.5 erreicht wird. Das bedeutet, dass bei mindestens der Hälfte aller Datensätze die gerundete Vorhersage korrekt ist.

4.3.3 Vergleich der Feature-Gruppen

Im Folgenden wurde versucht, die Effektivität der verschiedenen Feature-Gruppen, welche wir entwickelt haben, miteinander zu vergleichen. Dazu wurden sogenannte Ablation-Tests durchgeführt. Dabei wurden einzelne Feature-Gruppen deaktiviert und dann ein Modell mit ansonsten identischen Konfiguration trainiert. Die Unterschiede der Resultate, verglichen mit allen Features, kann Rückschlüsse auf den Effekt der weggelassenen Feature-Gruppe geben. Die Ergebnisse sind aber mit Zurückhaltung zu interpretieren. Da die Features auch untereinander interagieren und teilweise voneinander abhängig sind, bilden Ablation-Tests nicht die ganze Realität ab und können höchstens als Indikator dienen.

weggelassene Features	**Set**	**MAE**	**MDE**	**R^2**
Keine (Baseline)	Training	0.3084	0.1001	0.8439
	Test	0.8343	0.4732	0.7004
Lines-Of-Code	Training	0.0106	0.0000	−0.0094
	Test	−0.0124	−0.0013	−0.0041
Objektorientierte	Training	0.0106	0.0000	−0.0111
	Test	0.0008	−0.0036	−0.0112
Code-Complexity	Training	0.0174	0.0000	−0.0151
	Test	0.0348	0.0574	−0.0167
Anzahl-und-Typen	Training	0.0573	0.0000	−0.0564
	Test	0.0340	−0.0068	−0.0526
Temporale	Training	−0.1028	−0.0002	0.0944
	Test	−0.2508	−0.2944	0.1104
Textanalyse	Training	0.0219	0.0000	−0.0204
	Test	−0.0225	−0.0307	−0.0122

Tabelle 24: *Differenzen* zu den Metriken mit allen Features, wenn einzelne Feature-Gruppen weggelassen werden. Verwendetes ML-Modell: SVR mit RBF-Kernel, ohne N-Grams

Die Tabelle 24 zeigt die Ergebnisse dieser Ablation-Tests. Zu beachten ist, dass sie nicht die tatsächlichen Metriken anzeigt, sondern die Differenz zur jeweiligen Metrik wenn alle Features eingeschlossen werden (aber ohne N-Grams).

Die Effekte sind eher kleiner als wir erwartet haben. Ausserdem ist der Einfluss der meisten Feature-Gruppen ähnlich gross. Den grössten Einfluss gemäss diesen Resultaten hatten die Code-Complexity-Features. Sehr überraschend ist, dass das Weglassen der temporalen Features scheinbar einen positiven Effekt hat. Diesen Umstand konnten wir nicht erklären. Eigentlich wäre zu erwarten gewesen, dass ein “unnützes” Feature zu keiner Änderung im Ergebnis führt, dieses aber nicht verschlechtert. Die Gruppe der temporalen Features ist sehr gross ist, sie umfasst beinahe drei Mal so viele Features wie die restlichen Gruppen zusammen. Es wäre möglich, dass dieser Umstand einen Einfluss auf dieses Resultat hat. Da wir uns eigentlich viel von den temporalen Features erhofft haben, empfehlen wir etwaigen Nachfolgern dieses Phänomen noch genauer zu untersuchen. Es könnte sich auch herausstellen, dass es sich nur um eine Eigenheit der verwendeten Datensets handelt.

Als Ergänzung zu den Ablation-Tests sind in der Tabelle 25 Resultate mit

Aktivierte Features	Set	MAE	MDE	R^2
Lines-Of-Code	Training	0.5595	0.1001	0.5946
	Test	0.8475	0.1129	0.5064
Objektorientierte	Training	0.6189	0.1037	0.5653
	Test	0.8483	0.1765	0.4816
Code-Complexity	Training	0.5358	0.1002	0.6293
	Test	0.7394	0.1323	0.6808
Anzahl-und-Typen	Training	0.4412	0.1000	0.7097
	Test	0.6614	0.1710	0.7323
Temporale	Training	0.3133	0.1001	0.8171
	Test	1.0169	0.6163	0.4688
Textanalyse	Training	0.5669	0.1003	0.5274
	Test	0.8700	0.2482	0.4570
N-Grams	Training	0.1498	0.0999	0.9684
	Test	0.6301	0.1742	0.7626

Tabelle 25: Resultate, wenn Feature-Gruppen einzeln eingesetzt werden. Verwendetes ML-Modell: SVR mit RBF-Kernel, ohne N-Grams

jeweils nur einer aktivierten Feature-Gruppe aufgelistet. Erstaunlich ist, dass bei den gewählten Rahmenbedingungen der Tests die Anzahl-und-Typen-Features auf dem Testset besser abschneiden als auf dem Trainingsset. Dies können wir uns nicht erklären. Um dieses Phänomen zu analysieren und exaktere Aussagen darüber machen zu können, sind weiterführende Tests mit längeren Zeitspannen und unterschiedlichen Zeiträumen unabdingbar.

Überraschend ist auch die Leistung der N-Gram-Features. Diese zeigen die besten Werte auf dem Test- sowie auf dem Trainingsset. Auch interessant ist, dass die N-Gram-Features alleinstehend bessere Ergebnisse als in Kombination mit den anderen Features liefern. Eine mögliche Erklärung für dieses Verhalten könnte sein, dass im gewählten Trainings- und Testset möglicherweise mehrheitlich dieselben Files von Bugs betroffen sind, etwa weil sich diese dann gerade in Entwicklung befunden haben. Wir vermuten, dass N-Grams in diesem Fall gut darin sind, Files zu "identifizieren", da aufgrund der kleinen Datenmenge Kombinationen von grösseren N-Grams eher selten sind. Kommt dann ein seltenes N-Gram nur in einem Bug-behafteten File vor, so könnte dieses N-Gram stark gewichtet werden. Wenn im Testset dann dasselbe File wieder (oder immer noch) bugbehaftet ist und dieses N-Gram immer noch enthält, wird es dann korrekt mit einer hohen Bug-Anzahl gewertet. Das ist aber nur eine Vermutung, erst weitere Experimente können Aufschluss über dieses Phänomen geben.

Die temporalen Features scheinen auf den Trainingsset gut zu funktionieren, weisen jedoch auf dem Testset den höchsten MAE auf. Dies bedeutet, dass beim Vorhersagen auch die meisten Falschaussagen getroffen werden. Das schlechte Abschneiden der temporalen Features stimmt mit den schlechten Resultaten bei den Ablation-Tests in Tabelle 24 überein.

4.3.4 Log-Transform

Eine weitere Idee entstand aus der Überlegung, dass Bugs annähernd Poisson-verteilt sein müssten (siehe Kapitel 4.2.1). Um so verteilte Ereignisse vorherzu-

sagen, gibt es die sogenannte Poisson-Regression. Diese unterscheidet sich von Linearer Regression nur darin, dass der Target-Vektor log-transformiert wird:

$$\log(y) = \omega_0 + \omega_1 x_1 + \omega_2 x_2$$

Diese Transformation wurde kurzerhand in die ML-Pipeline eingebaut. Die Tabelle 26 zeigt, wie sich das Resultat mit aktivem Log-Transform verändert. Als Basis wurden dieselben Datensets wie bei den vorhergehenden Experimenten genutzt. Wie bei den Ablation-Tests wurde das SVR-Modell mit RBF-Kernel eingesetzt.

Log-Transform	**Set**	**MAE**	**MDE**	$\mathbf{R^2}$
Ohne	Training	0.3084	0.1001	0.8439
	Test	0.8343	0.4732	0.7004
Mit	Training	0.4361	0.1053	0.9774
	Test	0.7819	0.2880	0.6393

Tabelle 26: Einfluss eines Log-Transforms des Target-Vektors. Verwendetes ML-Modell: SVR mit rbf-Kernel, ohne N-Grams

Die Resultate zeigen, dass der R^2-Score auf etwas mehr Overfitting hindeutet. Dafür wird der MAE und MDE auf dem Testset tatsächlich besser. Wir denken, es ist lohnenswert diesen Umstand in Zukunft etwas genauer zu untersuchen.

5 Fazit

Diese Bachelorarbeit entwickelte sich zu einem umfassenden und spannenden Projekt. Im Folgenden soll mit einem Rückblick im Kapitel 5.1 das Getane reflektiert werden. Ausserdem bieten wir im Kapitel 5.2 einen umfassenden Ausblick auf die weiteren Möglichkeiten.

5.1 Diskussion

Der Grossteil dieser Arbeit befasste sich mit der Entwicklung eines Toolsets. Mit den entwickelten Tools GtSooG und Feature Extractor lassen sich Daten eines Softwareprojekts gezielt und strukturiert extrahieren, analysieren und verarbeiten. Die ML-Pipeline ermöglicht es, diese Daten zum Trainieren verschiedener Regressionsmodelle einzusetzen. Dies entspricht der Zielsetzung dieser Arbeit.

Diese Applikationen sind als Prototypen zu verstehen, sie erwiesen sich aber als robust und umfassen bereits eine breite Funktionspalette. Wir glauben, dass sie eine solide Grundlage für zukünftige Arbeiten bieten können. Aufgrund ihrer modularen Architektur sind sie insbesondere einfach zu erweitern. Damit erachten wir auch diesen Punkt unserer Zielsetzung als erfüllt.

Zuletzt versuchten wir, Konzepte aus der Textanalyse mit bestehenden Ansätzen der Fehlervorhersage zu kombinieren. Dies realisierten wir durch neue Features, insbesondere N-Grams. Die entsprechende Funktionalität wurde als Teil des Feature Extractors und der ML-Pipeline implementiert. Die begrenzte Anzahl von Experimenten, die uns zeitlich möglich waren, reichte leider nicht, um fundierte Aussagen über die Effektivität dieser Features zu treffen. Der tiefe MDE beim Versuch mit SVR mit RBF (siehe Tabelle 19) könnte aber ein Indiz auf einen positiven Effekt dieser Features sein.

Allgemein waren aus zeitlichen Gründen nicht genügend Experimente möglich, um die Leistung unserer Lösung abschliessend beurteilen zu können. Trotzdem liessen sich signifikante Zusammenhänge zwischen den von uns entwickelten Features und der Fehleranfälligkeit von Java-Dateien zeigen. Diese Ergebnisse haben unsere anfänglichen Erwartungen sogar übertroffen. Im Kapitel 5.2.5 schlagen wir eine Reihe von weiterführenden Experimenten vor, mit welchen die Qualität unserer Lösung weiter analysiert werden kann.

Wir sind der Meinung, dass sich eine Weiterführung dieser Arbeit lohnt. Unsere Recherchen zum Thema haben gezeigt, dass Fehlervorhersage mittels Machine Learning viel Potenzial aufweist. Die Resultate, welche wir im Zuge unserer Arbeit erhalten haben, unterstützen dies. Den Nutzen eines Systems zur Fehlervorhersage bewerten wir als sehr hoch. Wir nehmen an, dass insbesondere grosse Projekte, welche sich in reger Entwicklung befinden davon profitieren könnten.

5.2 Ausblick

Diese Arbeit legt den Grundstein für zukünftige Projekte im Bereich Fehlervorhersage mit Machine Learning. Mit den entwickelten Tools sind erste Gehversuche mit realen Java Projekten möglich. Bis zu einer schlüsselfertigen Lösung ist es allerdings noch ein weiter Weg. Ein Ausblick soll über die Weiterentwicklungsmöglichkeiten und das Potential der Arbeit Bescheid geben.

5.2.1 GtSooG

GtSooG wurde zwar mit einigen Repositories getestet, allerdings ist deren Anzahl zu gering, um eine verlässliche Aussage über die Qualität des Tools zu treffen. Die mangelnde Funktionalität von *GitPython* verkomplizierte den Programmablauf und somit auch die Zuverlässigkeit von GtSooG. Die im Kapitel 3.2.4 beschriebenen Erkenntnisse legen die Verwendung einer anderen Bibliothek für den Umgang mit *Git* nahe. Für den in Java geschrieben Feature Extractor kam *Eclipse JGit* zum Einsatz, was einen besseren Eindruck machte. Die Portierung auf Java mit *JGit* könnte die Zuverlässigkeit von GtSooG verbessern.

Während der Entwicklung stellte sich ein Multithreading-Design von GtSooG als schwierig heraus (siehe Kapitel 3.2.3). Als Ursache ist unter anderem das Design der *GitPython* Bibliothek auszumachen. *JGit* könnte hier ebenfalls eine bessere Grundlage bieten.

Der Ansatz, Source Code direkt von einem *Git*-Repository auszulesen ist für den Zweck dieser Arbeit optimal. Mit *Boa* [37] ist allerdings bereits eine mächtige Sprache für Aufgaben im Bereich Repository Mining vorhanden. Anstatt dass GtSooG den Source Code von einem *Git*-Repository einliest, wäre die Integration von *Boa* zu erwägen. Damit eröffnet sich die Möglichkeit auf eine Vielzahl von Projekten mit unterschiedlichen Versionsverwaltungen zuzugreifen.

5.2.2 Feature Extractor

Der Feature Extractor liest pro Commit nur die im Moment behandelte Version ein. Dies führt dazu, dass der gesamte restliche Projekt-Source-Code zum Commit-Zeitpunkt *nicht* zur Verfügung steht. Einige Features, welche die Abhängigkeit zwischen Projektklassen abbilden sollen (siehe Kapitel 3.4.2), können aufgrund dessen nicht berechnet werden. Um diese Einschränkung aufzuheben gälte es einen performanten Weg zu finden, den gesamten Projekt-Code einzulesen.

Das Tool benötigt, wie auch GtSooG, direkten Zugriff auf ein *Git*-Repository. Das Einbinden von *Boa* könnte den Feature Extractor von lokalen *Git*-Repositories entkoppeln und vielleicht auch hilfreich bei der oben erwähnten Einschränkung sein.

Der Feature Extractor berechnet bei jedem Durchgang alle Features für alle Dateiversionen neu, unabhängig davon, ob diese bereits in der Datenbank gespeichert sind oder nicht. Bei sehr grossen Repositories kann dies eine Weile dauern. In Zukunft sollten nur die Features zu den neuen Versionen berechnet werden. Dies wäre insbesondere beim Einsatz in einem realen Softwareprojekt sinnvoll, bei dem ständig neue Versionen dazukommen.

5.2.3 ML-Pipeline

Die ML-Pipeline kann trainierte Modelle nicht speichern. Für einige Experimente wäre es interessant, über einen grossen Zeitraum lernen zu lassen und dann die Testresultate verschiedener Zeiträume zu vergleichen. Beispielsweise kann man das Modell über die Daten des Zeitraumes von 2011 - 2014 trainieren. Als Testzeiträume könnten dann die einzelnen Monate des Jahres 2015 verwendet werden. Dies würde zeigen, wie sich die Qualität der Vorhersagen bei zunehmendem zeitlichem Abstand zum Trainingszeitraum verändert.

Die Reihenfolge der von der ML-Pipeline eingelesen Features ist von der Datenbank abhängig. Da in der Datenbank ein Index auf die Feature-ID gesetzt ist, führt dies dazu, dass die Features in alphabetischer Reihenfolge eingelesen werden. In der Konfiguration der ML-Pipeline lassen sich einzelne Features aktivieren oder deaktivieren. Die Reihenfolge der Features muss dabei zwingend mit der Reihenfolge in der Datenbank übereinstimmen (alphabetisch geordnet). Ist das nicht der Fall, wird Feature-Typ-A mit Feature-Typ-B verglichen und es entstehen unbrauchbare Resultate. Dasselbe gilt für Features, welche für Version-A vorhanden sind, jedoch nicht für Version-B. Dies kann den gesamten Feature-Vektor dieser Version verschieben, was das Resultat verfälscht. Allgemein werden fehlende Features bisher nicht ausreichend behandelt. Eine finale Version der ML-Pipeline sollte im Umgang mit Features robuster implementiert werden.

Eine weitere Verbesserung wäre die Möglichkeit, einzelne Dateien auszuschliessen. Im Moment arbeitet die ML-Pipeline mit den Dateiversionen aller Java-Dateien. Vielleicht würde es sich als sinnvoll erweisen, gewisse Dateien, wie etwa Tests, ausschliessen zu können. Dies würde auch interessante Experimente ermöglichen, welche den Unterschied zwischen Java-Source- und Java-Test-Code vergleichen.

5.2.4 Machine Learning

Auch im Bereich ML gibt es einige Ansätze, welche wir als erprobenswert erachten, aber aus zeitlichen Gründen nicht mehr umsetzen konnten. Die ML-Pipeline müsste um diese noch erweitert werden.

Um die Laufzeit für das Training und die Prediction zu verkürzen, könnte Feature Selection eingesetzt werden. Das ist eine Sammlung von Verfahren, die darauf abzielen, redundante oder irrelevante Features zu erkennen. Damit soll die Anzahl Features, welche in einem ML-Modell eingesetzt werden, verringert werden, ohne dass dabei viel Information verloren geht. So wird das Modell vereinfacht, was zu einer geringeren Laufzeit führt. Ein weiterer positiver Effekt, den Feature Selection haben kann, ist die Reduzierung von Overfitting indem die Varianz der Features verringert wird.

scikit-learn bietet verschiedene Werkzeuge für Feature Selection [116]. Zum Beispiel können Features mit tiefer Varianz eliminiert werden oder anhand des Gewichts, das sie in einem Modell erhalten. Auch Principal Component Analysis (PCA) [117] ist ein nützliches Verfahren um die Dimension eines Problems zu verringern. Alle Verfahren können auch als Teil einer Pipeline eingesetzt werden, was eine Erweiterung der ML-Pipeline vereinfacht.

Neben Feature Selection empfehlen wir auch den Einsatz eines Random Forests. Diese Art Meta-Modell trainiert eine Menge von Decision Trees auf ein Datenset und nutzt danach den Mittelwert über diese Trees um die Genauigkeit zu erhöhen und Overfitting zu vermeiden. Oft werden Random Forests für Klassifikationsprobleme eingesetzt, *scikit-learn* bietet aber auch eine Variante für Regression an [118]. Dieses Verfahren könnte sich als nützlich erweisen, da es sehr robust ist.

5.2.5 Experimente

Aufgrund des Umfangs der Experimente, welche in dieser Arbeit möglich waren, lassen sich noch keine endgültigen Schlüsse auf die Leistung der entwickelten

Features und Tools ziehen. In zukünftigen, darauf aufbauenden Projekten sollte das Experimentieren mit einem genügend grossen Trainingsset und verschiedenen Zeitspannen eine integrale Aufgabe darstellen.

Eine mögliche Auswahl an geeigneten Projekten wurde bereits in dieser Arbeit erstellt und in Tabelle 27 aufgelistet.

Name	**Commits**	**GitHub URL**
Liferay Portal	$\sim 180'000$	`https://github.com/liferay/liferay-portal`
Gradle	$\sim 36'000$	`https://github.com/gradle/gradle`
Elasticsearch	$\sim 22'000$	`https://github.com/elastic/elasticsearch`
Spring Framework	$\sim 12'000$	`https://github.com/spring-projects/spring-framework`
Guava	~ 3500	`https://github.com/google/guava`
JUnit 4	~ 2100	`https://github.com/junit-team/junit4`

Tabelle 27: Mögliche Projekte für weitere Experimente

Diese Projekte beinhalten alle mehrheitlich Java Code und verfügen über ein geeignetes Issue-Tracking-System (*JIRA* oder *GitHub*). Auch sind sie unterschiedlich gross, sodass auch das Verhalten bei kleiner Datenmenge untersucht werden kann.

Die Grösse der in dieser Arbeit untersuchten Zeiträume (siehe Kapitel 3.6) wurde so gewählt, dass sie auf der zur Verfügung stehenden Infrastruktur (siehe Kapitel 3.1.3) jeweils in praktikabler Zeit terminierten. Interessant wäre auf jeden Fall das Verhalten bei sehr grossen Trainingssets und verschiedenen Testsets zu untersuchen. Für Trainingssets werden von uns folgende Zeitspannen als prüfenswert erachtet:

- 3 Monate
- 6 Monate
- 1 Jahr
- 2 Jahre
- 3 Jahre
- Gesamte Projektlebensdauer

Für das Testset sollte die Dauer keine grosse Rolle spielen, solange es mindestens 100-200 Dateiversionen enthält.

Neben der Dauer ist auch der jeweilige Zeitpunkt der Datensets interessant. Folgt das Testset direkt auf das Trainingsset, erwarten wir andere Resultate, als wenn es zwischen Trainings- und Testset einen zeitlichen Abstand gibt. Es ist auch denkbar, dass aufgrund von unregelmässigen Entwicklungszyklen der Zeitpunkt (Sommer, Herbst, Winter, Frühling) einen messbaren Unterschied nach

sich zieht. Die Tabelle 28 zeigt solche Beispielzeitspannen. Dabei ist aber anzunehmen, dass die Resultate solcher Experimente stark abhängig vom jeweiligen Projekt sein werden.

Beschreibung	**Zeitspanne Trainingsset**	**Zeitspanne(n) Testset**
Trainings- und Testset mit grossem Abstand	01.01.2009 bis 31.12.2009	01. - 31.01.2015
Erstes Quartal	01.01.2009 bis 31.03.2009	01. - 31.06.2010
Zweites Quartal	01.04.2009 bis 30.06.2009	01. - 31.06.2010
Drittes Quartal	01.07.2009 bis 30.09.2009	01. - 31.06.2010
Viertes Quartal	01.10.2009 bis 31.12.2009	01. - 31.06.2010
Mehrere Testset mit identischem Trainingsset	01.01.2013 bis 31.12.2013	01. -31.01.2014 01. -31.02.2014 ... 01. - 31.12.2014

Tabelle 28: Beispielzeitspannen

Ein anderer interessanter Ansatz könnte die Anwendung einer kurzen Lernzeitspanne und nachfolgender Testzeitspanne sein, mit denen als Sliding Window über die ganze Projektlaufzeit getestet wird. Die Dauer des Testsets wird z. B. auf einen Monat festgelegt. Das Trainingsset trainiert ab dem Zeitpunkt der Projektgeburt. Anschliessend werden die Ergebnisse mit dem Testset validiert. Nun verschieben sich beide Zeitfenster um einen Monat in Richtung Gegenwart und das Ganze wird wiederholt. Mit dieser Methode könnte eine Aussage darüber gemacht werden, ab welchem Projektalter die Fehlervorhersage gut funktioniert. Abbildung 29 visualisiert das Sliding-Window-Verfahren.

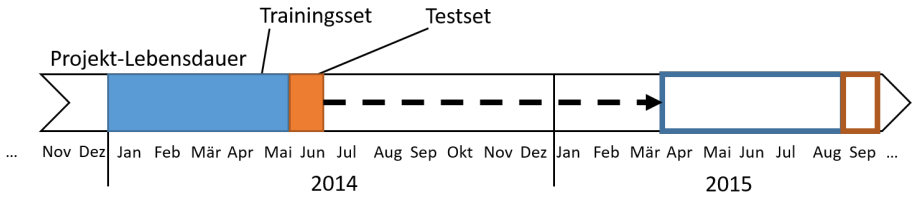

Abbildung 29: Das Sliding Window, bestehend aus Trainings- und Testset, wird über die Projekt-Lebensdauer geschoben.

5.2.6 Vision einer einsetzbaren Lösung

Falls sich die Methoden dieser Arbeit in Zukunft zur Fehlervorhersage mit realen Projekten als tauglich erweisen, wäre eine denkbare Anwendung das Bestimmen einer “Heatmap” für Softwareprojekte. Eine solche würde anzeigen, welche

Komponenten besonders fehleranfällig sind, was sowohl Entwicklern als auch Projektleitern helfen könnte.

Ein langfristiges Ziel könnte neben der Fehlervorsage auch die Fehlerlokalisierung sein. Dazu müsste allerdings die Granularität der Analyse verfeinert werden. Im Moment ist die kleinste Einheit eine Source-Code-Datei, d. h. in der Regel genau eine Java-Klasse. Für eine brauchbare Fehlerbestimmung müsste auf Methodenebene gearbeitet werden.

Eine produktiv eingesetzte Lösung sollte idealerweise auch weitere Programmiersprachen unterstützen. Die Integration der Sprache *Boa*, welche bereits Unterstützung für verschiedene Programmiersprachen bietet, könnte bei diesem Unterfangen sehr hilfreich sein. Damit die unterschiedlichen Programmiersprachen identisch verarbeitet werden können, wäre eine Zwischensprache notwendig. Neben der Definition dieser Zwischensprache ist auch ein Tool zur Übersetzung des Source Codes in einen AST notwendig.

Damit die in unserer Arbeit entwickelten Tools ihrer Arbeit nachgehen können, ist eine Menge an Konfiguration nötig. Zudem müssen sie der Reihe nach einzeln gestartet werden. Um später ein schlüsselfertiges Tool zur Verfügung zu stellen, sollten diese Einschränkungen aufgehoben werden.

Eine Idee ist, den Dienst direkt aus der Cloud als Service anzubieten, wie es zum Beispiel Code Climate [11] tut. Die Software sollte per Self-Service an die vom Programmierer verwendete Versionsverwaltung angebunden werden. Die Fehlervorsage könnte per Webhooks [119] sogar automatisch bei jedem Commit ausgelöst werden. Damit könnte der Qualitätsstand des Projektes quasi “live” mitverfolgt werden.

Natürlich ist das nur ein Beispiel von vielen Möglichkeiten, eine ML-gestützte Fehlervorhersage für Source Code einzusetzen. Auf jeden Fall ist es noch ein weiter Weg, bis eine produktiv nutzbare Lösung realisierbar ist. Wir glauben aber, dass es sich lohnt, diesen zu beschreiten.

6 Verzeichnisse

6.1 Literatur

[1] GitHub. (Mai 2016). GitHub Guides, Adresse: https://guides.github.com/.

[2] Git. (Mai 2016). Git documentation, Adresse: https://git-scm.com/doc.

[3] A. Ng. (Mai 2016). Machine Learning - Stanford University — Coursera, Adresse: https://www.coursera.org/learn/machine-learning.

[4] V. Developers. (Mai 2016). Valgrind Home, Adresse: http://valgrind.org/.

[5] S. Christou. (Mai 2016). Cobertura - A code coverage utility for Java., Adresse: http://cobertura.github.io/cobertura/.

[6] lcamtuf. (Mai 2016). American fuzzy lop, Adresse: http://lcamtuf.coredump.cx/afl/.

[7] Findbugs. (Mai 2016). FindBugs^{tm} - Find Bugs in Java Programs, Adresse: http://findbugs.sourceforge.net/.

[8] R. Ivanov. (Mai 2016). Checkstyle, Adresse: http://checkstyle.sourceforge.net/.

[9] A. Dangel. (Mai 2016). PMD - Don't shoot the messenger, Adresse: http://pmd.github.io/.

[10] Oracle. (Mai 2016). Man page lint, Adresse: http://docs.oracle.com/cd/E19205-01/820-4180/man1/lint.1.html.

[11] CodeClimate. (Mai 2016). Code Climate - Static analysis from your command line to the cloud, Adresse: https://codeclimate.com/.

[12] C. Lewis und R. Ou. (Mai 2016). Google Engineering Tools - Bug Prediction at Google, Adresse: http://promise.site.uottawa.ca/SERepository/datasets-page.html.

[13] F. Rahman, D. Posnett, A. Hindle, E. Barr und P. Devanbu, „BugCache for inspections: hit or miss?“, in *PROCEEDINGS OF THE 19TH ACM SIGSOFT SYMPOSIUM AND THE 13TH EUROPEAN CONFERENCE ON FOUNDATIONS OF SOFTWARE ENGINEERING*, ACM, 2011, S. 322–331.

[14] N. Nagappan und T. Ball, „Use of Relative Code Churn Measures to Predict System Defect Density“, in *PROCEEDINGS OF THE 27TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*, Ser. ICSE '05, St. Louis, MO, USA: ACM, 2005, S. 284–292, ISBN: 1-58113-963-2. DOI: 10.1145/1062455.1062514. Adresse: http://doi.acm.org/10.1145/1062455.1062514.

[15] E. Giger, M. Pinzger und H. C. Gall, „Comparing Fine-grained Source Code Changes and Code Churn for Bug Prediction“, in *PROCEEDINGS OF THE 8TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES*, Ser. MSR '11, Waikiki Honolulu HI USA: ACM, 2011, S. 83–92, ISBN: 978-1-4503-0574-7. DOI: 10.1145/1985441.1985456. Adresse: http://doi.acm.org/10.1145/1985441.1985456.

[16] V. Barstad, M. Goodwin und T. Gjøsæter, „Predicting Source Code Quality with Static Analysis and Machine Learning“, in *NORSK INFORMATIKKONFERANSE (NIK 2014)*, 2014. Adresse: ojs.bibsys.no/index.php/NIK/article/download/26/22.

[17] P. N. Juristo. (Sep. 2015). Promise - The 12th International Conference on Predictive Models and Data Analytics in Software Engineering, Adresse: http://promisedata.org/2016/index.html#.

[18] U. of Ottawa. (Mai 2016). Promise Software Engineering Repository - Public Datasets, Adresse: http://promise.site.uottawa.ca/SERepository/datasets-page.html.

[19] McCabe, „A Complexity Measure, journal = IEEE Transactions on Software Engineering“, Bd. 2, S. 308–320, 1976.

[20] M. H. Halstead, *ELEMENTS OF SOFTWARE SCIENCE (OPERATING AND PROGRAMMING SYSTEMS SERIES)*. New York, NY, USA: Elsevier Science Inc., 1977, ISBN: 0444002057.

[21] A. R. Sharafat und L. Tahvildari, „Change Prediction in Object-Oriented Software Systems: A Probabilistic Approach“, *JSW*, Bd. 3, Nr. 5, Mai 2008. DOI: 10.4304/jsw.3.5.26-39. Adresse: http://www.academypublisher.com/jsw/vol03/no05/jsw03052639.pdf.

[22] F. Rahman, S. Khatri, E. T. Barr und P. Devanbu, „Comparing static bug finders and statistical prediction“, in *PROCEEDINGS OF THE 36TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - ICSE 2014*, Association for Computing Machinery (ACM), 2014. DOI: 10.1145/2568225.2568269. Adresse: http://dx.doi.org/10.1145/2568225.2568269.

[23] ——, „Comparing static bug finders and statistical prediction“, in *PROCEEDINGS OF THE 36TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - ICSE 2014*, Association for Computing Machinery (ACM), 2014, S. 10. DOI: 10.1145/2568225.2568269. Adresse: http://dx.doi.org/10.1145/2568225.2568269.

[24] E. Arisholm, L. C. Briand und E. B. Johannessen, „A systematic and comprehensive investigation of methods to build and evaluate fault prediction models“, *Journal of Systems and Software*, Bd. 83, Nr. 1, S. 2–17, Jan. 2010. DOI: 10.1016/j.jss.2009.06.055. Adresse: http://dx.doi.org/10.1016/j.jss.2009.06.055.

[25] M. D'Ambros, M. Lanza und R. Robbes, „Evaluating defect prediction approaches: a benchmark and an extensive comparison“, *Empirical Software Engineering*, Bd. 17, Nr. 4-5, S. 531–577, Aug. 2011. DOI: 10.1007/s10664-011-9173-9. Adresse: http://dx.doi.org/10.1007/s10664-011-9173-9.

[26] S. Lessmann, B. Baesens, C. Mues und S. Pietsch, „Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings“, *IEEE Transactions on Software Engineering*, Bd. 34, Nr. 4, S. 485–496, Juli 2008. DOI: 10.1109/tse.2008.35. Adresse: http://dx.doi.org/10.1109/TSE.2008.35.

[27] M. Liljeson und A. Mohlin, *Software defect prediction using machine learning on test and source code metrics*, 2014.

[28] A. Schröter, T. Zimmermann und A. Zeller, „Predicting Component Failures at Design Time“, in *PROCEEDINGS OF THE 2006 ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING*, Ser. ISESE '06, Rio de Janeiro, Brazil: ACM, 2006, S. 18–27, ISBN: 1-59593-218-6. DOI: 10.1145/1159733.1159739. Adresse: http://doi.acm.org/10.1145/1159733.1159739.

[29] D. Chollak, „Software Bug Detection Using the N-gram Language Model“, master's thesis, University of Waterloo, 2015.

[30] S. Nessa, M. Abedin, W. E. Wong, L. Khan und Y. Qi, „Software Fault Localization Using N-gram Analysis“, in *Wireless Algorithms, Systems, and Applications*, Springer Science and Business Media, S. 548–559. DOI: 10.1007/978-3-540-88582-5_51. Adresse: http://dx.doi.org/10.1007/978-3-540-88582-5_51.

[31] J. Choi, H. Kim, C. Choi und P. Kim, „Efficient Malicious Code Detection Using N-Gram Analysis and SVM“, in *2011 14th International Conference on Network-Based Information Systems*, Institute of Electrical und Electronics Engineers (IEEE), Sep. 2011. DOI: 10.1109/nbis.2011.104. Adresse: http://dx.doi.org/10.1109/NBiS.2011.104.

[32] B. Zhang, J. Yin, J. Hao, S. Wang und D. Zhang, „New Malicious Code Detection Based on N-Gram Analysis and Rough Set Theory“, in *Computational Intelligence and Security*, Springer Science und Business Media, 2007, S. 626–633. DOI: 10.1007/978-3-540-74377-4_65. Adresse: http://dx.doi.org/10.1007/978-3-540-74377-4_65.

[33] M. Kim. (Mai 2016). MSR 2016, Adresse: http://2016.msrconf.org/.

[34] GitHub. (Apr. 2016). GitHub, Adresse: https://github.com/.

[35] SourceForge. (Apr. 2016). SourceForge, Adresse: https://sourceforge.net/.

[36] R. Dyer, H. A. Nguyen, H. Rajan und T. N. Nguyen, „Boa: Ultra-Large-Scale Software Repository and Source-Code Mining“, *ACM Trans. Softw. Eng. Methodol.*, Bd. 25, Nr. 1, 7:1–7:34, Dez. 2015, ISSN: 1049-331X. DOI: 10.1145/2803171. Adresse: http://doi.acm.org/10.1145/2803171.

[37] I. S. University. (Mai 2016). Boa - Iowa State University, Adresse: http://boa.cs.iastate.edu/.

[38] M. B. Dwyer. (Mai 2016). Software-artifact Infrastructure Repository, Adresse: http://sir.unl.edu/portal/index.php.

[39] T. M. Mitchell, *Machine Learning*. McGraw-Hill Education, 1997, ISBN: 0070428077. Adresse: http://www.amazon.com/Machine-Learning-Tom-M-Mitchell/dp/0070428077%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0070428077.

[40] U. Zürich. (Mai 2016). Methodenberatung Universität Zürich: Einfache lineare Regression, Adresse: http://www.methodenberatung.uzh.ch/datenanalyse/zusammenhaenge/ereg.html.

[41] S. Turner. (Okt. 2013). Net Prophet: Local Regression, Adresse: http://netprophetblog.blogspot.ch/2013/10/local-regression.html.

[42] scikit-learn developers. (Mai 2016). Scikit-Learn User Guide: Generalized Linear Models, Adresse: http://scikit-learn.org/stable/modules/linear_model.html.

[43] H. J. Oberle. (Mai 2016). Uni Hamburg: Das Gradientenverfahren, Adresse: http://www.math.uni-hamburg.de/home/oberle/skripte/optimierung/optim.pdf.

[44] H. Lohninger. (Mai 2016). Grundlagen der Statistik: MLR und (Multi)Kollinearität, Adresse: http://www.statistics4u.com/fundstat_germ/cc_mlr_collinvars.html.

[45] scikit-learn developers. (Mai 2016). Underfitting vs. Overfitting, Adresse: http://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html.

[46] H. Lohninger. (Mai 2016). Grundlagen der Statistik: Ridge Regression, Adresse: http://www.statistics4u.com/fundstat_germ/ee_ridge_regression.html.

[47] opencv dev team. (Mai 2016). OpenCV: Introduction to Support Vector Machines, Adresse: http://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html.

[48] E. Kim. (Sep. 2013). Eric Kim: Everything You Wanted to Know about the Kernel Trick, Adresse: http://www.eric-kim.net/eric-kim-net/posts/1/kernel_trick.html.

[49] M. I. Jordan und R. Thibaux. (Mai 2016). "THE KERNEL TRICK."LECTURE NOTES, Adresse: http://www.cs.berkeley.edu/~jordan/courses/281B-spring04/lectures/lec3.pdf.

[50] scikit-learn developers. (Mai 2016). Scikit-Learn User Guide: Support Vector Machines, Adresse: http://scikit-learn.org/stable/modules/svm.html.

[51] ——, (Mai 2016). Scikit-Learn: Grid Search: Searching for estimator parameters, Adresse: http://scikit-learn.org/stable/modules/grid_search.html.

[52] ——, (Mai 2016). Scikit-Learn: Cross-validation: evaluating estimator performance, Adresse: http://scikit-learn.org/stable/modules/cross_validation.html.

[53] J. Cohen, *STATISTICAL POWER ANALYSIS FOR THE BEHAVIORAL SCIENCES (2ND EDITION)*. Routledge, 1988, ISBN: 0805802835. Adresse: http://www.amazon.com/Statistical-Analysis-Behavioral-Sciences-Edition/dp/0805802835%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0805802835.

[54] S. M. Mohammad, S. Kiritchenko und X. Zhu, „NRC-Canada: Building the State-of-the-Art in Sentiment Analysis of Tweets“, *CoRR*, Bd. abs/1308.6242, 2013. Adresse: http://arxiv.org/abs/1308.6242.

[55] Git. (Apr. 2016). Git, Adresse: http://git-scm.com.

[56] ——, (Apr. 2016). Git, Adresse: https://subversion.apache.org.

[57] F. S. F. Inc. (Apr. 2016). Concurrent Versions System, Adresse: http://savannah.nongnu.org/projects/cvs.

[58] M. Foundation. (Apr. 2016). Bugzilla, Adresse: https://www.bugzilla.org/.

[59] Atlassian. (Apr. 2016). Jira Software, Adresse: https://www.atlassian.com/software/jira.

[60] M. Corp. (Apr. 2016). CodePlex - Hosting for Open Source Software, Adresse: https://www.codeplex.com/.

[61] Atlassian. (Apr. 2016). Atlassian Bitbucket, Adresse: https://bitbucket.org/.

[62] C. Ltd. (Juni 2016). Bazaar, Adresse: http://bazaar.canonical.com/en/.

[63] Mercurial. (Juni 2016). Mercurial SCM, Adresse: https://www.mercurial-scm.org/.

[64] S. Overflow. (Apr. 2016). Stack Overflow Developer Survey 2015, Adresse: http://stackoverflow.com/research/developer-survey-2015#tech-sourcecontrol.

[65] N. Willis. (Aug. 2013). LWN.net: SourceForge offering ßide-loadingïnstallers, Adresse: https://lwn.net/Articles/564250/.

[66] S. Gallagher. (Mai 2015). arstechnica: SourceForge grabs GIMP for Windows' account, wraps installer in bundle-pushing adware, Adresse: http://arstechnica.com/information-technology/2015/05/sourceforge-grabs-gimp-for-windows-account-wraps-installer-in-bundle-pushing-adware/.

[67] I. S. University. (Mai 2016). Dataset Statistics - Boa - Iowa State University, Adresse: http://boa.cs.iastate.edu/stats/index.php.

[68] GitHub. (Apr. 2016). GitHub Developer: API Overview, Adresse: https://developer.github.com/v3/.

[69] Atlassian. (Apr. 2016). JIRA REST API Reference, Adresse: https://docs.atlassian.com/jira/REST/latest/.

[70] Git. (Mai 2016). Git Internals - Git Objects, Adresse: https://git-scm.com/book/en/v1/Git-Internals-Git-Objects.

[71] GitHub. (Mai 2016). GitHub API Ratelimit, Adresse: https://developer.github.com/v3/rate_limit/.

[72] K. Reitz. (Mai 2016). Requests: HTTP for Humans, Adresse: http://docs.python-requests.org/en/master/.

[73] S. Thiel. (Mai 2016). GitHub: GitPython, Adresse: https://github.com/gitpython-developers/GitPython.

[74] M. Bayer. (Mai 2016). SQLAlchemy: The Python SQL Toolkit and Object Relational Mapper, Adresse: http://www.sqlalchemy.org/.

[75] S. Thiel. (Mai 2016). Gitpython Async Projekt, Adresse: https://github.com/gitpython-developers/async.

[76] E. Foundation. (Mai 2016). Eclipse - jGit, Adresse: https://eclipse.org/jgit/.

[77] ——, (Mai 2016). Eclipse JDT AST, Adresse: http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FAST.html.

[78] JavaParser. (Mai 2016). javaparser, Adresse: https://github.com/javaparser/javaparser.

[79] I. Moriggl, „Intelligent Code Inspection using Static Code Features : An approach for Java“, Magisterarb., Blekinge Institute of Technology, School of Computing, 2010, S. 54.

[80] N. Nagappan und T. Ball, „Use of Relative Code Churn Measures to Predict System Defect Density“, in *PROCEEDINGS OF THE 27TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*, Ser. ICSE '05, St. Louis, MO, USA: ACM, 2005, S. 284–292, ISBN: 1-58113-963-2. DOI: 10.1145/1062455.1062514. Adresse: http://doi.acm.org/10.1145/1062455.1062514.

[81] T. Gyimothy, R. Ferenc und I. Siket, „Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction“, *IEEE Trans. Softw. Eng.*, Bd. 31, Nr. 10, S. 897–910, Okt. 2005, ISSN: 0098-5589. DOI: 10.1109/TSE.2005.112. Adresse: http://dx.doi.org/10.1109/TSE.2005.112.

[82] I. Siket, „Evaluating the Effectiveness of Object-Oriented Metrics for Bug Prediction“, in *PROCEEDINGS OF THE SIXTH CONFERENCE OF PHD STUDENTS IN COMPUTER SCIENCE (CSCS'08)*, 2008, S. 177–186.

[83] F. Dinari, „Halstead Complexity Metrics in Software Engineering“, *Journal of Renewable Natural Resources Bhutan*, Bd. 3, S. 418–424, Juli 2015.

[84] gboissier. (Mai 2016). Eclipse Metrics plugin, Adresse: https://sourceforge.net/projects/metrics2/.

[85] D. Chollak, „Software Bug Detection Using the N-gram Language Model“, Magisterarb., University of Waterloo, 2015. Adresse: http://hdl.handle.net/10012/9250.

[86] N. Nagappan, „A Software Testing and Reliability Early Warning (STREW) Metric Suite“, Diss., NC State University, Raleigh, NC 27695-7103, Feb. 2005.

[87] N. Nagappan, L. Williams, M. Vouk und J. Osborne, „Using In-Process Testing Metrics to Estimate Post-Release Field Quality“, in *THE 18TH IEEE INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY (ISSRE '07)*, Nov. 2007, S. 209–214. DOI: 10.1109/ISSRE.2007.18.

[88] scikit-learn developers. (Mai 2016). scikit-learn: Machine Learning in Python, Adresse: http://scikit-learn.org/stable/.

[89] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot und E. Duchesnay, „Scikit-learn: Machine Learning in Python“, *Journal of Machine Learning Research*, Bd. 12, S. 2825–2830, 2011.

[90] T. A. S. Foundation. (Mai 2016). Spark MLlib, Adresse: http://spark.apache.org/mllib/.

[91] C.-C. Chang und C.-J. Lin. (Mai 2016). LIBLINEAR – A Library for Large Linear Classification, Adresse: http://www.csie.ntu.edu.tw/~cjlin/libsvm/.

[92] C.-J. Lin. (Mai 2016). LIBLINEAR – A Library for Large Linear Classification, Adresse: http://www.csie.ntu.edu.tw/~cjlin/liblinear/.

[93] scikit-learn developers. (Mai 2016). Scikit learn: sklearn.svm.SVC, Adresse: http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[94] M. Corp. (Mai 2016). Microsoft Azure: Machine Learning, Adresse: https://azure.microsoft.com/en-us/services/machine-learning/.

[95] ——, (Mai 2016). Amazon Machine Learning: Produktdetails, Adresse: http://aws.amazon.com/de/machine-learning/details/.

[96] D. Learning. (Mai 2016). Pylearn2 Vision, Adresse: http://deeplearning.net/software/pylearn2/.

[97] ——, (Mai 2016). Theano, Adresse: http://deeplearning.net/software/theano/.

[98] D. E. Marek Arnold, „Developing PlebML: A modular machine learning framework“, English, type, ZHAW Institut für angewandte Informationstechnologie, Mai 2015.

[99] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta und A. G. Gray, „mlpack: A Scalable C++ Machine Learning Library“, *Journal of Machine Learning Research*, Bd. 14, S. 801–805, 2013.

[100] N. developers. (Mai 2016). NumPy, Adresse: http://www.numpy.org/.

[101] S. community. (Mai 2016). SciPy Sparse matrices (scipy.sparse), Adresse: http://docs.scipy.org/doc/scipy/reference/sparse.html.

[102] M. Rodrigues. (Mai 2016). GitHub - PyMySQL/PyMySQL Pure Python MySQL Client, Adresse: https://github.com/PyMySQL/PyMySQL.

[103] J. Hunter, D. Dale, E. Firing und M. Droettboom. (Mai 2016). matplotlib: python plotting, Adresse: http://matplotlib.org/.

[104] Robpol86. (Mai 2016). GitHub - Robpol86/terminaltables: Generate simple tables in terminals from a nested list of strings., Adresse: https://github.com/Robpol86/terminaltables.

[105] scikit-learn developers. (Mai 2016). SciKit learn User Guide - Pipeline: chaining estimators, Adresse: http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html#sklearn.pipeline.Pipeline.

[106] S. V. Stehman, „Selecting and interpreting measures of thematic classification accuracy“, *Remote Sensing of Environment*, Bd. 62, Nr. 1, S. 77–89, Okt. 1997. DOI: 10.1016/s0034-4257(97)00083-7. Adresse: http://dx.doi.org/10.1016/S0034-4257(97)00083-7.

[107] Elastic. (Apr. 2016). Elastic: Elasticsearch, Adresse: https://www.elastic.co/products/elasticsearch.

[108] ——, (Apr. 2016). GitHub: Elasticsearch, Adresse: https://github.com/elastic/elasticsearch.

[109] ——, (Apr. 2016). elasticsearch. blog: You Know, for Search (Archived), Adresse: https://web.archive.org/web/20130116045454/http://www.elasticsearch.org/blog/2010/02/08/youknowforsearch.html.

[110] solid IT gmbh. (Apr. 2016). DB-Engines Ranking of Search Engines, Adresse: http://db-engines.com/en/ranking/search+engine.

[111] C. Horohoe. (Jan. 2014). Wikimedia Blog: Wikimedia moving to Elasticsearch, Adresse: https://blog.wikimedia.org/2014/01/06/wikimedia-moving-to-elasticsearch/.

[112] T. Pease. (Jan. 2013). GitHub: A Whole New Code Search, Adresse: https://github.com/blog/1381-a-whole-new-code-search.

[113] N. Craver. (Nov. 2013). What it takes to run Stack Overflow, Adresse: http://nickcraver.com/blog/2013/11/22/what-it-takes-to-run-stack-overflow/.

[114] S. Loke und C. Kalantzis. (Nov. 2014). The Netflix Tech Blog: Introducing Raigad - An Elasticsearch Sidecar, Adresse: http://techblog.netflix.com/2014/11/introducing-raigad-elasticsearch-sidecar.html.

[115] P. Djekic. (Dez. 2012). Soundcloud Backstage Blog: Architecture behind our new Search and Explore experience, Adresse: https://developers.soundcloud.com/blog/architecture-behind-our-new-search-and-explore-experience.

[116] scikit-learn developers. (Juni 2016). Feature selection - scikit-learn documentation, Adresse: http://scikit-learn.org/stable/modules/feature_selection.html.

[117] ——, (Juni 2016). sklearn.decomposition.PCA - scikit-learn documentation, Adresse: http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

[118] ——, (Juni 2016). sklearn.ensemble.RandomForestRegressor - scikit-learn documentation, Adresse: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[119] GitHub. (Juni 2016). GitHub WebHooks, Adresse: https://developer.github.com/webhooks/.

[120] T. Zimmermann, N. Nagappan und A. Zeller, „Predicting Bugs from History“, in *SOFTWARE EVOLUTION*. Springer, März 2008, Kap. 4, S. 69–88, ISBN: 9783540764397.

[121] C. Kolassa, D. Riehle und M. A. Salim, „The Empirical Commit Frequency Distribution of Open Source Projects“, in *SYMPOSIUM ON OPEN COLLABORATION*, ACM, 2013, 18:1–18:8. Adresse: http://www.se-rwth.de/publications/The-Empirical-Commit-Frequency-Distribution-of-Open-Source-Projects.pdf.

[122] H. Drucker, C. J. C. Burges, L. Kaufman, A. J. Smola und V. N. Vapnik, „Support Vector Regression Machines“, *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, 1996.

6.2 Listingsverzeichnis

6.3 Abbildungsverzeichnis

6.4 Tabellenverzeichnis

6.5 Abkürzungsverzeichnis

ML Machine Learning

UML Unified Modeling Language

SVN Subversion

CVS Concurrent Versions System

PaaS Platform as a Service

REST Representational State Transfer

API Application Programming Interface

DB Datenbank

DBMS Database Management System

SQL Structured Query Language

ORM Object Relational Mapping

AST Abstract Syntax Tree

MSR Mining Software Repositories

SVM Support Vector Machine

SVR Support Vector Regression

RBF Radial Basis Function

CV Cross Validation

MAE Mean Absolute Error

MSE Mean Squared Error

MDE Median Absolute Error

PCA Principal Component Analysis

A Projektmanagement

Das Projektmanagement beschreibt die zur Erreichung des Ziels dieser Arbeit notwendigen Hilfs- und Planungswerkzeuge. Das Ziel wurde unter anderem durch die ursprüngliche Aufgabenstellung beeinflusst. Diese ist in Abbildung 1 zu finden.

A.1 Offizielle Aufgabenstellung

Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

Projektidee: Fehler im Java-Code automatisch erkennen
[Machine Learning + Software Engineering]
BA16_ciel_5

BetreuerInnen:	Mark Cieliebak, ciel
Fachgebiete:	Datenanalyse (DA) Software (SOW)
Studiengang:	IT
Zuordnung:	Institut für angewandte Informationstechnologie (InIT)
Gruppengrösse:	2

Kurzbeschreibung:
Was wäre wenn man schon beim Programmieren wüsste, wie viele Fehler der neue Code haben wird?

Wir wollen diesen Traum eines Entwicklers wahr machen, und ein System entwickeln das vorhersagt wie viele Fehler in einem Java-Programm stecken. Die Idee beruht auf zwei Beobachtungen:

1. Im **Maschinellen Lernen** gibt es sehr **gute Algorithmen zum automatischen Textverständnis**. Damit kann man unterschiedlichste Aufgaben lösen, z.B. wichtige Aktoren in Texten erkennen (Personen, Firmen etc.), Texte nach vorgegebenen Kategorien klassifizieren oder automatisch das Thema eines Textes erkennen. Viele dieser Algorithmen sind bereits fertig implementiert und können für eine konkrete Aufgabe angepasst werden, indem sie einfach auf einer grossen Menge von Trainingsdaten (10'000 und mehr) trainiert werden.

2. Es gibt **Unmengen von Open-Source-Projekten**, die nicht nur ihren Code, sondern auch ihr **gesamtes Issue-Tracking online** stellen und frei verfügbar machen. In den Daten findet man strukturierte Informationen, wann ein Issue (z.B. ein Bug) gelöst wurde, und welcher Code dafür verändert wurde.

Unsere Idee ist nun, die Daten aus vielen Open-Source-Projekten zu verwenden, um einen **Maschinellen Classifier zu trainieren, der potentielle Bugs im Java-Code** erkennen kann.

Dies ist ein sehr umfangreiches Thema, das auch gut von mehr als 2 Studierenden bearbeitet werden kann.

Dies ist nur eine grobe Projektidee. Falls Sie Interesse an diesem spannenden Thema haben, können wir gern einen Termin abmachen und die konkrete Aufgabenstellung besprechen. Email: ciel@zhaw.ch oder Telefon: 058 934 72 39.

Voraussetzungen:
- gute Java-Kenntnisse
- Bereitschaft sich in Machine Learning einzuarbeiten
- Kreativität und innovative Ideen

Die Arbeit ist vereinbart mit:
Tobias Meier (meierto3)
Yacine Smail Mekesser (mekesyac)

Dienstag 23. Februar 2016 15:30

Abbildung 1: Offizielle Aufgabenstellung

Die Zielsetzung hat sich im Laufe der Arbeit entwickelt und hing stark von der Literaturrecherche ab. Wären zum Beispiel bereits geeignete Tools für das Repository Mininig vorhanden gewesen, hätte es keinen Grund für die Entwicklung von *GtSooG* gegeben. Aufgrund dessen entschieden wir uns für eine iterative, flexible Projektplanung. Zudem sollte das Projektmanagement in einem sinnvollen Mass betrieben werden.

Als Planungswerkzeug verwendeten wir das Issue-Tracking-System von GitHub. Dort wurden Milestones und Tickets für die zu erledigen Aufgaben erstellt.

A.2 Grob Planung

Nach der Literaturrecherche entschieden wir uns die Arbeit in drei Teile zu Unterteilen. Daraus entstanden die Milestones in Tabelle 1.

Milestone	**Deathline**
Repository Mining	27.03.2016
Feature Extraction	24.04.2016
Machine Learning	22.05.2016
Documentation	08.06.2016

Tabelle 1: Milestones

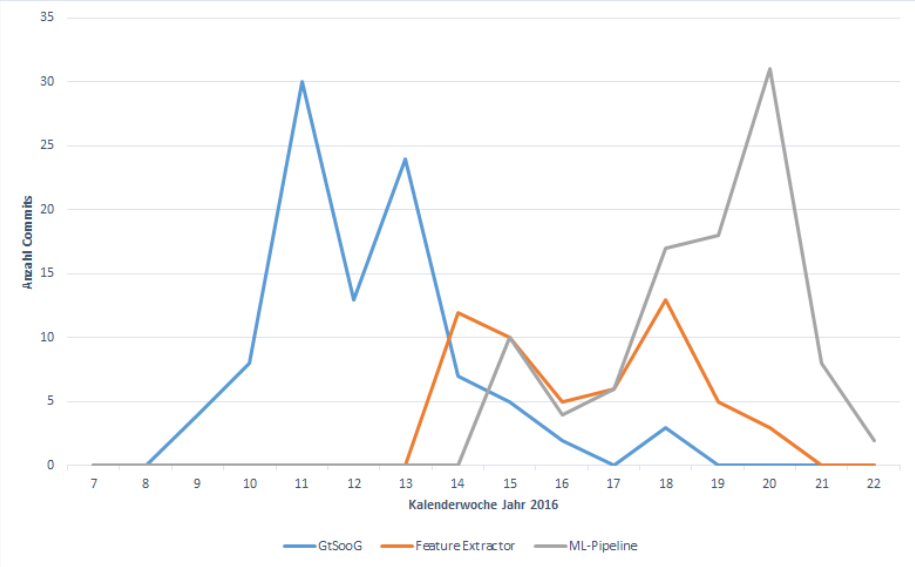

Abbildung 2: Projekt Commits geordnet nach Milestones über die Zeit

Den Milestone *Repository Mining* konnten wir wie geplant am 29.03.2016 abschliessen. Der Milestone *Feature Extraction* verzögerte sich und wurde am 18.05.2016 fertiggestellt. Im Laufe der Arbeit wurde uns klar, dass das sequentielle Abarbeiten von *Feature Extraction* und *Machine Learning* nicht optimal ist. Eine möglichst bald funktionierende Machine Learning Pipeline verschaffte uns mehr Zeit für Tests und die rechenintensive Machine-Learning-Durchläufe. Die Entwicklung vom Feature Extractor und der ML-Pipeline startete deshalb fast

parallel. Der Milestone *Machine Learning* konnte erst zum Abschluss der Arbeit fertiggestellt werden. Dies ist darauf zurückzuführen, dass die Experimente bis zum Schluss andauerten.

Die Abbildung 2 visualisiert die Anzahl Commits über die Zeit. Gut zu erkennen ist, wann wie intensiv an welchen Milestones gearbeitet worden ist.

A.3 Detail Planung

Ein Auszug aller GitHub Issues soll einen Einblick in die detaillierte Planung liefern.

Milestone	**Issue Titel**	**Datum erstellt**	**Datum geschlossen**
GtSoog	GtSooG: Commandline UI	08.03.2016	25.03.2016
GtSoog	GtSooG: Repository Miner	08.03.2016	21.03.2016
GtSoog	GtSooG: Issue Analyzer	08.03.2016	21.03.2016
GtSoog	GtSooG: DB Interface	14.03.2016	21.03.2016
GtSoog	GTSooG: IssueScanner Multithreading	21.03.2016	29.03.2016
GtSoog	GtSooG: IssueScanner Jira Schnittstelle	21.03.2016	26.03.2016
GtSoog	GtSooG: SQLite ersetzen	21.03.2016	26.03.2016
GtSoog	GTSoog: Changes müssen einer Zeile zugewiesen werden	21.03.2016	26.03.2016
GtSoog	GTSooG: RepositoryMiner Multithreading	25.03.2016	29.03.2016
Feature Extractor	FeatureExtractor: Architektur entwerfen	26.03.2016	17.04.2016
Feature Extractor	Feature: Lines of Code	29.03.2016	15.04.2016
Feature Extractor	Feature: Anzahl Code-Features	29.03.2016	16.05.2016
Feature Extractor	Feature: Vererbung	29.03.2016	26.04.2016
Feature Extractor	Feature: Coupling between Objects	29.03.2016	19.04.2016
Feature Extractor	GtSooG: Autor des Commits erfassen	29.03.2016	03.04.2016
Feature Extractor	Feature: Änderungsrate	29.03.2016	01.05.2016
Feature Extractor	Feature: Objekttyp	29.03.2016	16.05.2016
Feature Extractor	Feature: Länge von Namen	29.03.2016	16.04.2016
Feature Extractor	Lerndaten-Korpus erstellen	29.03.2016	12.04.2016
Feature Extractor	FeatureExtractor: Java Tokenizer evaluieren	29.03.2016	05.04.2016
Feature Extractor	GtSooG: Produktiv Machine installieren	29.03.2016	04.04.2016
Feature Extractor	GtSooG: Signal-Handling	04.04.2016	09.04.2016
Feature Extractor	GtSooG: Log soll in File und Konsole schreiben können.	04.04.2016	12.04.2016
Feature Extractor	Feature: Lack of Cohesion in Methods	05.04.2016	17.04.2016
Feature Extractor	Statistiken ziehen	05.04.2016	23.04.2016

Feature Extractor	FeatureExtractor: Grundgerüst implementieren	05.04.2016	09.04.2016
Feature Extractor	FeatureExtractor: Git-Anbindung	05.04.2016	09.04.2016
Feature Extractor	FeatureExtractor: DB-Design anpassen	05.04.2016	09.04.2016
Feature Extractor	FeatureExtractor: DB-Anbindung	05.04.2016	09.04.2016
Feature Extractor	Feature: N-Grams	05.04.2016	08.05.2016
Feature Extractor	FeatureExtractor: Signal Handling	05.04.2016	19.04.2016
Feature Extractor	Feature: Code Style nach Linus	06.04.2016	16.05.2016
Feature Extractor	GtSooG: Negative file size reparieren	18.04.2016	25.04.2016
Feature Extractor	Feature-Extractor: Code Komplexitätsfeature	26.04.2016	16.05.2016
Feature Extractor	Feature Extractor: Performance Optimierung Features in DB schreiben	03.05.2016	08.05.2016
Feature Extractor	Feature-Extractor: Feature-Groups ein und ausschalten	09.05.2016	10.05.2016
Feature Extractor	Feature-Extractor: History-Features gehen auf falsche Commits	10.05.2016	10.05.2016
Feature Extractor	Feature-Extractor: History-Features werden z.T. nicht angelegt	10.05.2016	13.05.2016
ML Application	ML-Pipeline: Feature Whitelist	23.04.2016	26.04.2016
ML Application	ML-Pipeline: Files ausschliessen	23.04.2016	17.05.2016
ML Application	ML-Pipeline: Baseline implementieren	26.04.2016	03.05.2016
ML Application	ML-Pipeline: Ridge Regression implementieren	26.04.2016	06.05.2016
ML Application	ML-Pipeline: SVR implementieren	26.04.2016	10.05.2016
ML Application	ML-Pipeline: Features cachen	26.04.2016	28.04.2016
ML Application	ML-Pipeline: SScoreboard"für Runs machen	02.05.2016	03.05.2016
ML Application	ML-Pipeline: Polynomielle Features einbauen	03.05.2016	08.05.2016
ML Application	ML-Pipeline: N-Grams einbinden	03.05.2016	15.05.2016
ML Application	ML-Pipeline mit elasticsearch Daten testen	06.05.2016	31.05.2016
ML Application	ML-Pipeline: Normalisierung auch für SVR einbauen.	08.05.2016	09.05.2016
ML Application	ML-Pipeline: Sinnvollen Output ausgeben	08.05.2016	10.05.2016

ML Application	ML-Pipeline: Validations- und Lernkurve als Bild exportieren	08.05.2016	09.05.2016
ML Application	ML-Pipeline: Reports speichern	08.05.2016	09.05.2016
ML Application	ML-Pipeline auf Server konfigurieren	09.05.2016	17.05.2016
ML Application	ML-Pipeline: Sortierung von Features / Konsistenz	10.05.2016	29.05.2016
ML Application	ML-Pipeline: Nur Java-Files berücksichtigen	10.05.2016	14.05.2016
ML Application	ML-Pipeline: Testfälle schreiben	10.05.2016	17.05.2016
ML Application	ML-Pipeline: Mit Mock-Datasets testen	10.05.2016	28.05.2016
ML Application	ML-Pipeline: Feature Scaling korrigieren	12.05.2016	12.05.2016
ML Application	ML-Pipeline: Lazy Loading konfigurierbar machen.	14.05.2016	14.05.2016
ML Application	ML-Pipeline: Matplotlib macht auf Server komische Dinge	17.05.2016	18.05.2016
ML Application	ML-Pipeline: In Reports auch Config anzeigen	18.05.2016	18.05.2016
ML Application	ML-Pipeline: Zusätzliche Parameter für SVR in Config	18.05.2016	21.05.2016
ML Application	ML-Pipeline: Sparse matrices in Datensets	18.05.2016	20.05.2016
ML Application	ML-Pipeline: Histogramm als Report aufnehmen	18.05.2016	18.05.2016
ML Application	ML-Pipeline: Log-Transformation von Target	18.05.2016	20.05.2016
ML Application	ML-Pipeline: StandardScaler für sparse implementieren	23.05.2016	24.05.2016
ML Application	ML-Pipeline: Detailliertere Auswertung nach Kategorien	23.05.2016	24.05.2016
ML Application	ML-Pipeline: Validation Curve hat kein Preprocessing.	23.05.2016	24.05.2016
BA Document	Feature Design dokumentieren	05.04.2016	29.05.2016
BA Document	Statistiken auswerten und Resultate/Erkenntnisse dokumentieren	19.04.2016	29.05.2016
BA Document	ML-Pipeline dokumentieren	19.04.2016	29.05.2016
BA Document	Feature-Extractor: Design dokumentieren	05.05.2016	28.05.2016
BA Document	Documentation: Installationsinstruktion für weitere Verwendung der Tools	09.05.2016	05.06.2016
BA Document	Documentation: Anwendungsinstruktion für alle Tools	09.05.2016	07.06.2016

BA Document	Documentation: Koordination mit SiB Dozent fürs Korrekturlesen	16.05.2016	21.05.2016
BA Document	Theoretische Grundlagen: Regressionsmodelle beschreiben	17.05.2016	28.05.2016
BA Document	Abstract / Zusammenfassung schreiben	17.05.2016	06.06.2016
BA Document	Projektleitung und Zeitaufwand dokumentieren	17.05.2016	30.05.2016
BA Document	Management Summary schreiben	17.05.2016	31.05.2016
BA Document	Titelblatt machen	17.05.2016	06.06.2016
BA Document	Vorwort schreiben	17.05.2016	06.06.2016
BA Document	Statistik: Histogramm Upcoming Bugs pro Version	20.05.2016	29.05.2016
BA Document	Documentation: Weitere Features	21.05.2016	29.05.2016
BA Document	Documentation: Diskussion	28.05.2016	07.06.2016
BA Document	Documentation: Ausblick	28.05.2016	05.06.2016
BA Document	Documentation: Resultate	28.05.2016	07.06.2016
BA Document	Documentation: Infrastruktur dokumentieren	28.05.2016	29.05.2016
BA Document	Documentation: Theoretische Grundlagen Machine Learning Einführung schreiben	31.05.2016	02.06.2016
BA Document	Documentation: GtSooG / Feature-Extractor Ablauf Vorgehensweise und Runtime Dauer	31.05.2016	04.06.2016
BA Document	Documentation: Literaturrecherche Gross / Kleinschreibung	31.05.2016	01.06.2016
BA Document	Documentation: Anhang in einzelne Dokumente verpacken	31.05.2016	01.06.2016
BA Document	BA-Dokument durchlesen und korrigieren	31.05.2016	07.06.2016
BA Document	N-Grams erklären	02.06.2016	04.06.2016
BA Document	Documentation: Ausblick Experimente Grafik für Sliding Window erstellen	04.06.2016	05.06.2016
BA Document	Documentation: Projektmanagement Issue Tabelle aktualisieren	06.06.2016	07.06.2016
BA Document	USB-Stick vorbereiten	06.06.2016	07.06.2016
BA Document	Arbeit in Publikationstool eintragen	07.06.2016	07.06.2016
BA Document	Finale, gemergte Version erstellen.	07.06.2016	08.06.2016

Tabelle 2: Issues

A.4 Meetings

Während der Arbeit fanden regelmässige Meetings mit unserem Betreuer Dr. Mark Cieliback, Fatih Uzdilli (wissenschaftlicher Mitarbeiter) und Dominic Egger (wissenschaftlicher Assistent) statt. Der Inhalt der Meetings bestand in der Regel aus folgenden Punkten:

- Statusbericht der Arbeit
- Klären offener Fragen
- Definieren des weiteren Vorgehen

Für jedes Meeting wurde ein Protokoll erstellt. Um den Umfang der Arbeit nicht unnötig zu erhöhen, wird darauf verzichtet alle Protokolle im Anhang abzubilden. Stattdessen soll Tabelle 3 die Meetings zusammenfassen.

Datum	**Traktanden**
07.03.2016	Zusammenfassung Literaturrecherche Definition Inhalt Bachelorarbeit Titel Bachelorarbeit Umfang Projektmanagement und Formalien
21.03.2016	Zeitplan GtSooG Vorstellen Machine Learning
29.03.2016	Design Feature Extractor
04.04.2016	Feature Extractor Vorstellen
11.04.2016	Machine Learning Design und offene Fragen
18.04.2016	Zielsetzung Statistiken Machine Learning Base-Line
25.04.2016	Nur Statusbericht
09.05.2016	Mikroplanung letzte Wochen Klären Formalien
25.05.2016	Mikroplanung letzte Wochen
06.06.2016	Klären Formalien vor Abgabe

Tabelle 3: Meetings

B Installationsanleitungen

Die folgende Anleitung beinhaltet Installationsinstruktionen zur Installation einer Machine-Learning-Umgebung sowie sie in der Bachelor Arbeit "Fehlervorhersage in Java-Code mittels Machine Learning"beschrieben ist.

Die zu installieren Tools sind:

- GtSooG
- Feature-Extractor
- ML-Pipeline

Aufgrund der verwendeten Programmiersprachen (Java und Python) ist das System Betriebssystem unabhängig. Diese Anleitungen beschreibt trotzdem die Installation auf einem unixähnlichem Betriebssystem (Ubuntu Server). Die Datenbank und die oben erwähnten Applikationen können auf autonomen Systemen installiert werden.

B.1 Betriebssystem Installation

Ein geeignetes System ist zum Beispiel Ubuntu Server 15.10. Da wir teilweise aktuelle Versionen von Paketen verwenden raten wir zum jetzigen Zeitpunkt von Debian ab. Natürlich funktionieren auch andere Linux und Windows Betriebssysteme.

Die Serverinstallation kann ohne spezielle Konfiguration durchgeführt werden.

B.2 Datenbank Installation

Als Datenbank kann grundsätzlich eine beliebige verwendet werden. Auf jeden Fall geeignet MySQL:

Listing 1: Datenbank Installation

```
$ wget https://dev.mysql.com/get/mysql-apt-config_0.6.0-1
_all.deb
$ dpkg -i mysql-apt-config_0.6.0-1_all.deb
$ apt-get update
$ apt-get install mysql-server
```

Die Datenbank kann auf dem selben System wie die oben erwähnten Applikationen installiert werden.

B.3 Anaconda Installation

GtSooG ist in Python programmiert. Da auch die ML-Pipeline auf Python basiert ist es empfehlenswert isolierte Python Umgebungen zu verwenden. Dies ist mit Anaconda möglich.

Die Installation per Anaconda kann wie folgt durchgeführt werden:

Listing 2: Anaconda Installation

```
$ wget http://repo.continuum.io/archive/Anaconda3-4.0.0
-Linux-x86_64.sh
$ ./Anaconda3-4.0.0-Linux-x86_64.sh
```

B.4 GtSooG Installation

GtSooG benötigt folgende Python Bibliotheken:

- requests
- sqlalchemy
- mysql-connector-python
- gitpython

Die Bibliotheken werden wie folgt installiert:

Listing 3: Einrichten Anaconda für GtSooG

```
$ conda create --name gtsoog requests sqlalchemy mysql
-connector-python
$ activate gtsoog
$ conda install -c https://conda.anaconda.org/conda-forge
gitpython
```

Die Programmdateien von GtSooG können direkt von der Git Repository geclont werden:

Listing 4: Installation GtSooG

```
$ cd /opt/
# Die Tools koennen entweder vom Git Repository geclont
werden oder vom Datentraeger kopiert werden
$ git clone https://github.engineering.zhaw.ch/BA16-ML
-Java-Analysis/GtSooG.git
$ cp ... /opt/
```

B.5 Feature Extractor Installation

Der Feature Extractor ist in Java programmiert. Die im Feature Extractor verwendeten Methoden sind auf Java Version 8 angewiesen. Die vom Feature Extractor verwendeten Java Bibliotheken werden mittels Maven automatisch installiert.

Folgendermassen kann Java auf einem Ubuntu Server System installiert werden:

Listing 5: Installieren Vorraussetzungen Feature Extractor

```
$ add-apt-repository ppa:webupd8team/java
$ apt-get update
$ apt-get install oracle-java8-installer
$ apt-get install oracle-java8-set-default
$ apt-get install maven
```

Der Feature Extractor installiert man wie folgt:

Listing 6: Installieren Feature Extractor

```
$ cd /opt/
# Die Tools koennen entweder vom Git Repository geclont
werden oder vom Datentraeger kopiert werden
$ git clone https://github.engineering.zhaw.ch/BA16-ML
-Java-Analysis/FeatureExtractor.git
$ cp ... /opt/
$ cd /opt/FeatureExtractor/
$ mvn install
```

B.6 ML-Pipeline Installation

GtSooG benötigt folgende Python Bibliotheken:

- scikit-learn
- sqlalchemy
- pymysql
- gitpython
- matplotlib
- terminaltables

Das Listing 7 erläutert die Installation.

Listing 7: Einrichten Anaconda für ML-Pipeline

```
$ conda create --name ml scikit-learn sqlalchemy pymysql
$ source activate ml
$ pip install terminaltables
$ conda install matplotlib
```

Die Programmdateien von der ML-Pipeline können wie folgt installiert werden:

Listing 8: Installieren ML-Pipeline

```
$ cd /opt/
# Die Tools koennen entweder vom Git Repository geclont
werden oder vom Datentraeger kopiert werden
$ git clone https://github.engineering.zhaw.ch/BA16-ML
-Java-Analysis/ML-Pipeline.git
$ cp ... /opt/
```

C Benutzeranleitung

Die Benutzeranleitung befasst sich mit der Anwendungsinstruktion. Es wird davon ausgegangen, dass die Umgebung bereits gemäss Der Installationsanleitung installiert ist. Die Benutzeranleitung beinhaltet Anweisungen für die folgenden Tools:

- GtSooG
- Feature Extractor
- ML-Pipeline

Die Anleitung beschreibt einen kompletten Durchlauf mit der Repository *elasticsearch*.

C.1 Vorbereitung Datenbank

Bevor die einzelnen Tools mit der Arbeit beginnen können, gilt es die Datenbank zu erstellen. Das Skript zur Initialisierung der Datenbank ist auf dem Datenträger im Ordner Datenbank vorhanden.

Listing 1: Datenbank erstellen

```
$ mysql --execute="CREATE SCHEMA gtsoog CHARACTER SET
utf8;"
$ mysql gtsoog < create_db.sql
```

C.2 Repository klonen

Nun gilt es ein Repository, das analysiert werden soll, (in unserem Fall elasticsearch) zu klonen:

Listing 2: Repository clonen

```
$ git clone https://github.com/elastic/elasticsearch.git
```

C.3 GtSooG

GtSooG wird per Config-Datei konfiguriert und anschliessend per Kommandozeilen-Befehl ausgeführt. In Listing 3 ist eine Beispielskonfiguration zu sehen. Weiter unten folgt eine Tabelle mit allen Konfigurationsparametern.

Listing 3: GtSooG Beispielskonfiguration

```
[DATABASE]
database_engine = mysql
database_name = gtsoog
database_user = root
database_user_password = root
database_host = localhost
database_port = 3306
```

```
[REPOSITORY]
repository_path = C:\dev\Repositories\elasticsearch
issue_tracking_system = GITHUB
issue_tracking_url = api.github.com/repos/elastic/
elasticsearch

[REPOSITORYMINER]
number_of_threads = 1
number_of_database_sessions = 1
write_lines_in_database = True

[PROGRAMMINGLANGUAGES]
Markdown=md
Python=py
Java=java

[ISSUESCANNER]
issue_id_regex=XD-[0-9]+

[LOGGING]
log_mode = BOTH
log_file = gtsoog.log
log_level = DEBUG
```

GtSooG kann nun wie folgt gestartet werden:

Listing 4: GtSooG ausführen

```
/opt/anaconda3/envs/gtsoog/bin/python3.5 GtSooG.py
-f gtsoog.cfg
```

C.3.1 CLI-Parameter

Parameter	Beschreibung
-f	Erforderlich. Gefolgt vom Pfad zur Konfigurationsdatei von GtSooG

Tabelle 1: GtSooG CLI-Parameter

C.3.2 Konfigurationsparameter

Sektion	Parameter	Beschreibung
DATABASE	database_engine	Datenbank Engine für sqlalchemy ORM
DATABASE	database_name	Datenbank Name für sqlalchemy ORM
DATABASE	database_user	Datenbank User für sqlalchemy ORM
DATABASE	database_password	Datenbank Password für sqlalchemy ORM
DATABASE	database_host	Datenbank Host für sqlalchemy ORM

DATABASE	database_port	Datenbank Port für sqlalchemy ORM
REPOSITORY	repository_path	Pfad zum lokalen Git Repository
REPOSITORY	issue_tracking_system	Issue Tracking System (JIRA oder GitHub)
REPOSITORY	issue_tracking_url	URL des Issue Tracking Systems
REPOSITORY	issue_tracking_username	Optional. Benutzername für das Issue Tracking.
REPOSITORY	issue_tracking_password	Optional. Passwort für das Issue Tracking.
REPOSITORY MINER	number_of_threads	Anzahl Threads (für zukünftige Verwendung)
REPOSITORY MINER	number_of_database_sessions	Anzahl Datenbanksitzungen (für zukünftige Verwendung)
REPOSITORY MINER	write_lines_in_database	Diff-Zeilenänderungen in Datenbank schreiben (True or False)
PROGRAMMING LANGUAGES	Java=java	Gibt die Datei Endung für eine Programmiersprache an. Für jede Programmiersprache, welche beachtet werden soll muss ein solcher Eintrag bestehen. GtSooG ignoriert Dateien mit anderen Endungen.
ISSUESCANNER	issue_id_regex	Identifiziert anhand von Regex Issues in Commits. Nur bei JIRA notwendig. Beispiel: XD-[0-9]+, erkennt die Issue-Nr XD-4125.
LOGGING	log_mode	Logmodus. PRINT, LOGFILE, BOTH
LOGGING	log_file	Dateiname für Logdatei
LOGGING	log_level	Logging Level ERROR, WARNING, INFO oder DEBUG

Tabelle 2: GtSooG Konfigurationsparameter

C.4 Feature Extractor

Der Feature Extractor wird per Config-Datei konfiguriert und anschliessend per Kommandozeilen-Befehl ausgeführt. In Listing 5 ist eine Beispielskonfiguration zu sehen. Weiter unten folgt eine Tabelle mit allen Konfigurationsparametern.

Listing 5: Feature Extractor Beispielskonfiguration

```
[DATABASE]
dialect = org.hibernate.dialect.MySQLDialect
driver = com.mysql.jdbc.Driver
url = jdbc:mysql://localhost:3306/gtsoog
user = root
userPassword = root

[REPOSITORY]
name = elasticsearch
```

```
[DEFAULT]
partitions = 250
logLevel = INFO
logFilename = FeatureExtractor.log

[FEATURES]
maxNGramSize = 5
featureGroups = ChangeRateFeatureGroup,
LengthOfNamesFeatureGroup, LinesOfCodeFeatureGroup,
ObjectOrientedFeatureGroup, NGramFeatureGroup
```

Der Feature Extractor kann nun wie folgt gestartet werden:

Listing 6: Feature Extractor ausführen

```
java -Xmx28g -cp target/feature-extractor-1.0-SNAPSHOT
-jar-with-dependencies.jar ba.ciel5.featureExtractor.
FeatureExtractor -f FeatureExtractor.cfg
```

C.4.1 CLI-Parameter

Parameter	**Beschreibung**
-Xmx28g	Optional, gibt die Heap Grösse von Java an. Standardwert 4 GB. Bei grossen Projekten muss dieser Wert entsprechend erhöht werden.
-f	Erforderlich. Gefolgt vom Pfad zur Konfigurationsdatei vom Feature Extractor

Tabelle 3: Feature Extractor CLI-Parameter

C.4.2 Konfigurationsparameter

Sektion	**Parameter**	**Beschreibung**
DATABASE	dialect	Datenbank Dialect für Hibernate ORM
DATABASE	driver	Datenbank Driver für Hibernate ORM
DATABASE	url	Datenbank URL für Hibernate ORM
DATABASE	user	Datenbank User für Hibernate ORM
DATABASE	userPassword	Datenbank Passwort für Hibernate ORM
DEFAULT	partitions	Maximale Anzahl Versions, welche pro Thread verarbeitet werden. Je grösser der Wert, desto mehr Versionen werden pro Thread verarbeitet. Default 250
DEFAULT	logLevel	Log Level ERROR, WARNING, INFO oder DEBUG
DEFAULT	logFilename	Dateiname Logfile

FEATURES	maxNGramSize	Maximale N-Gram Länge. Beim Wert von 5 werden N-Grams der Grösse 1 bis 5 gebildet.
FEATURES	featureGroups	Feature-Gruppen, welche generiert werden sollen

Tabelle 4: Feature Extractor Konfigurationsparameter

C.5 ML-Pipeline

Die ML-Pipeline wird per Config-Datei konfiguriert und anschliessend per Befehl ausgeführt. In Listing 7 ist eine Beispielskonfiguration zu sehen. Weiter unten folgt eine Tabelle mit allen Konfigurationsparametern.

Listing 7: ML-Pipeline Beispielskonfiguration

```
[DATABASE]
dialect = mysql
name = gtsoog
user = root
user_password = DogeHorse
host = localhost
port = 3306
eager_load = False

[LOGGING]
level = DEBUG
override = False
file = ml_pipeline_elastic

[REPORTING]
display_reports = False
save_reports = True
file = report_elastic
target_histogram = True
validation_curve = True
learning_curve = True
display_charts = False
save_charts = False

[REPOSITORY]
name = elasticsearch

[ML]
model = svr
feature_scaling = True
polynomial_degree = 1
cross_validation = True
log_transform_target = False
alpha = 1, 0, 0.01, 0.1, 10, 100, 1000
```

```
C = 1, 0.001, 0.01, 5, 10, 50, 100
kernel = rbf
svr_epsilon = 0.1
svr_gamma = 1000, 100, 10, 1, 0.1, 0.01, 0.001, 0.0001
svr_degree = 3
svr_coef0 = 0, 0.1, 1

[DATASET]
cache = False
sparse = True
target = sixmonths
train_start = 2014.10.01
train_end = 2014.12.31
test_start = 2015.01.01
test_end = 2015.01.31
ngram_sizes = 1, 2, 3
ngram_levels = 1, 2, 3, 4
features = AOAbstractClasses,
           AOClasses,
           AOConstants,
           AOEnums,
           AOFields,
           AOImplementedInterfaces,
           AOImports,
           [...]
```

Der ML-Pipeline kann nun wie folgt gestartet werden:

Listing 8: ML-Pipeline ausführen

```
/opt/anaconda3/envs/ml/bin/python3.5
        /opt/ML-Pipeline/ml_pipeline.py
```

C.5.1 CLI-Parameter

Parameter	Beschreibung
-c	Optional. Gefolgt vom Pfad zur Konfigurationsdatei. Falls nicht angegeben, wird eine Datei ml_pipeline.config im Working Directory gesucht.

Tabelle 5: ML-Pipeline CLI-Parameter

C.5.2 Konfigurationsparameter

Parameter	Beschreibung	Wertebereich	Opt.
Sektion: **DATABASE**			
dialect	Datenbank-Dialekt	*String*	Nein

name	Name der Datenbank bzw. des DB-Schemas	*String*	Nein
user	Username für den DB-Zugriff	*String*	Ja
user_password	Passwort für den DB-Zugriff	*String*	Ja
host	Hostname oder IP des DB-Servers	*String*	Ja
port	Port des DB-Servers	*String*	Ja
eager_load	Wenn aktiv, wird überall wo möglich Eager Loading eingesetzt. Kann zu hohem Speicherverbrauch führen.	*Boolean*	Ja

Section: **LOGGING**

level	Minimaler Logging-Level der geloggt werden soll	'debug', 'info', 'warning', 'error', 'critical'	Ja
file	Filename für das Logfile ohne Dateiendung	*String*	Ja
override	Wenn aktiv, wird dasselbe Logfile für jeden Durchlauf neu beschrieben. Ansonsten wird für jeden Lauf ein neues File erstellt.	*Boolean*	Ja
format	Format-String für die Log-Messages	*String*	Ja
date_format	Format-String für den Timestamp der Log-Messages	*String*	Ja

Section: **REPORTING**

display_reports	Ob Reports in der Kommandozeile angezeigt werden sollen.	*Boolean*	Ja
save_reports	Ob Reports in einem File gespeichert werden sollen.	*Boolean*	Ja
save_reports	Ob Reports in einem File gespeichert werden sollen.	*Boolean*	Ja
file	Filename für das Speichern von Reports, ohne Dateieindung.	*String*	Ja
target_histogram	Ob ein Chart mit einem Histogramm des Target-Vektors des Trainingsset gezeichnet werden soll	*Boolean*	Ja
validation_curve	Ob ein Chart mit der Validationskurve generiert werden soll (führt zusätzliche CV Durchläufe durch)	*Boolean*	Ja
learning_curve	Ob die generierten Charts angezeigt werden sollen. Macht nur auf einem OS mit grafischer Umgebung Sinn.	Ja	
display_charts	Ob die generierten Charts als PNG gespeichert werden sollen.	Ja	

Section: **REPOSITORY**			
name	Name des Repositories in der DB, mit dem gearbeitet werden soll	*String*	Nein
Section: **ML**			
model	Bezeichnung des Regressionsmodells, das benutzt werden soll.	'linear_regression', 'ridge_regression', 'svr'	Nein
polynomial_degree	Wenn grösser als 1 werden polynomiale Features mit dem entsprechenden Grad generiert. Benötigt viel Speicher.	*Integer*	Ja
feature_scaling	Ob Feature Scaling angewendet werden soll.	*Boolean*	Ja
log_transform_target	Ob ein Logarithmus auf den Target-Vektor der Datensets angwendet werden soll	*Boolean*	Ja
log_transform_base	Falls log_transform_target = True ist, kann hier zusätzlich noch die Basis angegeben werden.	*Integer*, n (für natürlichen Logarithmus)	Ja
alpha	Liste von α-Parametern für Ridge Regression. Falls cross_validation = False wird nur der erste Wert der Liste verwendet	*List[Float]*	Ja
C	Liste von C-Parametern für SVR. Falls cross_validation = False wird nur der erste Wert der Liste verwendet	*List[Float]*	Ja
cross_validation	Ob Cross Validation eingesetzt werden soll.	*Boolean*	Ja
kernel	Kernel-Funktion, die für SVR zu verwenden ist. Kann auch eine Liste für Cross Validation sein.	'linear', 'poly', 'rbf', 'sigmoid'	Ja
svr_degree	Liste von *degree*-Parametern für den SVR-Kernel 'poly'. Falls cross_validation = False wird nur der erste Wert der Liste verwendet	*List[Integer]*	Ja
svr_epsilon	Liste von *epsilon*-Parametern für alle SVR-Kernels. Falls cross_validation = False wird nur der erste Wert der Liste verwendet	*List[Float]*	Ja

svr_gamma	Liste von *gamma*-Parametern für den SVR-Kernel 'rbf'. Falls cross_validation = False wird nur der erste Wert der Liste verwendet	*List[Float]*	Ja
svr_coef0	Liste von *coef0*-Parametern für die SVR-Kernel 'poly' und 'sigmoid'. Falls cross_validation = False wird nur der erste Wert der Liste verwendet	*List[Float]*	Ja
Section: **DATASET**			
cache	Ob Datensets gecached werden sollen.	*Boolean*	Ja
sparse	Ob Sparse Datasets eingesetzt werden sollen. Nur in Kombination mit N-Grams sinnvoll	*Boolean*	Ja
cache_dir	Der Dateiordner, in dem die gecachten Datensets gespeichert werden sollen.	*String*	Ja
target	Die Grösse des Zeitraums, für den Bugs vorhergesagt werden sollen.	'month', 'six-month', 'year'	Nein
train_start	Das Startdatum des Trainingssets.	*Date*	Nein
train_end	Das Enddatum des Trainingssets.	*Date*	Nein
test_start	Das Startdatum des Testsets.	*Date*	Nein
test_end	Das Enddatum des Testsets.	*Date*	Nein
ngram_sizes	Liste von N-Gram-Grössen (entspricht dem N), welche in den Feature-Vektor aufgenommen werden sollen.	*List[Integer]*	Ja
ngram_levels	Liste von N-Gram-Levels (entspricht Abstraktionsgrad), welche in den Feature-Vektor aufgenommen werden sollen.	*List[Integer]*	Ja
features	Liste von Feature-IDs (wie sie in der DB gespeichert sind), welche in den Feature-Vektor aufgenommen werden sollen. Sollten alphabetisch absteigend geordnet sein.	*List[String]*	Nein

Tabelle 6: ML-Pipeline Konfigurationsparameter