HAW HAMBURG

MASTERTHESIS
Florian Schädler

Maschinelles Lernen für statische Codeanalysen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Florian Schädler

Maschinelles Lernen für statische Codeanalysen

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Bettina Buth
Zweitgutachter: Prof. Dr. Kai von Luck

Eingereicht am: 18. Mai 2021

Florian Schädler

Thema der Arbeit

Maschinelles Lernen für statische Codeanalysen

Stichworte

Statische Codeanalyse, Maschinelles Lernen, neuronale Netze, Graph Neural Network

Kurzzusammenfassung

In dieser Arbeit sollen die Aufgaben der statischen Codeanalyse durch die Methoden des maschinellen Lernens gelöst werden. Speziell geht es dabei um die Identifikation von Fehlern im Quellcode durch ein gelerntes Modell. Eine entsprechend gelernte Fehleranalyse soll Fehler mit unterschiedlichen Charakteristiken in beliebigen Abschnitten des Quellcodes ermitteln. Die vorliegende Aufgabe wird als Klassifikationsproblem definiert und durch die Konzeption und Implementation eines Prototyps auf die Machbarkeit überprüft. Durch eine Analyse des Prototyps auf neuen Eingangsdaten wird die Güte des Modells bewertet und diskutiert. Das Ergebnis zeigt, dass das Lernen einer solchen Fehleranalyse grundsätzlich möglich ist. Speziell in den Bereichen der Datenauswahl, der Datenvorbereitung, der Hyperparameteranpassung und der Interpretierbarkeit der Klassifizierungen bietet der Prototyp ein hohes Optimierungspotenzial.

Florian Schädler

Title of Thesis

Machine Learning for Static Code Analysis

Keywords

Static Code Analysis, Machine Learning, Neural Networks, Graph Neural Networks

Abstract In this work, the tasks of static code analysis are to be solved by machine learning methods. The focus is on the identification of errors in the source code by a trained model. A corresponding trained error analysis should identify errors with different characteristics in arbitrary sections of the source code. The task at hand is defined as a

classification problem and tested for feasibility by the conception and implementation of a prototype. The performance of the model is evaluated and discussed by analyzing the implementation on new input data. The results show that training such an error analysis is basically possible. Nevertheless, the subjects of data selection, data preparation, hyperparameter tuning as well as the interpretability of the classification results offer a high potential for optimization.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Bereits seit der Mitte des 20. Jahrhunderts wird Software für die Steuerung wichtiger Systeme verwendet [70]. Seitdem steigt die Bedeutung von Software, sodass immer mehr Bereiche des Alltags durch sie bestimmt werden. Neben Fernsehgeräten, Spielzeugen oder mobilen Endgeräten werden auch zunehmend sicherheitskritische Systeme wie Autos, Raketen, Ölpipelines oder Telekommunikationsnetze durch Software gesteuert. Mit der steigenden Bedeutung von Softwaresystemen wächst die Verantwortung, die die Verfasser:innen des Programmcodes tragen. Bereits kleine Unachtsamkeiten oder Missverständnisse können dabei zu Millionenschäden oder der Gefährdung von Menschenleben führen [30]. Darüber hinaus erschwert die im Durchschnitt zunehmende Software-Größe die Vermeidung von Fehlern. Im Schnitt geht man von 1 bis 25 Fehlern pro 1000 Zeilen Quellcode (LOC) in einer ausgelieferten Software aus [43]. Wird dieser Richtwert beispielhaft auf den stetig wachsenden Linux-Kernel angewandt, der im Januar des Jahres 2020 bereits 27,8 Millionen LOC enthält [38], resultiert eine Anzahl von 27.800 bis 695.000 Fehlern im aktuellen Linux-Kernel. Dabei lässt sich nur erahnen, welch schiere Größe die Fehlerzahl im Quellcode während der Entwicklung betrug; eine für Entwickler:innen nahezu unbeherrschbare Menge.

Im Laufe der Jahre haben sich verschiedene statische und dynamische Prüftechniken etabliert, um Fehler und das Fehlverhalten einer Software zu identifizieren [40]. Besonders die statischen Verfahren der Codeanalyse unterstützen Entwickler:innen ohne großen Aufwand, bereits während der Implementation durch Hinweise auf Fehler oder Konstrukte, die auf Fehler hinweisen. Unter dem Begriff *statische Codeanalyse* werden Softwareanalyseverfahren beschrieben, die ein Programm, ohne dessen Ausführung, verschiedenen Prüfungen unterziehen [40]. In den meisten Compilern sind bereits vielfältige lexikalische, syntaktische und semantische Prüfungen integriert, die den Quellcode auf Korrektheit überprüfen [30]. Jedoch sind diese bei Weitem nicht erschöpfend und begrenzen sich auf ein Minimum von Überprüfungen. Aus diesem Grund existieren verschiedene Programme zur werkzeugunterstützten statischen Codeanalyse, die über die grundlegen-

den Prüfungen des Compilers hinaus umfangreiche Analysen auf dem Programmcode durchführen [49].

Im Bereich der Softwareentwicklung sind die Methoden und Werkzeuge der statischen Codeanalyse allgegenwärtig. Sie sind in Entwicklungsumgebungen, Build-Pipelines oder weiteren Analysewerkzeugen integriert und unterstützen Entwickler:innen unter anderem sowohl durch die Überprüfung von Programmierkonventionen, Warnungen für potenziell fehlerhafte Programmkonstrukte als auch das Aufdecken von Daten- oder Kontrollflussanomalien [32]. Als Entwickler in verschiedenen Projekten, in denen unterschiedliche Programmiersprachen und Technologien verwendet werden, profitiert der Autor dieser Arbeit täglich von der Unterstützung statischer Codeanalysen. Vor allem während der Implementation erleichtern sie die Arbeit der Entwickler:innen essenziell. Durch die schnelle Identifikation von Problemen können viele Fehler bereits während der Implementierung behoben werden. Außerdem erleichtern diese Hinweise den Kontextwechsel zwischen den syntaktischen und semantischen Eigenheiten verschiedener Programmiersprachen.

Die meisten dieser Analyseprogramme verwenden für die Prüfungen ein manuell erstelltes Regelwerk [16]. Laut Chess [16] ist der Aufbau und die Pflege eines solchen Regelwerks ein großer Aufwand. Auch der Blick auf die Entwicklungshistorie einiger der meistverbreiteten statischen Analysewerkzeuge zeugt von einem zeitintensiven Prozess[1][2][3][4]. Im Grunde bestehen die Regeln aus Syntax und Daten- bzw. Kontrollflussmustern, die auf einen bestimmten Fehler hinweisen [49][40]. Die Analyse ist dementsprechend nur so gut wie das zuvor verfasste Regelwerk. Eine Analyse über dieses Wissen hinaus ist nicht möglich.

Vor dem Hintergrund, dass Open-Source Filehoster wie GitHub[5] eine stetig wachsende Anzahl von Projekten mit Programmcode öffentlich zur Verfügung stellen, erfreut sich seit einigen Jahren das Lernen von Programmanalysen durch das maschinelle Lernen an steigender Beliebtheit [3]. Aufbauend auf der erfolgreichen Anwendung vom maschinellen Lernen (ML) auf verschiedenen Problemfeldern wie z.B. der natürlichen Sprachverarbeitung (NLP) oder Netzwerkanalysen, hat sich ein neuer Bereich namens *Programm Language Processing* (PLP) entwickelt [47]. In diesem Rahmen wurden verschiedene Lernmethoden für die Anwendung auf Quellcodes optimiert. Hierdurch wird sowohl

[1] https://github.com/JetBrains/intellij-community
[2] https://github.com/pmd/pmd
[3] https://github.com/findbugsproject/findbugs
[4] https://github.com/checkstyle/checkstyle
[5] https://github.com/about/milestones

das Lernen von syntaktischen und semantischen Codeeigenschaften als auch dessen Prognose ermöglicht. In den vergangenen Jahren wurden bereits verschiedene Codeanalysen durch ML-Techniken gelernt. Das umfasst unter anderem sowohl das Vorschlagen von Variablennamen [6][17], die Identifikation von Fehlnutzungen einer Variable [4] oder semantischen Code-Clones [73] als auch Typinferenz [27] und die Identifikation von Bugs [53][67]. Letzteres bezieht sich vor allem auf die Identifikation von Fehlern im Quellcode, die sich in einer zusammenhängenden Tokensequenz manifestieren. Dies ist nur ein kleiner Ausschnitt einer stetig wachsenden Liste. Die Fähigkeit nichttriviale Eigenschaften von unscharfen Daten zu lernen, macht die ML-Techniken in diesen Bereichen so erfolgreich. Bisherige Arbeiten fokussieren sich maßgeblich auf Aufgaben, die bis dato nur schwer oder überhaupt nicht durch vordefinierte Heuristiken gelöst werden konnten.

Doch auch für das Lernen klassischer Aufgaben der statischen Codeanalyse, wie die Identifikation von Fehlern oder potenziell fehlerhaften Konstrukten, bieten die Techniken des ML verschiedene Vorteile. Nachfolgend werden hierfür die Begriffe *klassische Fehleranalyse* und *gelernte Fehleranalyse* verwendet. Eine somit automatisch gelernte Fehleranalyse bietet neben der offensichtlichen Zeitersparnis noch weitere Vorteile. So kann die Analyse ohne großen Aufwand von neuen Programmcodes lernen, somit stets aktuell gehalten werden und von neuen Erfahrungen lernen. Schließlich bietet der automatisierte Prozess die Möglichkeit, eine vom Menschen nahezu unüberschaubare Datenmenge zu erfassen und von dieser zu lernen. Hierdurch ergibt sich auf lange Sicht das Potenzial für weitere Vorteile. Dazu gehört die Identifikation von Zusammenhängen zwischen Fehlerindikatoren, die bisher verborgen blieben. Die Basis für das Lernen bietet die stetig wachsende Anzahl von Open-Source-Softwares (OSS) auf Filehostern wie GitHub[6]. Diese Arbeit basiert auf der Annahme, dass in den OSS genügend Fehler vorhanden sind, von denen eine Fehleranalyse gelernt werden kann. Einen weiteren langfristigen Vorteil für das Lernen einer solchen Analyse bietet das Forschungsgebiet *Mining Software Repositories*[7]. So ist es möglich, auf Basis von Commit-Nachrichten oder Code-Kommentaren fehlerhafte Codeabschnitte zu annotieren und die gelernte Analyse somit durch weitere Eingaben zu optimieren [34][41].

Die ML-Techniken bieten also eine vielversprechende Basis für die Konzeption und Erstellung einer gelernten Fehleranalyse. Bisherige Arbeiten in diesem Bereich unterscheiden bei der Fehleranalyse nicht zwischen Fehlerarten [67][39] oder analysieren lediglich zusammenhängende Tokensequenzen [53]. Letzteres verhindert die Analyse von Fehlern, die

[6] https://github.com/about/milestones
[7] http://www.msrconf.org/

sich über weit verteilte Elemente im Quellcode manifestieren. Aus diesem Grund soll in dieser Arbeit erprobt werden, wie sich die Aufgaben der klassischen Fehleranalyse durch ML-Methoden lernen lassen. Speziell geht es dabei um die Klassifikation von Fehlern oder potenziellen Fehlern, im Java-Quellcode. Zu diesem Zweck soll in dieser Arbeit ein Konzept für das Lernen von Fehlern im Programmcode erstellt und durch eine anschließende prototypische Implementation überprüft und diskutiert werden. Ferner soll die gelernte Fehleranalyse imstande sein, Fehler auf das korrespondierende syntaktische Element im Quellcode zurückzuführen und zwischen mehreren Fehlertypen zu unterscheiden. Dabei sollen auch Fehler identifiziert werden können, die sich durch weit verteilte Elemente im Quellcode ergeben. Das Konzept umfasst sowohl die Vorbereitung der Daten als auch den Aufbau eines Modells zur Fehleranalyse. Durch die anschließende Implementation soll die allgemeine Anwendbarkeit der gelernten Fehleranalyse auf neuen Daten bewertet werden.

Die ersten Schritte in diesem Themengebiet wurden in zwei vorbereitenden Arbeiten durchgeführt. In der ersten Ausarbeitung mit dem Titel *Datengetriebene statische Codeanalyse zur Ermittlung von nicht verwendeten Variablendeklarationen* [59] wurde ein erster Prototyp entwickelt, der nicht verwendete Variablendeklarationen in einem Java-Quellcode ermittelt. Die Identifikation eines einzelnen Fehlertyps ist jedoch weder sinnvoll noch in einem realistischen Anwendungsfall zu gebrauchen. Aus diesem Grund beschäftigt sich die zweite Ausarbeitung mit dem Titel *Datengetriebene statische Codeanalyse für Multi-Label Klassifikationsprobleme* [60] mit den unterschiedlichen Verfahren zur Lösung eines Multi-Label-Problems. Auch in der zweiten Arbeit wurden ausschließlich Variablen klassifiziert. Die Erkenntnisse der Arbeiten sind, dass die Auswahl der Methode zum Lösen eines Multi-Label-Problems stark abhängig von den zu lernenden Eigenschaften ist. Ferner wurde festgestellt, dass die Auswahl der Informationen, die dem System zum Lernen bereitgestellt werden, schlussendlich die Performance der Analyse maßgeblich beeinflussen. Weiterhin wurde bei dem verwendeten Verfahren die Notwendigkeit eines sehr großen Lerndatensatzes festgestellt. Es handelte sich schließlich um ein transduktives Verfahren.

Basierend auf der zuvor beschriebenen Aufgabe und den Erkenntnissen der vorangegangenen Arbeiten ist diese Arbeit wie folgt aufgebaut. Zunächst soll in Kapitel 2 eine Auswahl relevanter Fehler getroffen werden. Hier liegt das Hauptaugenmerk auf einer Auswahl von Fehlern, die sich auf unterschiedliche Art und Weise manifestieren. Das soll die allgemeine Anwendbarkeit des Konzepts demonstrieren. Nachfolgend werden in Kapitel 3 die Methoden bestehender klassischer Fehleranalysen zur Analyse der Fehlerauswahl

untersucht. Ziel dessen ist eine Übersicht über mögliche Repräsentationen des Quellcodes sowie die Identifikation relevanter Merkmale zum Auffinden der Fehler. Anschließend folgt in Kapitel 4 die Definition des ML-Problems. Das umfasst eine Erläuterung der Kriterien, nach welchen die Analyse gelernt werden soll. Daraufhin werden in Kapitel 5 die Besonderheiten beim Lernen vom Programmcode erläutert. Ferner werden gängige Verfahren aufgezählt und aufgrund ihres Nutzens für das vorliegende Problem bewertet. In Kapitel 6 wird auf Basis der Erkenntnisse der vorherigen Kapitel ein Konzept erstellt. Das Konzept wird im Anschluss in Kapitel 7 in Form eines Prototyps gelernt und auf seine Performance überprüft. Daraufhin folgt in Kapitel 8 sowohl eine Analyse der Ergebnisse als auch eine Diskussion im Hinblick auf den Nutzen einer gelernten Fehleranalyse. In Kapitel 9 wird die Arbeit durch ein Fazit mit Ausblick auf weiterführende Themen abgeschlossen.

2 Fehler im Programmcode

Je nachdem, ob eine Programmiersprache interpretiert oder kompiliert wird, sowie dem verwendeten Programmierparadigma, ergeben sich unterschiedliche Varianten und Auswirkungen von Programmfehlern. Damit in dieser Arbeit möglichst konkrete Probleme behandelt werden können, erfolgt zunächst die Eingrenzung auf eine Programmiersprache. Eine der relevantesten und weltweit verbreitetsten Sprachen ist die Programmiersprache *Java*[1]. Java ist eine streng statisch typisierte, objektorientierte und kompilierte Programmiersprache, die unter anderem in Serveranwendungen, mobilen Applikationen oder Embedded Systems eingesetzt wird [31]. Durch die strenge Typisierung und Kompilierung werden bereits vor der Ausführung des Programms lexikalische, syntaktische und einige semantischen Fehler identifiziert. Infolgedessen spezialisieren sich die klassischen Fehleranalysen vielmehr auf die Identifikation von potenziell fehlerhaften Programmkonstrukten, die ein unerwartetes Verhalten produzieren können.

2.1 Was sind Fehler?

Fehler können sich auf verschiedenen Ebnen im Programmcode etablieren. Aus diesem Grund folgt zunächst ein Überblick des Programmierprozesses der Programmiersprache Java. Im Zuge dessen werden wichtige Begriffe für den weiteren Verlauf der Arbeit definiert. Eine Software ist ein aus Daten und Programmen bestehendes System [20]. Das Programm ist ein, in einer spezifischen Programmiersprache verfasster Algorithmus, bestehend aus Algorithmen, Datenstrukturen, Funktionen und Prozeduren. Java ist eine hohe Programmiersprache. Diese bestehen aus klartextlichen Schlüsselwörtern sowie syntaktischen, grammatikalischen und semantischen Regeln, mit denen ein in Klartext lesbares Programm, der Quellcode, verfasst wird. Dieser besteht auf der lexikalischen Ebene aus einer Aneinanderreihung von Token [30]. Ein Token ist eine lexikalische Einheit, die für den Parser der Programmiersprache einen semantischen Sinn ergibt. In der

[1] https://www.java.com/

Programmiersprache Java gibt es beispielsweise die fünf Tokenarten: *Schlüsselwörter*, *Bezeichner*, *Literale*, *Operatoren* und *Separatoren* [23]. Schlüsselwörter sind reservierte Wörter der Programmiersprache, wie *if* und *return*, die eine feste Bedeutung für den Compiler haben. Bezeichner sind alphanumerische Begriffe, die als Name für Methoden, Klassen, Variablen usw. verwendet werden. Literale sind konstante und fixe Werte wie Nummern, Texte oder boolesche Werte. Durch Operatoren wie + oder - werden Operationen auf zwei Operanten definiert. Die Separatoren wie *{ }* oder *( )* werden von dem Compiler verwendet, um beispielsweise Blöcke oder Parameteraufzählungen zu erkennen. Der aus Token bestehende Quellcode wird vor der Ausführung durch den Compiler in von Maschinen interpretierbaren Zwischencode und Maschinencode übersetzt. Ein Compiler ist ein Computerprogramm, das Quellcode einer bestimmten Programmiersprache in eine Form übersetzt, die von einem Computer ausgeführt werden kann. Zu diesem Zweck wird die Konformitätsanalyse (lexikalische, syntaktische und semantische Analyse) ausgeführt [30]. Der resultierende Syntaxbaum wird anschließend in eine vom Computer ausführbare Sprache übersetzt [69]. Quellcode, Zwischencode und Maschinencode sind aufeinander aufbauende Programmcodearten, die allesamt unter dem Begriff Programmcode zusammengefasst werden.

Laut Liggesmeyer [40] beschreibt der Begriff *Fehler* in der Softwareentwicklung die Ursache für den Ausfall oder das Fehlverhalten einer Software. Ferner ist jeder Fehler statisch im Programmcode vorhanden. Die Ursache für Fehler sind in der Regel Irrtümer oder Tippfehler der Entwickler:innen. Ein Irrtum ist unter Umständen das falsche Verständnis über die Funktionsweise von Anweisungen einer Programmiersprache. Fehler können sich auf verschiedenen Ebnen des Programms etablieren und äußern sich dementsprechend durch unterschiedliche Fehlverhalten. Allgemein werden diese Fehler in *lexikalische*, *syntaktische* und *semantische* Fehler kategorisiert [30].

Ein lexikalischer Fehler manifestiert sich in einem Programm auf der lexikalischen Ebene [30]. Wie eingangs erwähnt, besteht ein typisches Programm auf dieser Ebene aus einer Aneinanderreihung von Token. Wenn sich unter diesen ein fehlerhafter Token befindet, dann führt das bei kompilierten Sprachen zum Abbruch des Scanners, also einem lexikalischen Fehler. Fehlerhafte Token sind solche, die nicht den Spezifikationen der Programmiersprache entsprechen. Das können beispielsweise falsch geschriebene Schlüsselwörter oder Bezeichner mit nicht zulässigen Symbolen sein. In der Regel führen Fehler auf der lexikalischen Ebene zu einer Verletzung der zugrunde liegenden Grammatikregeln und dadurch zu einem Abbruch der Kompilierung.

Syntaktische Fehler, kurz Syntaxfehler, sind Grammatikfehler. Sie treten auf, wenn Fehler in der syntaktischen Struktur des Quellcodes vorhanden sind [69]. Das können unter anderem Fehler in der Groß- und Kleinschreibung, fehlende Token oder dessen falsche Reihenfolge sein. Infolgedessen kann der Parser die syntaktische Struktur des Quellcodes nicht erkennen, sodass dieser mit einem Abbruch reagiert. Das resultiert, wie bereits beim lexikalischen Fehler, in einem Abbruch der Kompilierung und hat ein nicht ausführbares Programm zur Folge.

Die Einordnung von semantischen Fehlern wird in der Literatur häufig unterschiedlich gehandhabt [30][69][31][48]. Im Kern gleichen sich die Definitionen jedoch in einem Punkt: Semantisch fehlerbehafteter Programmcode ist lexikalisch und syntaktisch fehlerfrei. Der Begriff Semantik beschreibt in der Softwareentwicklung die inhaltliche Bedeutung der im Programmcode verwendeten Begriffe [20]. Das bedeutet, jeder Fehler, der auf einer inhaltlich inkorrekten Anweisung im Programmcode basiert, ist ein semantischer Fehler. Typische Fehler dieser Art sind nicht deklariert Variablen, inkompatible Typenkonvertierungen, die Verwendung von nicht anwendbaren Operatoren, Präzisionsverlust durch das Konvertieren zwischen Zahlentypen oder der Zugriff auf ein nicht vorhandenes Array-Element [48]. Bei stark typisierten Programmiersprachen wie Java werden viele dieser Fehler von Compiler erkannt. So wird der Programmcode bei der semantischen Prüfung des Compilers beispielsweise auf Typsicherheit und vorhandene Deklarationen von Variablen untersucht. Ein Verstoß der Regeln führt auch hier zum Abbruch der Kompilierung und hat ein nicht ausführbares Programm zur Folge. Es gibt jedoch semantische Fehler, wie das Teilen durch Null oder der Zugriff auf nicht vorhandene Array-Elemente, die nicht während der Kompilierung auffallen. Fehler dieser Art führen dann zu einem Laufzeitfehler. Das Themengebiet der statischen Codeanalyse bietet jedoch Methoden, mit denen ein Großteil der semantischen Fehler vor der Ausführung des Programms identifiziert werden können. Die bisher genannten Fehler resultieren in den meisten Fällen aus Fehlern in der Nutzung von Elementen der Programmiersprache. Es gibt jedoch auch semantische Fehler, die diesbezüglich fehlerfrei sind. Sie resultieren vielmehr aus einem falsch einprogrammierten Verhalten der Software, das Abweichung zur Spezifikation oder der Intention der Entwickler:innen darstellt [48]. Folglich führt ein solcher Fehler auch nicht zum Abbruch des Programms, sondern es führt zum Fehlverhalten der Software und Ergebnissen, die von der Spezifikation abweichen. Beispiele für diese Fehler sind die falsche Reihenfolge oder Verwechslung von Operanden, Auswahl falscher Datentypen oder fehlplatzierte Semikola. Weil diese Fehler in einem lexikalisch, syntaktisch und für den Compiler semantisch fehlerfreien Quellcode vorkommen, sind sie besonders schwer

zu identifizieren. Aus diesem Grund können sie kritische Folgen haben. In der Vergangenheit haben Fehler dieser Art schon zu erheblichen wirtschaftlichen Schäden und sogar zur Gefährdung von Menschenleben geführt [30].

2.2 Bad Coding Practices

Entsprechende Fehler sind für Entwickler:innen schwer zu identifizieren und können erhebliche Folgen haben. Aus diesem Grund soll in dieser Arbeit eine Codeanalyse gelernt werden, die Entwickler:innen zur Implementationszeit auf potenzielle Fehler hinweist. Dabei muss es sich nicht um einen konkreten Fehler handeln. Es geht vielmehr darum, eine Warnung zu produzieren, wenn das Zusammenspiel von Codeelementen oder ein bestimmtes Programmkonstrukt einen Fehler indizieren. Während der Implementation werden die Komponenten des Programms umgesetzt. Durch die Erstellung und Vergabe von Klassen, Datentypen, Methoden und Namen wird das Programm erstellt [20]. Die Qualität der Implementation hat großen Einfluss auf die Komplexität und Wartbarkeit des Programms. Im Laufe der vergangenen Jahrzehnte haben sich verschiedene Programmierpraktiken, sogenannte *Bad Coding Practices*[2] (BCP), herauskristallisiert, die auf nicht sorgfältig entwickelte oder gewartete Software hinweisen. Diese Codepraktiken erhöhen die Komplexität und verschlechtern sowohl Wartbarkeit als auch Leserlichkeit. Das steigert insbesondere bei der Weiterentwicklung die Wahrscheinlichkeit auf Fehler und somit dem Fehlverhalten des Programms. Aus diesem Grund sollen in dieser Arbeit BCPs im Programm identifiziert werden.

Auf der Plattform *Common-Weakness-Enumeration*[3] (CWE) wird ein gemeinschaftlich geführtes Verzeichnis gängiger Fehlertypen geführt. Das beinhaltet unter anderem eine Auflistung verschiedener BCPs für die Programmiersprache Java. Auf dessen Basis soll eine Auswahl von BCPs getroffen werden, die sich auf verschiedene Programmelemente zurückführen lassen. Diese Arbeit stellt die Hypothese auf, dass alle Fehler die durch eine klassische Fehleranalyse identifiziert werden können, auch durch eine gelernte Fehleranalyse erkannt werden können. Dabei begrenzt sich diese Arbeit jedoch auf Fehler, die sich innerhalb einer Java-Klasse etablieren. Eine Klassenübergreifende Fehleranalyse ist nicht Bestandteil dieser Arbeit. Zur Demonstration der allgemeinen Anwendbarkeit zeichnet sich die nachfolgende Fehlerauswahl durch Fehler mit unterschiedlichen Charakteristiken

[2] https://cwe.mitre.org/data/definitions/1006.html
[3] https://cwe.mitre.org/

aus. Das umfasst Fehler, die auf unterschiedliche syntaktische Elemente des Quellcodes zurückgeführt werden. Ferner sind Fehler enthalten, die sich durch das Zusammenspiel weit verteilter Elemente im Quellcode etabliert und solche, die sich durch ein einzelnes syntaktisches Element manifestieren. Hierdurch soll die allgemeine Anwendbarkeit der gelernten Fehleranalyse demonstriert werden.

Fehlender Default-Case im Switch-Statement Laut der CWE stellt dieser Fehler ein häufiges Problem in der Softwareentwicklung dar. Wenn in einem Switch-Statement der Default-Case fehlt, dann werden im Entscheidungsfall nicht alle Werte berücksichtigt. Infolgedessen arbeiten weiterführende Prozesse möglicherweise auf falschen Informationen und kaskadieren den Fehler im System. Das kann im Anschluss zu Fehlern führen, die in einem Laufzeitfehler oder dem Fehlverhalten des Programms resultieren.

```
switch (x) {
 case 1 -> y += 1;
 case 2 -> y += 2;
}
```

Listing 2.1: Fehlender Default-Case im Switch-Statement

Ein Beispiel der BCP ist dem Listing 2.1 zu entnehmen. Wenn x nicht den Wert 1 oder 2 hat, dann wird keine Veränderung an y vorgenommen. Das vorliegende Problem äußert sich dementsprechend innerhalb des Switch-Statements durch das Fehlen entsprechender Case-Statements.

Abhängigkeit vom Package-Level-Scope Dieser Fehler beschreibt eher den grundlegenden Fall, dass Entwickler:innen sich bei sicherheitsrelevanten Themen nicht auf den Package-Level-Scope verlassen sollen. Der Scope in Java-Programmen ist nämlich eine Hilfe beim Gestalten der Software und kein Sicherheits-Feature. Diese BCP hängt entsprechend stark vom Kontext ab und ist nicht ohne Weiteres durch ein generelles Schema zu erkennen.

```
class Test {
 String var1 = "kann veraendert werden";
 static final String var2 = "kann nicht veraendert werden";
}
```

Listing 2.2: Abhängigkeit vom Package-Level-Scope

Es gibt jedoch Richtlinien, die dieses Problem reduzieren können. So sollen z.B. Feldvariablen wie *var2* in Listing 2.2 nach Möglichkeit immer final und statisch sein. Hierdurch wird das Verändern der Werte von außerhalb der Klasse verhindert. Die Variable *var1* in Listing 2.2 kann trotz des Package-Level-Scopes von außerhalb der Klasse verändert werden. In diesem Fall lässt sich das Problem also auf die Variablendeklaration zurückführen.

Aktiver Debug Code Während der Entwicklung werden oft Programmkonstrukte in den Quellcode eingebaut, die ausschließlich zum Debuggen der Software gedacht sind. Wenn diese Konstrukte mit ausgeliefert werden, kann dies Sicherheitsrisiken mit sich bringen und schließlich zur ungewollten Manipulation des Programmablaufs führen. Bereits einfache Methodenaufrufe können für ein entsprechendes Risiko sorgen.

```
try {
 System.out.println("Verbinde mit " + ip + "! Passwort: " + password);
}catch(Exception e){
 e.printStackTrace();
 System.err.print(e);
}
```

Listing 2.3: Aktiver Debug Code

Ein Beispiel dessen ist in Listing 2.3 zu finden. Die Methoden *printStackTrace() System.out.print* oder *System.err.print* schreiben die Ausgabe direkt in die Konsole. Dabei wird keine Unterscheidung zwischen Test und Produktivsystemen gemacht. Befinden sich in der entsprechenden Nachricht sensible Informationen, so können diese von Angreifern verwendet werden, um das System zu manipulieren. Deshalb sollte es vermieden werden, diese zwei Methoden zu verwenden. Diese Probleme lassen sich entsprechend auf die expliziten Methodenaufrufe zurückführen.

Dead-Code In der Literatur finden sich verschiedene Definitionen für den Begriff *Dead-Code*. In dieser Arbeit umfasst der Begriff sowohl Programmteile, die nie ausgeführt werden [72], als auch Codeabschnitte, dessen Ergebnisse im weiteren Verlauf des Programms nicht verwendet werden [8]. Während der Kompilierung überprüft der Java-Compiler im Rahmen der Konformitätsanalyse den Quellcode bereits auf unerreichbare Statements [23]. Das umfasst alle Codeabschnitte, die im Kontrollfluss nach einer unendlichen Schleife, in einer niemals aktivierten Schleife oder nach Schlüsselwörtern wie z.B. break, continue oder return liegen. Für nachfolgende Analysen bleibt die Identifikation

nicht verwendeter Codeelemente. Dabei handelt es sich um nicht verwendete Imports, Variablen oder Methoden. Nicht verwendete Codeelemente weisen auf eine schlechte Codequalität hin. Sie sind ein Anzeichen dafür, dass etwas im Programmablauf vergessen wurde oder die entsprechenden Codezeilen noch nicht fertig entwickelt wurden. Die Konsequenzen können sich auch hier wieder durch Laufzeitfehler oder unerwartetes Verhalten der Software ausdrücken.

```
1 import com.example.NichtVerwendeterImport;
2 class Test {
3  private int nichtVerwendeteFeldvariable = 1;
4  private void nichtVerwendeteMethode(int nichtVerwendeterParameter) {
5   int nichtVerwendeteLokaleVariable = 1;
6   nichtVerwendeteLokaleVariable = 2;
7   nichtVerwendeteLokaleVariable = 3;
8  }
9 }
```

Listing 2.4: Dead-Code

Nicht verwendete Imports, Variablen und Methoden sind relativ einfach zu erklären. In Listing 2.4 befindet sich ein Beispiel für jedes der nicht verwendeten Codeelemente. Sie äußern sich bei Imports (Zeile 1) und Methoden (Zeile 4) dadurch, dass auf die Deklaration keine Referenzierung folgt. Bei Variablen ist das Problem etwas komplizierter, denn der Wert von Variablen kann zur Laufzeit verändert werden. Außerdem muss man hier zwischen schreibenden und lesenden Zugriffen auf die Variable unterscheiden. Eine nicht verwendete Variable ist eine solche, auf die im weiteren Verlauf des Programms nicht lesend zugegriffen wird. Variablen, bei denen auf eine Wertzuweisung eine weitere Wertzuweisung folgt, ohne dass in der Zwischenzeit der erste Wert verwendet wurde, nennt man *Dead-Store*. Ein entsprechender Dead-Store ist in Listing 2.4 in Zeile 5 und 6 verzeichnet. Auf die Deklaration in Zeile 5 folgen erneute Wertzuweisungen in Zeile 6 und 7. In diesem Fall sind die Variablendeklaration in Zeile 5 und die Variablenreferenz in Zeile 6 also Dead-Stores. Im oben gelisteten Beispiel werden also die Variablendeklarationen in Zeile 3, 4 und 5 nicht verwendet. Die Variable in Zeile 6 hat gleich zwei Probleme, denn sie ist ein Dead-Store und wird nicht verwendet. Während nicht verwendete Imports, Methoden und Variablen direkt auf die Deklaration zurückgeführt werden können, so kann sowohl eine Variablenreferenz als auch eine Variablendeklaration das Problem eines Dead-Stores beinhalten.

Nur eine der Methoden *equals* und *hashcode* definiert In Java verlassen sich eine Vielzahl von Methoden darauf, dass zwei Klassen, die sich über die Methoden *equals*

gleichen auch einen entsprechend identischen hashcode als Rückgabewert der Methode *hashcode* haben [23]. Wird diese Konvention nicht eingehalten, kann es speziell bei der Verwendung von Collections zu Fehlern und unvorhersehbarem Fehlverhalten kommen.

```
class Test {
 private int property;
 @Override
 public boolean equals(Object o) {
  return property == ((Test) o).property;
 }
}
```

Listing 2.5: Nur *equals* definiert

Im Listing 2.5 ist ein entsprechender Fehler abgebildet. Die *equals* Methode überprüft die Gleichheit zweier Klassen über die Feldvariable *property*. Weil die *hashcode* Methode nicht definiert wurde, wird die Standard-Implementation verwendet, die z.B. den hashcode durch die interne Speicheradresse berechnet. Somit gleichen sich zwei Klassen, die den gleichen Wert im Feld *property* haben über die *equals* Methode, jedoch nicht über die *hashcode* Methode. Deshalb ist es eine Konvention, dass immer beide Methoden überladen werden müssen. Dieser Fehler ergibt sich aus den implementierten Methoden und lässt sich direkt auf die Methodendeklarationen zurückführen.

Funktionsaufruf mit fehlerhaft spezifizierten Parametern Dieser Fehler beschreibt den Fall, dass eine Methode mit fehlerhaften Parametern aufgerufen wurde. In Java wird dieser Fehler weitgehend durch die Typisierung und Fixierung der Parameteranzahl reduziert.

```
1 int hoehe = 100; int breite = 100; int laenge = 100; int tiefe = 100;
2 wichtigeBerechnung(breite, tiefe, laenge, hoehe);
3
4 void wichtigeBerechnung(int hoehe, int breite, int laenge, int tiefe) {
5 //Wichtige Entscheidungen auf Basis der Parameter
6 }
```

Listing 2.6: Vertauschte Parameter beim Funktionsaufruf

Wie in Listing 2.6 abgebildet, kann es bei Methoden, die viele Parameter mit dem gleichen Typen haben dennoch zu entsprechenden Problemen kommen. Die Methode *wichtigeBerechnung* wird in Zeile 4 deklariert und in Zeile 2 mit vertauschten Parametern aufgerufen. Der Fehler wird vom Compiler nicht erkannt, da die Anzahl und Typen der

Parameter korrekt sind. Die *Wichtigen Entscheidungen* in Zeile 5 basieren also auf fehlerhaften Werten. Der Fehler kann über die Aufrufe oder den Rückgabewert der Methode entsprechend im System kaskadieren und zu Laufzeitfehlern oder unerwartetem Verhalten führen. Entsprechend sollten Methoden mit zu vielen Parametern vermieden werden. Dieses Problem kann direkt auf die Methodendeklaration zurückgeführt werden.

3 Statische Codeanalysen

Bisher wurden einige Fehlerarten und dessen Folgen erläutert. Ferner wurde eine Auswahl relevanter Fehler für diese Arbeit getroffen. Im nächsten Schritt soll die Funktionsweise bestehender klassischer Fehleranalysen untersucht werden. Ziel dessen ist die Einsicht in etablierte Prozesse, die aufschlussreich für das Lernen einer Codeanalyse sein können. Von Interesse sind dabei maßgeblich die unterschiedlichen Repräsentationen während der Analyse sowie die Logik zum Auffinden der Fehler. Letzteres bietet Aufschluss über relevante Merkmale zur Identifizierung eines Fehlers.

Der Ursprung der werkzeugunterstützten statischen Codeanalyse, die über eine rein lexikalische und syntaktische Prüfung hinausgeht, liegt in den 70er Jahren [30]. Durch die wachsende Bedeutung der Programmiersprache *C* und der einhergehenden steigenden Heterogenität der Systeme wurden auf vielen Systemen immer mehr Laufzeitfehler festgestellt. Um dem entgegenzuwirken, hat Stephen C. Johnson [33] einen C-Compiler entwickelt, der neben einer lexikalischen und syntaktischen Analyse eine semantische Überprüfung durchführte. Hierdurch konnte bereits vor der Ausführung der Programme ein Großteil der Codeabschnitte, die potenziell Fehlerhaft sind, identifiziert werden. Der Analyseteil des Compilers wurde in den nachfolgenden Jahrzehnten als eigenständiges Programm namens *Lint* stetig weiterentwickelt. Der Erfolg führte zu einer bis heute stetig steigenden Anzahl an statischen Codeanalyse Programmen für die unterschiedlichsten Programmiersprachen. In modernen Compilern, wie dem Java-Compiler von Oracle[1], befinden sich bereits verschiedene semantischen Analysen. Diese decken jedoch nur sehr grundlegende Prüfungen ab, sodass in den meisten Fällen weitere statische Codeanalyse Programme in den Build-Pipelines, Entwicklungsumgebungen oder Qualitätsverwaltungsprogrammen eingesetzt werden.

In diesem Abschnitt wird der Prozess einer klassischen statischen Codeanalyse zum Auffinden von (potenziellen) Programmfehlern vorgestellt. Anschließend werden die Verfahren zur Identifikation der Fehlerauswahl in Abschnitt 2.2 in diesem Prozess eingeordnet.

[1] https://docs.oracle.com/en/java/javase/12/tools/javac.html

Hierdurch soll maßgeblich das Verständnis für den Prozess geschaffen werden, der im Anschluss gelernt wird. Das ist sowohl für die Definition der Anforderungen an das zu lernende Modell relevant als auch grundlegend für den Aufbau des Datenverständnisses.

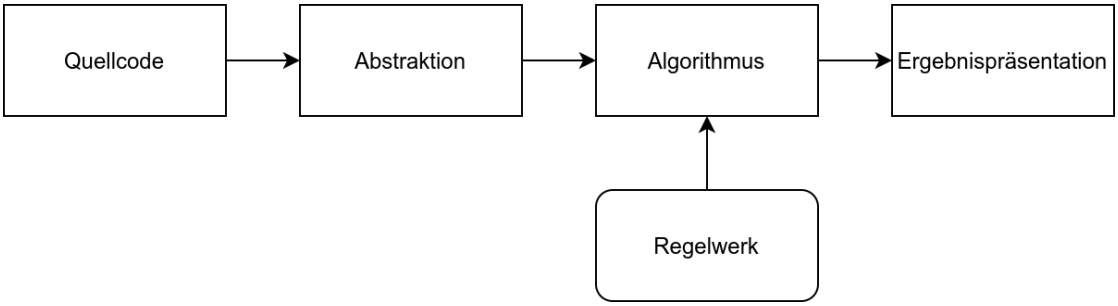

Abbildung 3.1: Prozess der klassischen statischen Codeanalyse [16]

Alle statischen Codeanalysen folgen grob dem in Abbildung 3.1 skizzierten Ablaufplan [16]. Eine aus Quellcode bestehende Eingabe wird abstrahiert und im Anschluss durch einen Algorithmus auf Basis eines Regelwerks analysiert. Nach Abschluss der Analyse werden die Ergebnisse den Entwickler:innen in einer aufbereiteten Form präsentiert. Der erste Block *Quellcode* wurde bereits in Kapitel 2 ausreichend erläutert. Er beinhaltet die in einer Programmiersprache verfassten Anweisungen an einen Computer. Der darauffolgende Block *Abstraktion* beschreibt den Prozess der Transformation des Quellcodes in einen Zwischencode (IR). Ziel dessen ist die Aufbereitung des Quellcodes in eine abstrahierte Ansicht, die eine Analyse des Programms unterstützt. Im nachfolgenden Abschnitt 3.1 werden gängige Abstraktionen vorgestellt. Der anschließende Block *Algorithmus* wendet die Regeln *Regelwerk* auf die *Abstraktion* an. Zum Verständnis der Blöcke *Algorithmus* und *Regelwerk* werden die Methoden der statischen Codeanalyse PMD[2] zum Auffinden der ausgewählten Fehler untersucht. Der abschließende Block der Ergebnispräsentation wird in dieser Arbeit vernachlässigt, da der Fokus auf dem Lernen der Analyse liegt.

3.1 Abstraktion des Quellcodes

Die meisten Methoden für die Erstellung einer Abstraktion des Quellcodes stammen aus der Forschung im Compilerbau. Die Funktionsweise von Werkzeugen zur statischen Codeanalyse ähneln denen des Compilers ohnehin sehr stark [30]. So beginnen beide mit

[2] https://pmd.github.io/

dem Bau einer Abstraktion durch die lexikalische und syntaktische Analyse. Die lexikalische Analyse bildet zumeist den ersten Schritt. Während der Analyse werden die Token des Quellcodes gelesen [65]. In diesem Zuge werden unwichtige Token wie Leerzeichen oder Kommentare entfernt. Befinden sich im Quellcode ungültige Zeichen oder Schlüsselwörter, so führt dies zu einem lexikalischen Fehler. Die syntaktische Analyse wird auch Parsen genannt [30]. Hierbei wird das Resultat der lexikalischen Analyse, also eine Aneinanderreihung von Token, auf syntaktische Vorgaben überprüft. Syntaktische Fehler führen hier zum Abbruch der Analyse. Bei syntaktischer Korrektheit kann der Parser die syntaktische Struktur des Programms identifizieren [69]. Das Resultat der Analyse besteht zumeist aus einem *Syntaxbaum*, der die ineinander geschachtelten Einheiten des Programms enthält. *Syntaktische Einheiten* sind zum Beispiel Variablen, Ausdrücke, Anweisungen, Anweisungsfolgen und Deklarationen.

Der **Syntaxbaum** ist eine der direktesten Abstraktionen des Quellcodes [16]. Die Repräsentation ist sehr nahe an dem tatsächlichen Quellcode und deshalb gut für Style Checks geeignet. Da er jedoch Details der Grammatik, syntaktische Feinheiten und Symbole für den Parser beinhaltet, ist er für weiterführende Überprüfungen nur begrenzt zu gebrauchen.

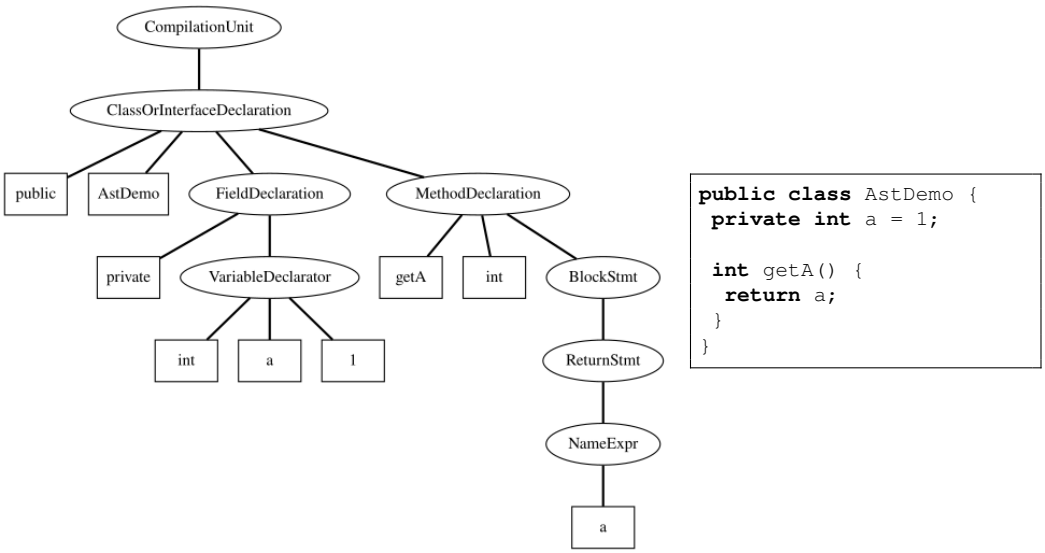

Abbildung 3.2: Skizzierung eines AST mit dem zugehörigen Quellcode

Eine Repräsentation, die von diesen Eigenschaften abstrahiert und eine standardisierte Version des Programms bereitstellt, ist der **Abstract Syntax Tree (AST)**. Eigenheiten in der Syntax des Quellcodes werden unter anderem durch Vereinfachungen in eine

gleichmäßige Struktur übertragen, die eine Analyse des Programms erleichtert. In Abbildung 3.2 ist ein Beispielprogramm und dessen AST abgebildet. Aus Platzgründen und zur besseren Übersicht ist der AST stark vereinfacht dargestellt. Die ovalen Knoten sind sogenannte *innere Knoten* und spiegeln die syntaktischen Elemente des Quellcodes wider. Hierfür wurde in dem Beispiel jeweils die Bezeichnung des syntaktischen Elements in den Knoten geschrieben. Die rechteckigen Knoten sind *Blätter*. Sie bilden auf das konkrete lexikalische Element im Quellcode ab. Um das zu verdeutlichen, wurden die Blätter mit den konkreten Token beschriftet. Unwichtige Elemente wie z.B. Klammern oder Semikola werden im AST nicht abgebildet. Die Knoten im AST werden durch Kanten verbunden. In der Regel wird der AST mit weiteren semantischen Informationen wie Typen, Deklarationen oder der Gültigkeit von Variablen angereichert. Das geschieht während der semantischen Analyse durch eine Symboltabelle, in der der Kontext zu jeder Verwendung oder Deklaration von Variablen, Klassen und Methoden verzeichnet ist. Das ermöglicht dem Analyseprogramm Zugriff auf Datentypen, überladene Funktionen sowie Informationen über die Nutzung von Funktionen und Variablen. Im Falle des Beispiels in Abbildung 3.2 kann also bei der Nutzung der Variable *a* im *ReturnStmt* auf den *VariableDeclarator* und somit auf den Typen und Wert der Variable geschlossen werden. Im gegensatz zum Quellcode in Textform bietet der AST eine einheitlich strukturierte Repräsentation der Programmsyntax.

Der **Control-Flow-Graph (CFG)** bietet eine weitere Darstellungsmöglichkeit des Quellcodes, welche den Kontrollfluss der Befehle des Programms hervorhebt [40]. Der CFG ist ein gerichteter Graph in dem jeder Knoten einem sogenannten *Basic Block* entspricht und jede Kante den Kontrollfluss zwischen zwei Knoten darstellt [2]. Ein Basic Block ist eine Folge von Anweseisungen, die im Quellcode der Reihenfolge nach ausgeführt werden und in denen es keine Verzweigungen gibt. Im Gegensatz zum AST bietet eine Darstellung mit Fokus auf dem Kontrollfluss wichtige Programmablaufinformationen. Das ermöglicht unter anderem das Auffinden von nicht erreichbaren Segmenten im Quellcode. Eine weitere gängige Vorgehensweise ist es, den CFG um Datenflussattribute zu erweitern, um somit einen **Datenflussgraphen (DFG)** zu bilden. Diese beziehen sich auf das Lesen oder Schreiben von Variablen. Kanten können somit das Attribut *p-use* haben, wenn die Entscheidung für eine Verzweigung auf einer Variable basiert. Konten können mit den Attributen *c-use* und *def* annotiert werden. Das Attribut def bezieht sich auf den schreibenden Zugriff einer Variable. Wenn in einem Knoten eine Variable zum Berechnen eines neuen Werts verwendet wird, dann wird der Knoten mit dem Attribut c-use annotiert. Datenflussattribute geben Aufschluss darüber, in welcher Art und Weise auf Variablen

zugegriffen wird. Das hilft z.B. beim Auffinden von nicht verwendeten Variablen oder dem Bestimmen von Nebenläufigkeiten. Eine weitere spezielle Form des CFG ist der **Call-Graph (CG)**. Anstatt den Kontrollfluss einzelner Anweisungen innerhalb einer Routine darzustellen, bezieht sich der CG auf die Aufrufe zwischen unterschiedlichen Subroutinen oder Klassen.

3.2 Analyse des Quellcodes

Die eigentliche Analyse des Quellcodes besteht aus Algorithmen, die auf Basis der zuvor erläuterten Abstraktionen und des Regelwerks eine Prüfung durchführen [16]. Die meisten Analyseprogramme führen dabei sowohl eine lokale als auch eine globale Überprüfung durch. Durch die lokale Analyse werden individuelle Funktionen überprüft. Zu diesem Zweck werden in der Regel die Abstraktionen AST und CFG verwendet [16]. Die globale Analyse beschäftigt sich mit der Überprüfung von Interaktionen zwischen den Funktionen oder Klassen. Beide Varianten verwenden einen Algorithmus, der definiert, wie die Überprüfung durchgeführt wird. Im Regelwerk ist unterdessen zumeist in Form von Bedingungen festgehalten, was überprüft werden soll.

Nachfolgend werden die Methoden des Programms PMD zum Auffinden der Fehlerauswahl untersucht. PMD ist eine in Java geschriebene quelloffene statische Codeanalyse zum Finden von Fehlern im Quellcode. Dadurch, dass der Quellcode von PMD öffentlich zugänglich ist, kann eine direkte Untersuchung der Funktionsweise durchgeführt werden. PMD umfasst insgesamt[3] 310 Regeln für Java-Programme. Die Basis für die meisten Regeln ist ein AST, der durch eine Symboltabelle mit semantischen Informationen angereichert wurde. Für einige Regeln werden jedoch auch CFGs und DFGs verwendet. PMD verwendet verschiedene Algorithmen, um die Regeln auf die Abstraktionen anzuwenden. Einfache Regeln sind in Form einer XPath-Suche hinterlegt. Das ist eine Query auf der XML-Repräsentation des ASTs. Einfache Regeln sind solche, in denen z.B. nur nach einem bestimmten Bezeichner oder Schlüsselwort gesucht wird. Kompliziertere Regeln werden durch das Besucher-Entwurfsmuster realisiert. Die Regeln werden entsprechend als Java-Klassen definiert und durch das Besucher-Entwurfsmuster auf dem AST angewendet. PMD bietet ausschließlich Prüfungen im Rahmen einer Java-Klasse an. Das bedeutet, dass für die Analyse ausschließlich die Programmkonstrukte einer Java-Datei verwendet werden. Eine Analyse von Fehlern, die sich dateiübergreifend äußern, wird

[3] https://pmd.github.io/pmd-6.31.0/pmd_rules_java.html

von PMD nicht durchgeführt und ist auch nicht Bestandteil dieser Arbeit. Nachfolgend befindet sich eine Beschreibung der Regeln für die ausgewählten Fehler aus Kapitel 2.

Fehlender Default-Case im Switch-Statement Diese BCP gehört zu den einfacheren und wird durch die Regel *SwitchStmtsShouldHaveDefault* mit einer entsprechenden XPath-Suche realisiert. Dabei werden alle Switch-Statements im AST auf die Boolschen-Eigenschaften *DefaultCase* und *ExhaustiveEnumSwitch* überprüft. Das Switch-Statement im AST wird während der semantischen Analyse durch die Symboltabelle um die entsprechenden Eigenschaften erweitert. Die Eigenschaft *DefaultCase* ist wahr, wenn der default-case im Switch-Statement vorhanden ist. Die Eigenschaft *ExhaustiveEnumSwitch* ist wahr, wenn der Eingabetyp des Switch-Statements ein Enum ist und alle möglichen Enum-Werte durch entsprechende Case-Einträge abgedeckt wurden. Wenn beide Eigenschaften nicht Wahr sind, dann liegt ein Fehler vor. Entsprechend wichtige Merkmale für die Identifikation dieses Fehlers sind sowohl die Case-Einträge des Switch-Statements als auch der Typ und die möglichen Werte der Eingabe. Informationen über die möglichen Werte der Eingabe sind nur dann vorhanden, wenn der Eingabetyp in der vorliegenden Java-Datei definiert ist.

Abhängigkeit vom Package-Level-Scope PMD bietet für diesen Fehler zwei Regeln. Eine Feldvariable, deren Wert nur bei der Initialisierung zugewiesen wird, kann final sein. Der Name der Regel ist *ImmutableField*. Eine Feldvariable, die final ist, kann auch statisch sein. Der Name dieser Regel ist *FinalFieldCouldBeStatic*. Diese beiden Regeln werden ausschließlich auf privaten Feldvariablen angewandt. Beide Regeln sind in Java-Klassen definiert und werden durch das Besucher-Entwurfsmuster angewandt. Durch die Regel *ImmutableField* werden alle Feldvariablen, die privat, aber nicht final sind, auf dessen Referenzen untersucht. Referenzen aus dem Konstruktor der Klasse werden nicht untersucht. Wenn kein schreibender Zugriff vorliegt, dann handelt es sich um einen Fehler. Ausgenommen von der Analyse sind Feldvariablen, die unter dem Einfluss eines vordefinierten Subsets an Annotationen stehen. Entsprechend wichtige Merkmale für dessen Identifizierung sind also sowohl die Modifizierer und Annotationen der Feldvariable als auch alle Referenzen und deren umfassende Methode. Feldvariablen, die final sind, aber nicht statisch werden durch eine entpsrechende XPath-Suche identifiziert. Die Suche findet alle Felddeklarationen die den Modifizierer final, aber nicht den Modifizierer static, haben. In diesem Fall sind also nur die Modifizierer der Feldvariable relevant.

Aktiver Debug Code PMD bietet zwei Regeln, die dem Auffinden dieser Fehler dienen. Diese Fehler sind relativ einfach zu finden und werden durch eine XPath-Suche realisiert. Dabei werden alle Methodenaufrufe gesucht, die genau auf Methoden mit dem Namen *System.out.println*, *System.err.print* oder *printStackTrace* zugreifen. Entsprechend kann der Fehler alleinig durch den Namen der Methode, die aufgerufen wird, identifiziert werden. Die Namen der Regeln sind entsprechend *SystemPrintln* und *AvoidPrintStackTrace*.

Dead-Code Für die Identifizierung von Dead-Code gibt es insgesamt sechs Regeln, die allesamt durch das Besucher-Entwurfsmuster realisiert werden. Die PMD-Regel zum Auffinden nicht verwendeter Imports *UnusedImports* vergleicht für jede Import-Deklaration alle im AST enthaltenen Typen-, Methoden- und Variablenreferenzen. Wenn der Import kein einziges Mal referenziert wird, dann liegt ein Fehler vor. Relevante Informationen sind also alle Referenzen auf die Import-Deklaration.

Die Regel zum Auffinden nicht verwendeter privater Methoden *UnusedPrivateMethod* überprüft im AST alle Referenzen der Methodendeklaration. Methoden, die unter dem Einfluss eines bestimmten Subsets an Annotationen stehen, werden in der Analyse ignoriert. Liegen keine Referenzen vor, so handelt es sich um eine nicht verwendete Methode. Wenn Referenzen gefunden werden, wird überprüft, ob sich der Methodenaufruf innerhalb der korrespondierenden Methode befindet. Befindet sich mindestens eine Methodenreferenz nicht in dem eigenen Methodenkörper, so handelt es sich um eine reguläre Methode, andernfalls liegt ein Fehler vor. Entsprechend wichtige Merkmale für die Identifikation nicht verwendeter Methoden sind sowohl die Annotationen an der Methodendeklaration als auch die Methodenreferenzen sowie der Kontext, in dem die Methode referenziert wurde.

Zum Auffinden nicht verwendeter Variablen stellt PMD insgesamt drei Regeln zur Verfügung. Aufgrund der unterschiedlichen Rahmenbedingungen gibt es für lokale Variablen *UnusedLocalVariable*, Parameter *UnusedFormalParameter* und Feldvariablen *UnusedPrivateField* verschiedene Regeln. Im Prinzip ähneln sich die Regeln insofern, als dass in jeder die Referenzen einer Variable auf ihre tatsächliche Nutzung untersucht werden. Eine Nutzung besteht nur bei lesenden Zugriffen auf die Variable. Schreibende Zugriffe sind entsprechend keine Nutzung. Feldvariablen haben die größte Reichweite und können somit Referenzen im gesamten AST haben. Parameter und lokale Variablen befinden sich nur im Rahmen einer Methode und haben dementsprechend nur hier ihre Referenzen.

Relevante Merkmale für die Identifikation von nicht verwendeten Variablen sind somit alle lesenden Referenzen der Variable.

Dead-Stores werden durch die Regel *UnusedAssignment* mithilfe eines DFG identifiziert. Im DFG sind die Def-Use-Ketten der einzelnen Variablen verzeichnet. Eine Anomalie in dieser Kette kann indikativ für einen Dead-Store sein. Die Regel zum Auffinden dieser Probleme generiert aus dem AST einen DFG und führt eine Reaching-Definition-Analyse durch. Wenn mindestens zwei defines direkt aufeinander folgen, dann liegt eine Dead-Store vor. Für dessen Identifikation ist also der Datenfluss wichtig. Speziell die Information, ob die Wertezuweisung einer schreibenden Referenz, eine lesende Referenz erreich.

Nur eine der Methoden *equals* und *hashcode* definiert Die Regel mit dem Namen *OverrideBothEqualsAndHashcode* für dieses Problem ist durch eine Java-Klasse mit dem Besucher-Entwurfsmuster implementiert. Zunächst wird die Klassen-Deklaration im AST auf die Implementierung des Comparable-Interface überprüft. Klassen mit dem Comparable-Interface überprüfen die Gleichheit durch eine separate Methode und sind für die Analyse dieses Problems nicht von Bedeutung. Wenn keine Implementation vorliegt, wird die Klasse auf die Überladung der Methoden *equals* und *hashcode* überprüft. Dabei wird die spezielle Signatur der Methoden verifiziert. Wenn nur eine der beiden Methoden überladen wurde, dann liegt ein Fehler vor. Wichtige Indikatoren sind sowohl die Implementationen der Klasse als auch die überladenen Methoden und deren Signatur.

Funktionsaufruf mit fehlerhaft spezifizierten Parametern Für diese BCP beinhaltet PMD eine Regel mit dem Namen *ExcessiveParameterList*, die die Parameteranzahl einer Methode mit einem Schwellwert vergleicht. Liegt die Zahl über dem Schwellwert, der standardmäßig mit dem Wert 10 belegt ist, so liegt ein Fehler vor. Dieser Fehler kann entsprechend auf Basis der Methodensignatur identifiziert werden.

4 Statische Codeanalyse als ML-Problem

In dieser Arbeit wird die Anwendbarkeit von ML-Methoden auf das Aufgabenfeld der klassischen statischen Codeanalyse untersucht. Die Vorteile einer gelernten Fehleranalyse ergeben sich aus dem automatisierten Lernprozess und der Fähigkeit, nichttriviale Eigenschaften in unscharfen Daten zu erkennen. Im Bezug auf die Erstellung und Weiterentwicklung der gelernten Fehleranalyse, wird der Aufwand durch den automatisierten Lernprozess im Vergleich zur klassischen Fehleranalyse reduziert. Schließlich sind für die Erstellung und Erweiterung der Analyse lediglich weitere Beispieldaten vonnöten, die auf Filehostern wie GitHub massenhaft zur Verfügung stehen. Außerdem ermöglicht der automatisierte Lernprozess die Verarbeitung von großen Datenmengen. In diesen können ML-Methoden komplizierte Zusammenhänge und nichttriviale Eigenschaften erkennen. Hierdurch wird das Erlernen einer Fehleranalyse ermöglicht, die sich nur schwer durch manuell verfasste Heuristiken realisieren lässt. Schlussendlich bietet dies Grund zu der Annahme, dass eine gelernte Fehleranalyse möglicherweise Zusammenhänge zwischen den Merkmalen des Quellcodes und den zu identifizierenden Fehlern erkennt, die durch ein manuell verfasstes Regelwerk nicht zu realisieren sind. In der Konsequenz bietet eine gelernte Fehleranalyse das Potenzial für ein umfassenderes Regelwerk und eine bessere Performance.

Um sich diesem Thema zu anzunähern wurde in Kapitel 2 bereits eine Eingrenzung des Problems auf eine Auswahl von Fehlern und die Programmiersprache Java getätigt. Ferner wurden in Kapitel 3 die Methoden der klassischen statischen Codeanalyse für Identifikation der ausgewählten Fehler untersucht. Das gewährte außerdem Einblick in wichtige Merkmale des Quellcodes, die für Erkennung der Fehler von hoher Relevanz sind. Schließlich bildet der Quellcode die Grundlage für das Lernen der Fehleranalyse und entsprechende Merkmale sind ausschlaggebend für die Performance der Analyse [75]. Nachfolgend soll schließlich die Fehleranalyse als ML-Problem definiert werden.

Durch die Methoden des ML werden Computer so programmiert, dass auf Basis von Beispieldaten ein bestimmtes Leistungskriterium optimiert wird [7]. Zur Definition dessen,

was ein System lernen soll, kann die Beschreibung des Lernens von Mitchell [45] verwendet werden. *"Für ein Computerprogramm lässt sich sagen, es lerne aus **Erfahrung E** hinsichtlich einer Klasse von **Aufgaben T** und einer **Leistungsbewertung P**, wenn seine Leistungsfähigkeit bezüglich der mit dem Maß P bewerteten Aufgaben aus T mit der Erfahrung E steigt."* Durch die Definition der Komponenten T, P und E wird festgelegt, was gelernt werden soll. Um zu verstehen, wie die Aufgabe gelernt wird, bedarf es der Erklärung weiterer Komponenten.

Das Lernen der Aufgabe T wird durch ML-Algorithmen realisiert. Egal ob überwachte, unüberwachte oder bestärkende Lernalgorithmen verwendet werden, der ML-Algorithmus lernt eine Funktion $f(x) = y$ [21]. Dabei passt der ML-Algorithmus die Parameter der Funktion f auf Basis der Erfahrungen E in Bezug auf das Maß P an. Die gelernte Funktion wird im ML-Kontext auch Modell genannt. Erfahrungen befinden sich in einem Lerndatensatz, der durch die verschiedene Vorbereitungsschritte für die Lernaufgabe optimiert wird [58]. Die Vorbereitung der Daten wird als *Feature-Engineering* bezeichnet, an dessen Ende eine numerische Repräsentation der Rohdaten steht [75]. Eine numerische Repräsentation ist in der Regel ein Vektor oder eine Matrix. Das Feature-Engineering ist ein wichtiges Bindeglied zwischen den Rohdaten und dem ML-Algorithmus. Denn bei ML-Algorithmen handelt es sich um mathematische Verfahren, die eine numerische Eingabe erfordern. Diese numerische Repräsentation wird in der Regel als Merkmale (*Features*) bezeichnet und entspricht einem Vektor $x \in \mathbb{R}^n$. Jede Komponente in x entspricht einem Feature. Das Modell lernt auf dessen Basis in Abhängigkeit des Maßes P die zugrundeliegende Verteilung der Daten, das sogenannte Signal. Features sind Aspekte oder Eigenschaften von rohen Daten wie Codeabschnitten, die als Teil der Feature-Engineerings manuell, oder durch ML-Methoden automatisch, extrahiert werden.

Neben der Auswahl des ML-Algorithmus beeinflussen sowohl der Datensatz, das Feature-Engineering als auch das Maß die Güte (Performance) des gelernten Modells in Bezug auf die Aufgabe. Aufgrund dessen werden nachfolgend für jede der Komponenten grundlegende Definitionen und Annahmen für diese Arbeit in Bezug auf das vorliegende Problem definiert.

4.1 Die Aufgabe: Fehleridentifikation

Unter dem Begriff Aufgabe versteht sich in diesem Kontext nicht das Lernen an sich, sondern jene Aufgabe, welche gelernt werden soll [22]. Die Aufgabe wurde bereits in den

vorherigen Kapiteln deutlich vermittelt, nämlich die Identifikation der Fehlerauswahl aus Abschnitt 2.2. Speziell sollen die in Abschnitt 3.2 beschriebenen Regeln der Analyse gelernt werden. Eine solche Analyse entspricht einer Multi-Label-Klassifikation, in der für jede Eingabe $0...n$ voneinander unabhängige Fehler prognostiziert werden [28]. Wie bereits in Abschnitt 2.2 demonstriert, kann nämlich ein untersuchtes Codeelement mehrere Fehler beinhalten. Im Rahmen dieser Arbeit soll das Klassifizierungsproblem jedoch vereinfacht durch eine Multi-Class-Klassifizierung beschrieben werden. Eine Multi-Label-Klassifizierung hat einen weitaus größeren Wertebereich in der Ausgabe und ist daher komplizierter zu lösen [28]. Im Kontext der Klassifizierung entspricht eine Klasse einem zu prognostizierenden Fehler. Nachfolgend werden diese Begriffe austauschbar verwendet. Weil ML-Algorithmen nur mit numerischen Werten arbeiten, kann die zu lernende Funktion wie folgt definiert werden: $f : \mathbb{R}^n \rightarrow \{k_1, ..., k_i | k \in \mathbb{R}, 0 \leq k, \sum_{i=1}^{K} k_i = 1\}$. Der Eingabevektor, bestehend aus n Komponenten reeller Zahlen, wird auf eine Menge von i reellen Zahlen, die größer als 0 sind, abgebildet. Dabei bildet die Summe aller k stets 1; i ist die Anzahl aller Klassen und k_i der Wahrscheinlichkeitswert für die i-te Klasse.

In der obenstehenden Erklärung wird also jede Eingabe auf genau eine Klasse abgebildet. Das würde für die Fehleranalyse bedeuten, dass jede Eingabe genau ein syntaktisches Element des Quellcodes ist. Schließlich sollen Fehler in verschiedenen Bereichen des Quellcodes identifiziert werden können. Das bedeutet im Umkehrschluss, dass jede Eingabe genau ein zu klassifizierendes Element ist. Ferner soll die Klassifizierung auf das Element im Quellcode zurückgeführt werden können. Wie genau die Eingabe bzw. Ausgabedaten des Modells aussehen, sodass die vorliegenden Kriterien erfüllt werden, ist Bestandteil der Kapitel 2 und 6 und wird an dieser Stelle nicht näher erläutert.

4.2 Die Erfahrung: Dokumentierte Fehler

ML-Algorithmen werden Erfahrungen ausgesetzt, auf dessen Basis die Parameter des Modells angepasst werden [22]. Dieser Prozess wird als Lernen bezeichnet. Der Datensatz, in dem die Erfahrung verzeichnet sind, wird Lerndatensatz genannt. Die verwendbaren ML-Algorithmen sind abhängig von der Art der Erfahrungen. Dabei können alle ML-Algorithmen in die grundlegenden Kategorien *überwachtes*, *unüberwachtes* und *bestärkendes* Lernen unterteilt werden [21]. Wie eingangs erwähnt, geht es bei allen ML-Algorithmen darum, die Funktion $f(x) = y$ zu erlernen. Bei überwachten Lernverfahren muss im Lerndatensatz zu jedem x das entsprechende y verzeichnet sein. Während des

Lernens werden Ein- und Ausgabenpaare beobachtet und die Modellparameter so angepasst, dass die Eingabe x auf die entsprechende Ausgabe y abgebildet wird. Beim unüberwachten Lernen besteht der Lerndatensatz nur aus x-Werten [21]. Eine direkte Bewertung durch das Maß P ist also nicht möglich. Vielmehr werden Ähnlichkeiten in den Eingabedaten gesucht, um somit Anomalien zu erkennen oder die Daten zu Clustern. Mit dem bestärkenden Lernen wird in der Regel ein Agent gelernt, z.B. ein Roboter, der durch bestärkendes Feedback die optimale Strategie zur Lösung eines Problems lernt. Auf die Eingabe x folgt eine entsprechende Ausgabe y die bewertet wird und anschließend zu einer Anpassung der Modellparameter führt. Die Art der Erfahrung und der Typ der Aufgabe definieren also die Auswahl des Lernverfahrens.

Diese Arbeit basiert auf der Annahme, dass auf Filehostern wie GitHub große Datenmengen vorhanden sind, aus denen Beispiele für Fehler im Programmcode entnommen werden können. Im Falle dieser Arbeit ist ein Beispiel die Repräsentation eines Codeabschnitts und die Klassen der im Beispiel vorhandenen Fehler. Die Beispiele bestehen also aus Merkmalen x in Form von Codefragmenten und der entsprechenden Klassen y. Demnach ist die Verwendung überwachter Lernalgorithmen möglich.

4.3 Das Maß

Das Maß ist ein Indikator für die Performance des Modells. Es gibt also Aufschluss darüber, wie gut die Identifikation der Fehler gelernt wurde. Die meisten Metriken basieren auf den Kennzahlen *True-Positive* (TP), *False-Positive* (FP), *True-Negative* (TN) und *False-Negative* (FN) [62]. Die Begriffe *Positive* und *Negative* beschreiben dabei, ob die Eingabe der entsprechenden Klasse zugeordnet wurde (Positive) oder nicht (Negative). Mit den Begriffen *True* und *False* wird angegeben, ob es sich um eine korrekte Klassifizierung handelt (True) oder nicht (False). Entsprechende Metriken werden also pro Klasse ausgerechnet. Zur Ermittlung einer einzelnen Performance-Kennzahl für das Modell müssen die Werte aggregiert werden. Zu diesem Zweck kann der *macro-* oder *micro-*Durchschnitt berechnet werden [62]. Der macro-Durchschnitt wird auf Klassen-Basis berechnet. Das bedeutet, die Metrik wird für jede Klasse einzeln berechnet, anschließend aufsummiert und durch die Anzahl aller Klassen geteilt. Bei dieser Art der Aggregierung haben Klassen mit einer geringen Anzahl an Beispielen den gleichen Einfluss wie Klassen mit einer großen Anzahl an Beispielen [71]. Der micro-Durchschnitt wird auf Basis aller Beispiele berechnet. Zu diesem Zweck wird die Anzahl der TPs, TNs, FPs und FNs aller

Klassen kumuliert und anschließend zur Berechnung des Metrik verwendet. Auf diese Weise bekommen die Klassen mit den meisten Beispielen den größten Einfluss auf das Endergebnis [71].

$$recall = \frac{TP}{TP + FN} \tag{4.1}$$

$$precision = \frac{TP}{TP + FP} \tag{4.2}$$

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \tag{4.3}$$

Jede Metrik beantwortet eine Frage in Hinblick auf die Performance des Modells. Ziel des Modells ist eine möglichst präzise Identifikation von Fehlern. Das bedeutet, dass das Modell im Idealfall für jeden Fehlertyp auch alle Fehler als solche erkennt. Diese Eigenschaft kann durch die *recall* Metrik berechnet werden. Sie ist in der Formel 4.1 abgebildet. Je höher der Wert ≤ 1 desto mehr Fehler wurden identifiziert. Das Problem dieser Metrik besteht darin, dass ein Modell, in dem jedes Beispiel als Positive klassifiziert wird, einen hohen Wert erzielt. Ein solches Modell ist nicht hilfreich, denn eine hohe Anzahl von FPs verringert die Glaubwürdigkeit der Fehleranalyse. Je mehr FPs das Modell generiert, desto wahrscheinlicher ist es, dass ein vorhergesagter Fehler gar keiner ist. Um zu bewerten, wie viele der vorhergesagten Fehler tatsächlich Fehler sind, wird die *precision* Metrik verwendet. Sie ist in der Formel 4.2 abgebildet. Je höher der Wert ≤ 1 desto mehr identifizierte Fehler sind auch tatsächlich Fehler. Durch die Kombination der beiden Metriken, dem F_1-Score, entsteht eine Metrik, die Aufschluss darüber gibt, wie viele der Fehler identifiziert wurden und wie viele der identifizierten Fehler auch tatsächlich Fehler sind. Der F_1-Score bildet sich durch das harmonische Mittel von precision und recall. Die Berechnung des F_1-Scores ist in Formel 4.3 abgebildet. Der Wert des F_1-Score befindet sich auf einer Skala von 0 bis 1, wobei ein höherer Wert einer besseren Performance entspricht.

5 Vom Programmcode lernen

Nachdem die Fehleranalyse als ML-Problem definiert wurde, sollen relevante Methoden zum Lernen von Programmen untersucht werden. Im Grunde besteht der Lernprozess daraus, auf Basis einer begrenzten Anzahl von Features ein Modell zu lernen, das eine Wahrscheinlichkeitsverteilung über relevante Codeeigenschaften generiert. Features sind aus den Rohdaten ermittelte numerische Eigenschaften, die bei der Lösung des gegebenen Problems hilfreich sein sollen [74]. Im Idealfall repräsentieren die Features nur das zugrundeliegende Problem und reduzieren somit Redundanz und Rauschen der Daten [75]. Hierdurch können die Features einfacher vom Modell verarbeitet werden und resultieren in einer besseren Performance. Gute Features führen außerdem zu mehr Flexibilität bei der Modellauswahl und schließlich zu einfacheren Modellen. Eine Auswahl suboptimaler Features führt jedoch dazu, dass das Modell das Signal der Daten schlecht oder gar nicht erkennen kann. Folglich leidet die Performance und das Modell wird komplizierter. Eine suboptimale Auswahl kann aus zu vielen, zu wenigen oder falschen Features resultieren. Infolgedessen kann es zu unzureichenden Informationen oder einem erhöhten Rauschen und zu viel Redundanz kommen. Die Auswahl der Features hängt stark von den Rohdaten, der Aufgabe und schlussendlich dem Modell ab.

Aussagekräftige Features bilden also einen Grundpfeiler für die Performance eines Modells, dementsprechend auch beim Lernen von Programmcode. Programme bestehen jedoch aus Textdateien und liegen nicht in numerischer Form vor. Wie bereits in Kapitel 2 erwähnt, wird der Quellcode eines Programms durch Entwickler:innen in einer Programmiersprache verfasst. Programmiersprachen haben, genau wie natürliche Sprachen, grammatikalische Regeln, eine Syntax und semantische Bedeutungen. Nach Knuth [37] ist der Quellcode außerdem ein Kommunikationsmittel zwischen Entwickler:innen. Demnach ist der Quellcode eine an weitere Entwickler:innen gerichtete Beschreibung dessen, was die Autor:in den Computer anweist zu tun. Deshalb liegt der Schluss nahe, die Feature-Engineering Methoden der natürlichen Sprachverarbeitung (NLP) zu verwenden. Hindle et. al. [29] unterstreichen diesen Gedanken mit einer Hypothese. Demnach sind Programmiersprachen und natürliche Sprachen, trotz theoretischer Komplexität und großer

Ausdrucksstärke, in der Praxis eher einfach und repetitiv. Das bedeutet, dass Quellcode nützliche und vorhersagbare statistische Eigenschaften besitzt, die durch ein statistisches Sprachmodell abgebildet werden können. Diese Hypothese wird in der Arbeit [29] durch die Implementation einer Codevervollständigung demonstriert. Während die Verarbeitung von Quellcode in einfacher Textform möglicherweise Aufschluss über syntaktische oder lexikalische Fehler bieten kann, so sind semantische Fehler weitaus schwieriger zu ermitteln. Nicht ohne Grund verwenden statische Codeanalyseprogramme eine für die Analyse optimierte Repräsentation des Quellcodes. Auch weiterführende Arbeiten auf diesem Gebiet führen an, dass die Verarbeitung von Quellcode in einfacher Textform wichtige syntaktische Eigenschaften und explizite strukturelle Informationen unterschlagen [73][47]. Ferner trennen semantisch und syntaktisch zusammenhängende Elemente im Quellcode oft mehrere Zeilen [3]. Das erschwert Modellen, die Quellcode wie einen einfachen Text behandeln, die Erfassung dieser Zusammenhänge. Schlussendlich sind Programmiersprachen formale Sprachen, die entscheidende Unterschiede zur natürlichen Sprache aufweisen [3]. Quellcode ist beispielsweise ausführbar und bietet deshalb neben Quellcode und AST außerdem einen Kontroll- und Datenfluss. Diese können, wie bereits in Abschnitt 3.1 erläutert, in verschiedenen Repräsentationen dargestellt werden. Diese strukturierten Darstellungen bieten, durch die Maskierung einiger Elemente des Quellcodes und die Anreicherung durch Kontextinformationen, einzigartige Sichten auf den Quellcode und dessen Struktur. Hierdurch werden die syntaktische und semantische Informationen des Quellcodes in verdichteter Form verfügbar gemacht.

An dieser Stelle sei erwähnt, dass Quellcodeabstraktionen, genau wie der Quellcode in Textform, keine Features sind und ohne eine entsprechende Kodierung nicht von ML-Algorithmen verwendet werden können. Letztendlich sind die gewünschten Features ausschlaggebend für die Auswahl der Quellcodeabstraktion. Was *sinnvolle* Features sind, hängt davon ab, welche Eigenschaften des Codes analysiert werden sollen.

Die Literatur im Bereich des ML auf Programmcode zeichnete in den vergangenen Jahren ein einheitliches Bild in Bezug auf die Ermittlung von Features aus Programmcode [3]. So werden manuell erstellte Features als ineffizient und unzureichend beschrieben [47]. Einerseits müssen manuell erstellte Features für jeden Fehler in einem zeitintensiven Prozess vor dem Lernen des Modells erarbeitet werden. Andererseits mindern sie die Flexibilität in Bezug auf die Weiterentwicklung des Modells. Für jeden neuen Fehler muss die Feature-Auswahl angepasst werden. Denn möglicherweise besitzen die zuvor gewählten Features keine Informationen über die zu erweiternden Fehlertypen. Nicht zuletzt besteht die Gefahr, dass die manuelle Auswahl der Features nicht ausreichend Informationen ent-

hält, um das Signal der Daten in Bezug auf das Problem zu lernen. Eine erfolgversprechende Alternative zum manuellen Feature-Engineering ist das *Feature Learning* bzw. *Representation Learning* [74]. Grundsätzlich kann das Feature-Learning auch zusammen mit dem Prognose-Algorithmus realisiert werden. In diesem Fall spricht man vom *End-To-End-Learning* [74]. In der Regel werden diese Features durch mehrere Schichten eines neuronalen Netzes automatisch gelernt [47]. Die Eingabe in ein solches Netz wird *Intermediate Representation (IR)* genannt. Die Features sind die Ausgabe des Netzes (Feature-Learning) oder befinden sich in einer Zwischenschicht des Netzes (End-To-End-Learning) [74]. Im Prinzip sind beim Lernen von Features also zwei Schritte von grundlegender Bedeutung: das Erstellen einer IR und die Wahl des Feature-Lernalgorithmus. Die IR sollte in einer Form vorliegen, die den Lernprozess unterstützt und wichtige Features leicht erkennen lässt. Der Feature-Lernalgorithmus sollte die Struktur und Kapazität besitzen, aussagekräftige Features zu erkennen.

Da es sich um ML-Algorithmen handelt, gilt auch hier wieder die Unterscheidung zwischen überwachten und unüberwachten Verfahren [74]. Beim unüberwachten Feature-Learning werden die Features labelunabhängig gelernt. Das kann zum Beispiel ein Autoencoder sein, der den Informationsgehalt einer Eingabe in eine geringere Anzahl von Features kodiert. Diese Features sind effizient und können für weiterführende ML-Aufgabe verwendet werden [47][73][12]. Das überwachte Feature-Learning verwendet einen gelabelten Datensatz und optimiert die Features auf Basis eines Kriteriums. Die resultierenden Features beider Verfahren unterscheiden sich in den kodierten Informationen. Unüberwacht gelernte Features kodieren Informationen über die generelle Struktur oder Semantik der Eingabe, während überwacht gelernte Features spezielle Informationen in Bezug auf das korrespondierende Label enthalten.

Dass die Verwendung des Programmcodes in reiner Textform nicht optimal ist, wurde zuvor bereits erläutert. Ferner wurde in Abschnitt 3.1 aufgezeigt, dass sich viele Aspekte eines Programms in Form von Graphen repräsentieren lassen. Durch Graphen lassen sich komplexe Strukturen und Abhängigkeiten zwischen den einzelnen Komponenten eines Programms in abstrahierter Form konsolidiert darstellen. Da sich die Fehler einer Software in der Regel durch das Zusammenspiel verschiedener Komponenten eines Programms ergeben, bietet die Darstellung von Quellcode als Graph eine gute Grundlage, um syntaktische und semantische Informationen zu kombinieren. Die Verwendung eines Graphen würde entsprechend in einer Klassifizierung des Graphen oder einzelner Knoten resultieren.

Traditionell wurden die Features eines Graphen oder der Knoten eines Graphen vor der Anwendung des ML-Algorithmus manuell definiert [25]. Dazu wurden verschiedene Statistiken des Graphen wie z.B. Node-Degree, Node-Centrality extrahiert oder Kernel-Methoden auf Graphen angewandt. Kernel Methoden sind ML-Algorithmen, die durch die Anwendung eines Kernels auf Datenpunktpaaren lernen. Durch die Anwendung dieser Methoden können sich die Features jedoch nicht während des Lernens anpassen und müssen entsprechend vor dem Lernvorgang gewählt werden. Die Nachteile von manuell gewählten Features wurden bereits erläutert. Weiterhin verwenden diese Verfahren ausschließlich die Struktur des Graphen und keine Attribute der Knoten oder Kanten. Insofern sollte ein Verfahren in dieser Arbeit gewählt werden, das flexibel Features lernen kann und sowohl strukturelle als auch semantische Eigenschaften erfasst.

Die Verwendung von Attributen in Form von *Node-Features* ermöglichen das Erfassen von semantischen Informationen des Graphen [25]. Node-Features sind z.B. die Knoten-Typen oder Bezeichner, die in Abbildung 3.2 in den Knoten stehen. Also beliebige Informationen die zu einem Knoten gehören. Auch diese Informationen müssen in numerische Form vorliegen, um von einem ML-Algorithmus verwendet zu werden. Deshalb bedient man sich in diesem Fall, insofern das Attribut aus einem Text besteht, oft bei den Methoden der NLP und erstellt ein sogenanntes Word-Embedding.

5.1 Tree-Based Neural Networks

Klassische neuronale Netze können zwar durch verschiedene Kodierungsmethoden auf Graphen angewendet werden, sind jedoch nicht dafür optimiert, die Strukturen eines Graphen zu erfassen. Mit klassischen neuronalen Netzen sind z.B. Multi-Layer-Perceptrons (MLP), Convolutional-Neural-Networks (CNN) und Recurrent-Neural-Networks (RNN) gemeint. MLPs sind neuronale Netze die aus mehreren linearen Schichten aufgebaut sind [54]. Sie eigenen sich für Daten, deren Datenpunkte identisch und unabhängig verteilt sind. Bei Bildern und Texten ist das jedoch nicht der Fall. Aus diesem Grund wurden RNNs für sequenzielle Eingaben (z.B. Text) und CNNs (z.B. Bilder) für räumliche Eingaben entwickelt. RNNs lernen einen verdeckten Zustand der Informationen über alle Elemente einer Sequenz kodiert. Bei CNNs wird ein zuvor definierter Filter auf die Eingabe angewandt, sodass räumliche Strukturen transformationsinvarant erkannt werden können.

Im Bereich der NLP sind Bäume eine gängige Form der Repräsentation von Sätzen. Aus diesem Grund haben sich in den vergangenen Jahren einige Varianten der klassischen künstlichen neuronalen Netze entwickelt, die speziell für die Verarbeitung von Bäumen angepasst wurden. Eine Verallgemeinerung der RNNs für Baumstrukturen ist das Tree-LSTM [64]. Im Gegensatz zum ursprünglichen LSTM wird der versteckte Zustand durch die versteckten Zustände der Kinder-Knoten berechnet. Die so erstellte Kodierung enthält neben der Semantik der Knoten ebenfalls topologische Informationen des Baums. Tai et. al. [64] konnten durch ein Experiment nachweisen, dass das Tree-LSTM die Semantik eines Satzes besser erfassen kann als ein herkömmliches LSTM. Das Experiment bestand darin, den semantischen Gehalt zweier Sätze zu vergleichen. In diesem Fall hat allerdings jeder Knoten den gleichen Einfluss auf die resultierende Vektor-Kodierung des Baums. Das entspricht in vielen Szenarien jedoch nicht der Realität, sodass Ahmed et. al. [1] einen Aufmerksamkeitsmechanismus für Tree-LSTMS vorschlagen. In Vergleich zum normalen Tree-LSTM haben die Tree-LSTMs mit Aufmerksamkeitsmechanismus bessere Ergebnisse bei der selben Aufgabe erzielt. Das Tree-CNN ist die Verallgemeinerung von CNNS auf Baumstrukturen. Die Features des Baumes werde durch einen Filter mit fester Größe über die Knoten extrahiert [47]. Der Filter wird dabei von unten nach oben, also bei den Blättern beginnend, angewandt. Das Resultat wird durch verschiedene Pooling-Schichten zu einem Vektor aggregiert.

Motivation und Hintergrund dieser Anpassungen sind NLP-Aufgaben. Trotzdem bieten sie durch die Fähigkeit, die semantische Bedeutung eines Baums zu erfassen, einen validen Ansatz für die Kodierung eines ASTs. Der AST eines Programms ist in der Regel jedoch wesentlich größer als der Syntaxbaum eines Satzes [3]. Je größer der Baum, desto höher ist die Gefahr des Vanishing-Gradient-Problems, sodass wichtige Kontextinformationen verloren gehen können [73]. Um diesem Problem entgegenzuwirken haben Zhang et. al. [73] ein neuronales Netz speziell für Programm-ASTs (ASTNN) entwickelt. In ASTNN wird der AST eines Programms in Statement-Trees aufgeteilt. Jeder Statement-Tree besteht aus einer Sequenz von Statements, die durch ein RNN in einen Statement-Vektor kodiert werden. Im Anschluss werden alle Statement-Vektoren durch ein Bidirektionales-RNN in einen Programmvektor kodiert. Auf diese Art und Weise werden lexikalische Informationen und die Syntax der Statements erfasst.

Die Nachteile dieser Methode liegen darin, dass ausschließlich Baumstrukturen verarbeitet werden können. Daten- oder Kontrollflussgraphen bieten wichtige Informationen über die Abläufe des Programms, können jedoch nicht verwendet werden. Der AST kann ein beliebig feingranuliertes Codefragment repräsentieren und der somit erstellte Vektor

beinhaltet Informationen über genau dieses Codefragment. Viele Fehler setzten sich jedoch aus einzelnen wenigen Elementen zusammen, die im AST weit verteilt sind. Zum Beispiel eine nicht verwendete Feldvariable, deren Referenzen in nahezu jedem Teil des Baums vorkommen können. Um dem Modell alle relevanten Informationen zur Verfügung zu stellen, muss ein großer AST verwendet werden, der viele irrelevante Informationen enthält. Eine solche Methode ist für die Klassifizierung auf Klassen- oder Dateiebene sinnvoll. Für eine feingranulare Klassifizierung, die ein beliebiges Element im Code analysiert, ist sie jedoch nicht zu gebrauchen.

5.2 Shallow-Embeddings

Eine weitere Möglichkeit die Features eines Graphen zu lernen, sind sogenannte *Shallow-Embeddings* [18][15][25]. Im Grunde bedeutet es, die Knoten auf Basis ihrer Attribute, der Position im Graphen und der Nachbarn in niedrigdimensionale Vektoren (Embedding) zu kodieren. Je nach Lernverfahren enthalten entsprechende Vektoren wichtige Eigenschaften des Graphen verteilt über die Vektorkomponenten. Bei unüberwachten Verfahren sind es Informationen über die Topologie und Semantik der Knoten, bei überwachten Verfahren sind es aufgabenspezifische Informationen. Für jeden Knoten des Graphen, der im Lerndatensatz vorliegt, wird ein Embedding erstellt. Im Anschluss können diese bei Bedarf zu einem Graph-Embedding aggregiert werden. Im Laufe der Jahre wurden verschiedene Methoden entwickelt, um entsprechende Embeddings durch überwachte oder unüberwachte Verfahren zu lernen. In der Regel werden diese Embeddings so gelernt, dass sich *gleichende* Knoten im Vektorraum nah beieinander liegen. Der Unterschied zwischen den Methoden besteht im Grunde darin, inwiefern die *Gleichheit* definiert wird [14].

Zwei weitverbreitete unüberwachte Methoden sind DeepWalk [52] und node2vec [24]. Sie basieren auf dem Skip-Gram Modell [44]. Das Skip-Gram Modell lernt Embeddings für jedes Wort einer Sequenz von Wörtern, indem es das Zielwort als Eingabe und die restlichen Wörter der Sequenz als erwartete Ausgabe verwendet. Das Embedding ist die versteckte Schicht des Skip-Gram Modells. Es wird durch die umliegenden Wörter, also dem Kontext des Satzes, definiert. In Bezug auf Graphen entspricht der Kontext den Nachbarknoten. DeepWalk führt eine feste Anzahl von *Random Walks* über den Graphen aus, um Pfade zu erstellen. Dabei startet jeder Random Walk bei dem korrespondierenden Knoten und überquert eine feste Anzahl von Knoten in zufälliger Reihenfolge. Analog zu einem Satz, bildet jeder überquerte Knoten ein Wort. Ein Pfad ist dementsprechend

ein Satz, von dem jedes Wort durch das Skip-Gram Modell in Vektoren kodiert wird. Insofern gleichen sich Knoten, die eine ähnliche Nachbarschaft haben. Durch node2vec wird dieser Ansatz erweitert und führt zwei Hyperparameter ein, um die Random-Walks so zu beeinflussen, dass neben lokalen Strukturen im Graph auch globale Strukturen erfasst werden. Die Random-Walk Variante wurde in verschiedenen Ausprägungen erweitert, um unterschiedliche Eigenschaften der Embeddings zu optimieren.

Im Bereich des ML auf Programmcodes haben sich die Methoden des Shallow-Embeddings bereits etabliert. Die Arbeiten *Code2Vec* [6] und *Func2Vec* [19] lernen entsprechend Vektoren, die Programmelemente repräsentieren. Bei Code2Vec wird als Basis der AST verwendet. Hier werden jedoch keine Random-Walks verwendet. Die Pfade werden vielmehr nach einem festen Schema extrahiert. Jeder Pfad, der mit einem Blatt startet und endet und über den zu vektorisierenden Knoten führt, definiert dessen Kontext [5]. Diese Pfade werden durch ein Embedding vektorisiert und anschließend mit einem Aufmerksamkeitsmechanismus aggregiert. Als Label für die entsprechenden Vektoren werden die Variablen-, Methoden- oder Klassenbezeichner verwendet. Bezeichner von Code-Elementen enthalten wichtige Informationen über die beabsichtigte Semantik des Codes. Sie sind eine Hauptquelle für konzeptuelle Informationen [13]. Aus diesem Grund enthalten die gelernten Vektoren semantische und syntaktische Informationen über das repräsentierte Codeelement. Dieses Schema kann jedoch auch für beliebige Label angewandt werden. In func2vec wird die Random-Walk-Methode auf einem CFG verwendet. Die Label der Knoten entsprechen einer Bezeichnung der Programmelemente, die sie repräsentieren. In beiden vorgestellten Verfahren haben die Autor:innen den semantischen Gehalt der Vektoren nachgewiesen, indem sie auf dessen Basis Synonymfunktionen entdeckt haben. Synonymfunktionen sind Funktionen, die eine ähnliche oder gleiche Aufgaben im Programm übernehmen.

Shallow-Embeddings bieten also eine Möglichkeit, Features aus Quellcode zu lernen, die Programmeigenschaften repräsentieren und im Anschluss für Prognosen auf dem Quellcode verwendet werden können. Ein großes Problem dieser Methoden besteht darin, dass diese Embeddings nur für Pfade und Knoten generiert werden können, die während des Trainings vorhanden gewesen sind [25]. Es handelt sich um ein transduktives Lernverfahren. In der vorausgehenden Arbeit, dem Hauptprojekt [60], ist aufgefallen, dass aus diesem Grund über 50 % der Testdaten nicht verwendet werden konnten. Entsprechend benötigen diese Methoden einen sehr großen Lerndatensatz, in dem möglichst viele Pfade vorliegen. Einen generelleren Ansatz bieten Graph Neural Networks (GNN), die mehr auf

der Struktur und den Attributen des Graphen basieren. Die Verfahren sind induktiv und können auf ungesehenen Graphen und Knoten angewandt werden.

5.3 Graph Neural Networks

Graph Neural Networks (GNNs) sind neuronale Netzwerke, die direkt auf Graphen angewendet werden können. Ein Graph G ist eine Datenstruktur bestehend aus einer Menge von Knoten V und Kanten E, also $G = (V, E)$. Er bietet die Möglichkeit komplexe Abhängigkeiten zwischen verschiedenen Objekten zu modellieren. Graphen existieren jedoch nicht im euklidischen Raum und können somit nicht durch Koordinaten dargestellt werden. Ferner haben sie variable Formen und Strukturen. Diese Eigenschaften erschweren die Verarbeitung von Graphen in herkömmlichen neuronalen Netzen wie dem MLP, CNN oder RNN [26].

GNNs basieren auf der Idee, dass der Knoten eines Graphs natürlich durch seine Features und Nachbarknoten definiert wird [76]. Die Grundlage aller GNNs ist das sogenannte *neural message passing* [26]. Dabei sendet jeder Knoten v über eine bestimmte Anzahl von Iterationen k Nachrichten an seine Nachbarn $\{u, \forall u \in N(v)\}$. Jeder Knoten **aggregiert** die Nachrichten seiner Nachbarn $m_{N(v)} = \{h_u, \forall u \in N(v)\}$ und **aktualisiert** damit seinen *hidden embedding* h_v. Die Nachrichten, die verschickt werden, sind die hidden embeddings der jeweiligen Knoten. Initial wird das hidden embedding durch die Node-Features x_v definiert. Somit ist die Dimension von h_v mindestens so groß wie die von x_v. Durch das sogenannte *padding* (Auffüllen mit Nullen) kann die Dimension von h_v initial vergrößert werden. Damit lässt sich die Kapazität des Netzes erhöhen. Die Aggregierung und Aktualisierung wird durch Differentialfunktionen definiert und während des Lernens angepasst. Auf diese Weise werden durch jede Iteration die Informationen der Knoten weiter im Netz verteilt. Hierbei spricht man von einer Informations-Diffusion. In der Abbildung 5.1 ist die Informationsverteilung nach 3 Iterationen skizziert. Es ist ersichtlich, dass mit jeder Iteration die Informationen eines Knotens weiter im Netz verteilt werden. Im neuronalen Netz werden die Iterationen

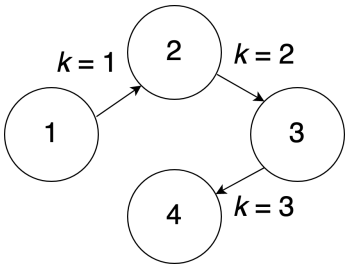

Abbildung 5.1: Skizze der Informationsverteilung durch das *neural message passing*

k durch k Schichten von Matrixmultiplikationen realisiert. Auf der finalen Schicht ergeben sich die Knotenrepräsentationen $e_v = h_v, \forall v \in V$ aus den hidden embeddings der einzelnen Knoten. Entsprechende Vektoren e_v enthalten strukturelle Informationen über den Graphen. Am Beispiel aus Abbildung 5.1 enthält Knoten 4 z.B. nach der dritten Iteration Informationen über die Nachbarschaft und Verbindungen von Knoten 1, also auch über den Grad des ersten Knotens. Aufgrund der Einbeziehung von Node-Features werden mit jeder Iteration ebenfalls semantische Informationen verteilt [15]. Am Beispiel des ASTs kodieren die Blätter z.B., dass sie Blätter sind und ob sie sich beispielsweise innerhalb einer Variablendeklaration befinden. Diese Informationen bieten eine wichtige Grundlage zum Ermitteln von Fehlern im Programmcode. Schließlich erstrecken sich die meisten Fehler über mehrere Komponenten eines Quellcodes.

Die gelernten Knotenrepräsentationen können als Eingabe von weiterführenden ML-Algorithmen verwendet werden oder im Sinne des End-To-End Learning durch das Hinzufügen von Schichten zur Klassifikation verwendet werden. Eine weitere Möglichkeit ist das Hinzufügen von Pooling-Layern, um die Node-Embeddings zu einem Graph-Embedding zu aggregieren.

Seit der Einführung des ersten GNNs [56] wurden verschiedene Varianten entwickelt, um unter anderem spatiale-, spektrale und temporale Graphen zu verarbeiten [26]. Die Variationen unterscheiden sich maßgeblich in der Art, wie aggregiert und aktualisiert wird. Varianten der Aggregation umfassen unter anderem das Normalisieren der Nachbarschaft und das Verwenden von Aufmerksamkeitsmechanismen. Wenn die Aggregation der Nachrichten durch eine Addition definiert ist, dann wird h_v stark von der Anzahl der Nachbarknoten beeinflusst. Durch die Nachbarschafts-Normalisierung kann dieses Verhalten verhindert werden. Hierdurch wird wiederum die Möglichkeit, strukturelle Informationen zu kodieren, verhindert. Eine Normalisierung ist nur dann hilfreich, wenn die Node-Features wichtiger sind als die Topologie des Graphen. Durch den Aufmerksamkeitsmechanismus kann jedem Nachbarknoten ein individueller und lernbarer Gewichtungsskalar zugewiesen werden, sodass wichtige Nachbarn einen größeren Einfluss auf h_v haben. Varianten der Aktualisierung fassen unter anderem die Einführung einer Gating-Mechanik, Konkatenationsmethoden oder Jumping-Knowledge-Connections. Dabei wirken die Aktualisierungsvariationen maßgeblich dem *oversmoothing* entgegen. Das tritt auf, wenn die knotenspezifischen Informationen nach einigen Iterationen anfangen zu verschwinden. Also wenn die aggregierten Nachrichten der Nachbarn einen zu großen Einfluss auf h_v haben.

Durch die Verwendung von GNNs werden entsprechend nicht die Knoten- oder Graphenrepräsentationen an sich gelernt, sondern die Aggregations- und Aktualisierungsfunktion. Das ist von Vorteil, weil Graphen, die nicht in den Lerndaten vorhanden gewesen sind, für eine Prognose verwendet werden können. Das bezieht sich jedoch nicht auf die Node-Features. Diese müssen, wie bei fast allen anderen vorgestellten Methoden, in dem Lernvokabular vorhanden gewesen sein. Das GNN lernt auf welche Art und Weise die Node-Features durch den Graphen propagiert werden, damit im Anschluss jede Knotenrepräsentation Informationen über seine Nachbarschaft beinhaltet. Der Vorteil gegenüber den Tree-Based-Methoden ist die Flexibilität der Graph-Struktur. So können nicht nur Baumstrukturen, sondern beliebige Graph-Strukturen verwendet werden. Das ermöglicht die Erweiterung des ASTs durch Daten- und Kontrollflusskanten oder die Verwendung von anderen Graphrepräsentationen des Programms. Im Vergleich zu den Shallow-Embeddings haben GNNs den Vorteil, dass sie auf beliebige Graphen angewendet werden können, die nicht in den Lerndaten vorhanden gewesen sind. Die einzige Limitierung in dieser Hinsicht sind die Node-Features, die aus den Lerndaten bekannt sein müssen.

Weil sich viele Programmaspekte gut durch Graphen abbilden lassen und die GNNs eine flexible Möglichkeit bieten, die Repräsentation von Graphen zu lernen, wurden in der Literatur bereits verschiedene GNNs für das Lernen von Features aus Programmen verwendet. Allamanis et. al. [4] erweitern einen AST um bidirektionale Datenflusskanten und lernen mit einem Gated-Graph-Neural-Network (GGNN) dessen Knotenrepräsentationen. Die Knotenrepräsentationen werden anschließend verwendet, um Variablen-Fehlnutzungen zu identifizieren und Namensvorschläge für Variablen zu generieren. Weil die Kanten bidirektional gestaltet wurden, verbreiten sich die Informationen schneller im Netz. Laut Allamanis et. al. [4] haben die Node-Embeddings hierdurch eine größere Ausdrucksstärke. Wang et al [68] verwenden den gleichen Ansatz, um semantische Code-Clones zu identifizieren. Sie weisen eine höhere Genauigkeit im Vergleich mit Code-Clone-Identifikatoren, die auf ASTNN basieren, nach. In der Literatur sind noch weitere Beispiele von erfolgreichen Anwendungen von GNNs auf Quellcode vorhanden.

Aufgrund der zuvor aufgezählten Vorteile scheinen GNNs eine optimale Wahl für die Identifizierung verschiedener Fehler im Quellcode zu sein. Durch das induktive Lernen kann das Gelernte auf beliebige neue Eingangsdaten angewendet werden. Die Klassifizierung einzelner Knoten bietet eine beliebige Flexibilität bei der Fehleridentifikation. So kann z.B. bei der Klassifizierung der Knoten eines ASTs jedes Ergebnis direkt auf die syntaktische Einheit zurückgeführt werden.

6 Konzept

In diesem Kapitel wird das Konzept zum Lernen einer Fehleranalyse vorgestellt. Zu diesem Zweck sind in den vorherigen Kapiteln bereits geeignete Methoden für die Datenvorbereitung und das Lernen von Programmcodes aufgezählt. Dabei hat sich die Verwendung von Quellcodes in Form eines einfachen Texts als suboptimal herausgestellt. Ferner wird Gebrauch von Compiler-Zwischendarstellungen wie dem AST oder Daten- und Kontrollflussgraphen als sinnvoll erachtet. Für das Lernen vom Programmcode haben sich GNNs als geeignete Methode hervorgehoben. Sie ermöglichen das Erzeugen von Graph- und Knotenrepräsentationen durch die Propagierung von Knoteninformationen über gelernte Aggregations- und Aktualisierungsmethoden. Auf den Knotenrepräsentationen kann im Anschluss eine Klassifizierung in die entsprechenden Fehlerklassen getätigt werden. Weil diese Prozesse nacheinander ausgeführt werden, spricht man von einer ML-Pipeline. In Abbildung 6.1 sind die wichtigsten Prozessschritte dieser Pipeline skizziert. Dabei ist jeder Prozessschritt durch eine rechteckige Box dargestellt. Die Reihenfolge wird durch die Pfeile gekennzeichnet und erstreckt sich von oben links nach unten rechts. Entsprechend ist vor und hinter jeder Box eine Skizze der Ein- bzw. Ausgabe abgebildet. Der erste Prozessschritt ist die Datenverarbeitung. Dabei wird der Java-Quellcode in eine möglichst aussagekräftige Form (IR) transformiert. Im nachfolgendem Abschnitt 6.1 *Erstellung der Graphen* wird die Auswahl der Kanten und Knoten des Prozessschritts erläutert. Anschließend folgt in Abschnitt 6.2 *Node-Features* eine Auswahl der Node-Features, also den inhaltlichen Informationen eines Knotens. Das beinhaltet darüber hinaus den nachfolgenden Prozessschritt der Kodierung. Dieser beschreibt die Kodierung der ausgewählten Node-Features in eine numerische Form. Schlussendlich folgt der Abschnitt 6.3 *Das Netz*, in dem das künstliche neuronale Netz bestehend aus den Schritten Kodierung, GNN und Klassifizierung aufgebaut wird.

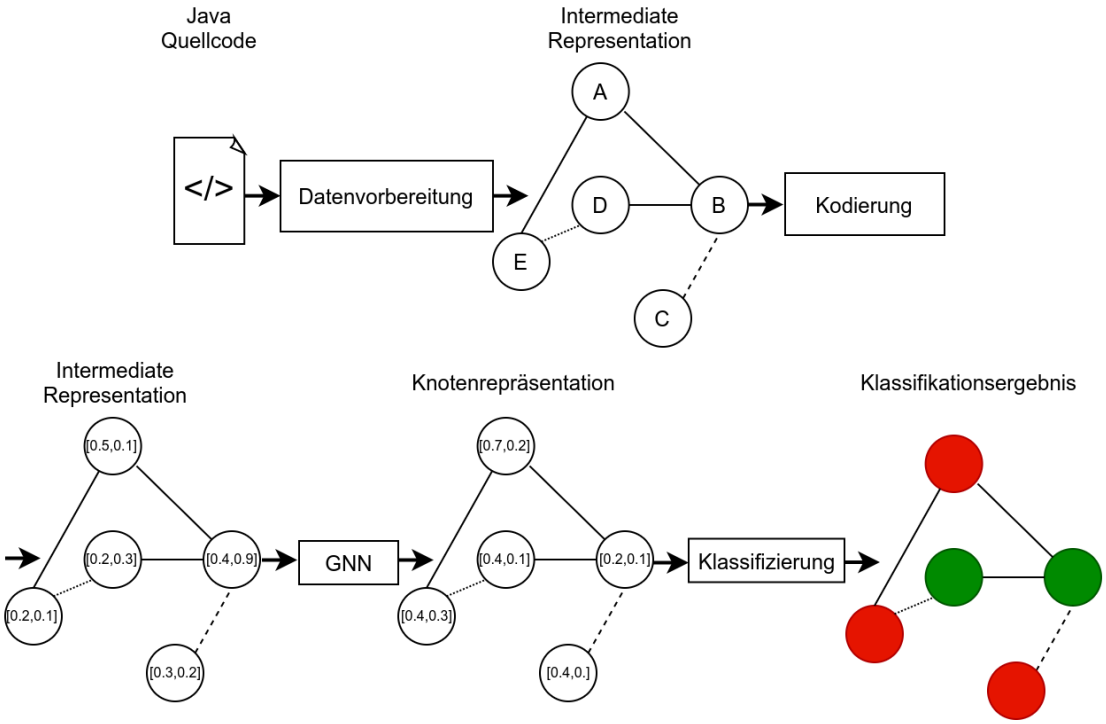

Abbildung 6.1: Pipeline für die Klassifizierung von Fehlern im Java-Quellcode

6.1 Erstellung der Graphen

In diesem Abschnitt wird die Struktur der IR festgelegt. Die IR beinhaltet die Informationen, auf dessen Basis das Modell das Signal der Daten erlernen soll. Deshalb ist es sinnvoll, irrelevante Informationen zu entfernen um somit relevante Informationen hervorzuheben. Folglich wird durch eine gute Datenvorbereitung das Rauschen der Daten reduziert und das Erlernen des Signals vereinfacht [11]. In der Folge soll das Modell eine bessere Performance auf zuvor unbekannten Daten erzielen. Die Auswahl relevanter Informationen stützt sich auf den Erkenntnissen aus den Kapiteln 3 und 5. In Kapitel 3 wurde ermittelt, welche Informationen des Quellcodes von der statischen Codeanalyse PMD verwendet werden, um die Fehler der Fehlerauswahl zu identifizieren. Es wird angenommen, dass diese Informationen wichtige Merkmale für die Identifikation der Fehler darstellen. Ferner wurde in Kapitel 5 erörtert, dass die Verwendung des Quellcodes in einfacher Textform entscheidende Nachteile beinhaltet. Aus diesem Grund bildet der AST die Basis für die IR.

Der AST repräsentiert die syntaktische Struktur des Codes. Dabei bezieht sich jeder Knoten im AST auf ein bestimmtes syntaktisches Element im Quellcode. Diese Eigenschaft ermöglicht es, die Klassifizierungsergebnisse beliebig feingranular auf den Quellcode zurückzuführen. Das ist ein entscheidender Vorteil gegenüber der Daten- oder Kontrollflussgraphen. Sie bilden in der Regel auf Basic-Blocks ab, die normalerweise auf eine Folge von Anweisungen abbilden. Das begrenzt die Aussagekraft der IR und versteckt Informationen, die für eine Analyse wichtig sein können. Der Nachteil am AST ist jedoch das Fehlen von Kontroll- oder Datenflussinformationen. Diese sind gerade für das Auffinden vom Dead-Code wichtig. Eine Kombination des ASTs mit Daten- und Kontrollflussinformationen würde jedoch eine gute Datenrepräsentation für das Lernen auf Quellcodes darstellen. Das hat jedoch zur Folge, dass für Quellcodeelemente, die nicht im AST vorhanden sind, keine Klassifizierung vorgenommen werden. Dabei handelt es sich jedoch größtenteils um Symbole, die nur für den Parser eine Relevanz haben. Dieser Nachteil ist dementsprechend zu vernachlässigen.

Für die Generierung des ASTs werden in dieser Arbeit die Werkzeuge des Programms PMD verwendet. PMD nutzt die *Eclipse Java development tools* (JDT)[1] um Java-Quellcode in einen AST und dessen Symboltabelle zu transformieren. Der generierte AST und die Symboltabelle werden durch Java-Objekte in einer Baumstruktur zur Verfügung gestellt. Jedes Objekt in der Baumstruktur entspricht einem Knoten des ASTs. Dabei ist jedes Objekt von einer Java-Klasse, die die Art des Knotens repräsentiert. Die Art des Knotens ist zum Beispiel eine *FieldDeclaration*, *MethodDeclaration* oder ein *ReturnStatement*. In den Objekten befinden sich außerdem die semantischen Informationen der Symboltabelle und lexikalische Informationen in Form von Token. Ferner wird PMD als Werkzeug für die Erstellung des ASTs verwendet, weil es Methoden beinhaltet, um den AST in einen DFG zu transformieren und auf diesem beispielsweise die Reaching-Definitions einer Variable zu ermitteln.

Zunächst wird die Topologie des AST besprochen. Die Grundlage für die IR bildet der AST. Der AST repräsentiert die grundlegende syntaktische Struktur der Quellcodes und bildet deshalb die Basis für den Graphen. Der syntaktische Kontext jedes Knoten ergibt sich somit durch seine Nachbarknoten. Das ist eine wichtige Eigenschaft, denn einige Fehler ergeben sich durch den unmittelbaren Kontext. Obwohl in der Objektstruktur des ASTs jeder Knoten von einer bestimmten Knotenart ist, sollen die Knoten in der IR alle vom selben Typen sein. Verschiedene Knoten-Typen haben den Vorteil, dass für

[1] https://www.eclipse.org/jdt/

jeden Typen eigene Aggregations- oder Aktualisierungsfunktionen gelernt werden können. Das erhöht jedoch den Rechenaufwand beim Lernen des Modells und steigert dessen Kapazität. Eine zu hohe Kapazität erhöht die Gefahr der Überanpassung und kann somit die Performance des Modells verschlechtern [22]. Ferner werden bei der Verwendung von GNNs Node-Features verwendet. Diese sollen so gewählt werden, dass sie ausreichend Informationen über den Knotentypen kodieren. Aus diesen Gründen wird auf die Einführung verschiedener Knotentypen verzichtet.

Bis hierhin besteht die IR also aus einem Graphen, dessen Struktur der eines ASTs entspricht. Ein AST wird durch einen gewurzelten Baum abgebildet. Dabei handelt es sich um einen gerichtet stark zusammenhängenden kreisfreien Graphen $G = \{V, E\}$ [63][20], wobei V für die Menge aller Knoten und E die Menge aller Kanten steht. In G werden zwei Knoten immer durch genau eine Kante verbunden. Diese Kanten werden im weiteren Verlauf als *child*-Kanten bezeichnet, denn zwischen den Knoten gibt es eine Eltern-Kind Verbindung. In Abbildung 3.2 ist der AST eines Beispielprogramms abgebildet. In jedem AST gibt es genau einen Knoten, der den Eingangsgrad von 0 hat. Dies ist der Wurzelknoten. In Abbildung 3.2 ist das der Knoten mit der Bezeichnung *CompilationUnit*. Knoten die einen Ausgangsgrad von 0 haben, werden als Blätter bezeichnet. Alle weiteren Knoten sind innere Knoten und haben immer mindestens einen Elternknoten (Eingänge) und einen Kinderknoten (Ausgänge). Im Gegensatz zum AST sollen in der IR jedoch keine gerichteten Kanten verwendet werden. Schließlich lernt ein GNN Knotenrepräsentationen, indem die Node-Features durch den Graphen propagiert. In einem gerichteten Baum werden also ausschließlich Informationen von den Elternknoten an die Kinderknoten propagiert. Da ein Kinderknoten jedoch auch wichtige Informationen für die Elternknoten enthalten kann, sollen Kinderknoten auch die Node-Features an die Elternknoten propagieren können. Aus diesem Grund wird die IR als ungerichteter Graph dargestellt, in dem über jede Kante in beide Richtungen propagiert werden kann.

Eine bereits angesprochener Vor- und Nachteil ist die ausschließlich syntaktische Struktur des ASTs. In dieser sind semantisch zusammenhängende Elemente jedoch oft weit voneinander entfernt. Das erschwert einem GNN den Zusammenhang dieser Elemente zu erfassen. Ferner werden die Informationen in einem GNN nur über eine festgelegte Anzahl von Nachbarknoten propagiert. Das kann dazu führen, dass wichtige semantische Informationen nie den korrespondierenden Knoten erreichen. Wie eingangs erwähnt, soll der Graph aus diesen Gründen um einige semantische Kanten erweitert werden. Diese sollen relevante Verbindungen zwischen den Knoten abbilden. Die Auswahl zusätzlicher

Kanten basiert auf den in Kapitel 3 identifizierten, relevanten Merkmalen bei der Fehleranalyse.

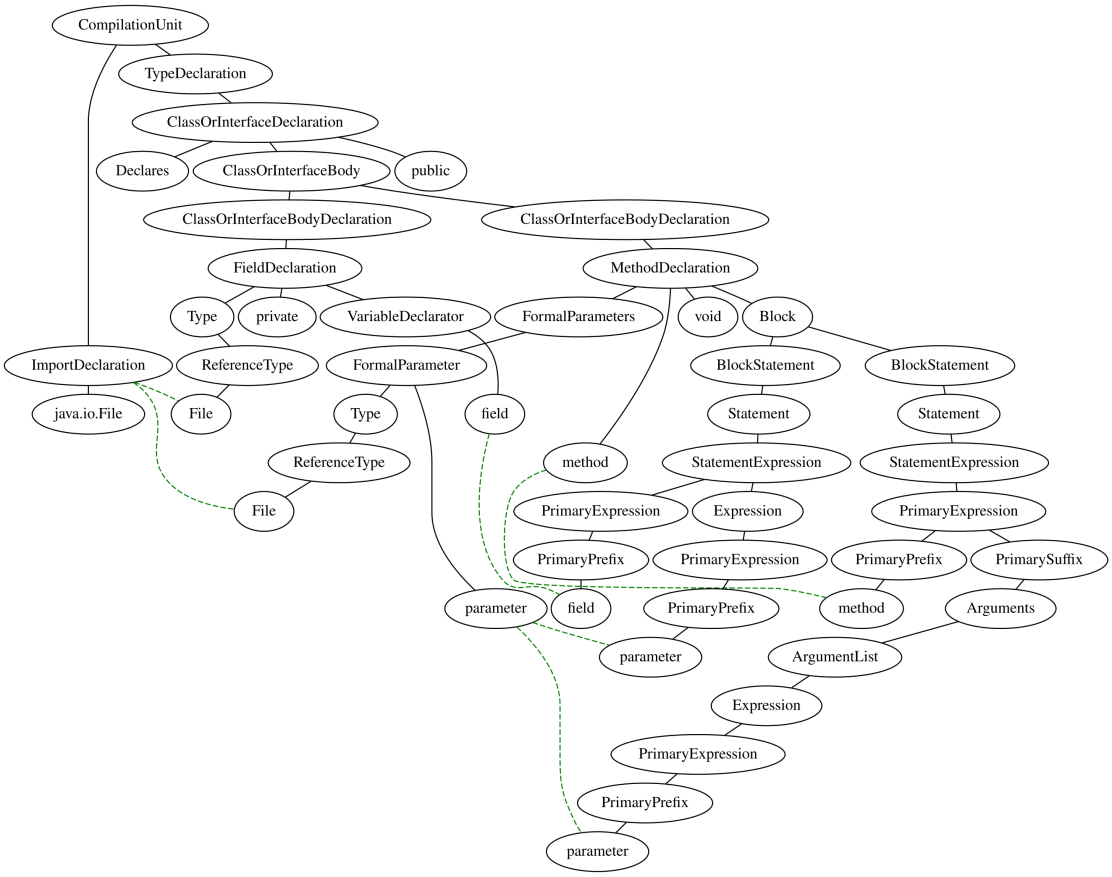

Abbildung 6.2: AST als ungerichteter Graph mit *child* (durchgezogene Linie) und *declares* (gestrichelte Linie) Kanten

Zu diesem Zweck wird die IR zunächst um die Kante *declares* erweitert. Sie verbindet alle Referenzen auf Variablen, Methoden Import oder Klassen mit den entsprechenden Deklarationen. Wenn sich eine Deklaration nicht in der Java-Datei befindet, sondern importiert wird, so führt die Kante zur Import-Deklaration. In Abbildung 6.2 ist die IR des Programms aus Listing 6.1 als ungerichteter Graph mit den Kanten *child* (durchgezogene Linie) und *declares* (grüne gestrichelte Linie) dargestellt.

```
import java.io.File;
public class Declares {
 private File field;

 void method(File parameter){
  field = parameter;
  method(parameter);
 }
}
```

Listing 6.1: Beispielprogramm zur Veranschaulichung der *declares* Kante

Durch die ungerichteten Kanten wird das GNN die Node-Features über jede Kante in beide Richtungen propagieren. Weiterhin ist ersichtlich, dass die *declares* Kante referenzierende und deklarierende Knoten verbindet. Das ist für die Analyse verschiedener Fehler von Vorteil. Zunächst erhalten Variablen-, Methoden- und Import-Deklarationen somit Informationen über dessen Referenzierungen. Ferner wird der Kontext, in dem die Deklarationen referenziert werden, an die Deklaration propagiert. Das ist unter anderem beim Identifizieren von nicht verwendeten Methoden notwendig. Denn eine Methode, die nur in ihrem eigenen Methodenkörper referenziert wird, ist eine nicht verwendete Methode. Die Methode *method* in Listing 6.1 ist entsprechend eine nicht verwendete Methode. Außerdem bietet der Kontext Aufschluss über die Art (lesend oder schreibend) der Referenzen einer Variablendeklaration. Das ist außerdem für die Identifizierung von nicht verwendeten Variablen oder Import-Deklarationen und von Feldvariablen, die final sein können, vorteilhaft. Die Kante *declares* ist entsprechend maßgeblich für das Auffinden von Dead-Code oder Feldvariablen, die final sein können, notwendig. Die Annahme ist, dass durch die *declares* Kante Informationen über das Schreiben und Referenzieren an die Deklarations-Knoten propagiert werden, sodass auf Basis von diesen Knotenrepräsentationen eine Aussage über das Nutzungsverhalten der Variable, Methode oder des Imports getätigt werden kann.

Speziell für das Auffinden von Dead-Stores sind die Informationen des DFG notwendig. Wie bereits in Kapitel 3 erwähnt, werden die Dead-Stores von PMD durch eine *Reaching-Definition-Analyse* identifiziert. Aus diesem Grund soll der Graph um die Kante *reaches* erweitert werden. Sie verbindet jede Nutzung einer Variablen mit ihren Reaching-Definitions, also mit den Definitionen der Variable, die diese Variablennutzung potenziell erreichen können. In Abbildung 6.3 ist ein Teil des ASTs der in Listing 6.2 abgebildeten Java-Methode mit *child* (durchgezogene Linie) und *reaches* (rote gepunktete Linie) Kanten dargestellt. Hier ist ersichtlich, dass die *reaches* Kante nur den schreibenden Zugriff in Zeile 4 mit dem lesenden Zugriff in Zeile 5 verbindet. Die Wertzuweisung in Zeile 3 erreicht keine lesende Variablenreferenz und ist somit ein Dead-Store. Durch die Propagierung über die *reaches* Kante enthalten schreibende Variablenreferenzen unter anderem Informationen darüber, ob sie verwendet werden. Wertzuweisungen, die nicht durch eine

```
1 private void method() {
3  int variable = 1;
4  variable = 2;
5  doSomething(variable);
6 }
```

Listing 6.2: Beispielprogramm zur Veranschaulichung der *reaches* Kante

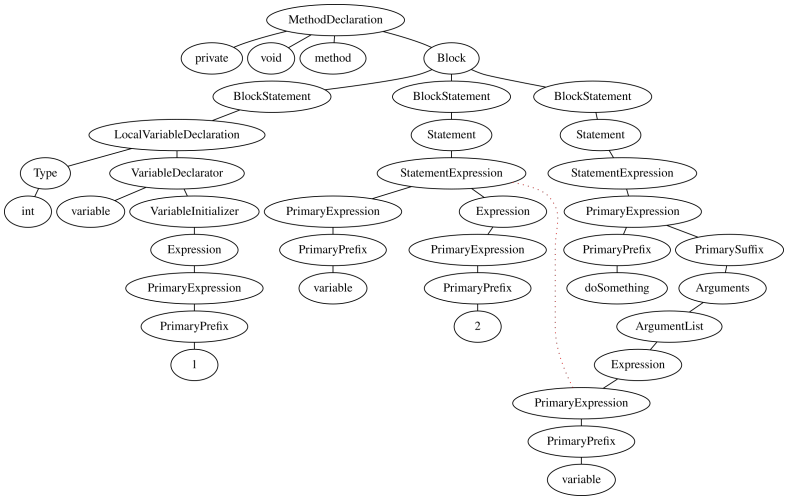

Abbildung 6.3: Teil eines AST als ungerichteter Graph mit *child* (durchgezogene Linie) und *reaches* (gepunktete Linie) Kanten

reaches Kante mit einer Referenzierung verbunden sind, sind entsprechend ein Dead-Store.

Durch die Erweiterung der Kanten *declares* und *reaches* wurde die IR um semantische Informationen und Informationen über den Datenfluss erweitert, die das Signal für die Identifizierung von Dead-Code hervorheben sollen. Die restlichen Fehler beziehen sich auf lokale Informationen, die sich im unmittelbaren Kontext der zu klassifizierenden Knoten befinden. Das umfasst unter anderem die Switch-Einträge für die Identifizierung eines fehlenden Default-Eintrags im Switch-Statement. Das beinhaltet außerdem die Modifizierer einer Feldvariable für die Klassifizierung von Feldvariablen, die final oder statisch sein können. Der lexikalische Token für die Identifikation von Aufrufen der Methoden *System.out.println* oder *printStackTrace* ist ebenfalls im lokalen Umfeld enthalten. Das Ermitteln von Klassen in denen nur die *equals* oder *hashcode* Methode implementiert ist basiert sowohl auf dem lexikalischen Token als auch den Methodendeklarationen oder Implementationen der Klasse. Diese Informationen sind ebenfalls im über wenige Kanten durch den Wurzelknoten zu erreichen. Zu guter Letzt sind auch die Merkmale zum Ermitteln von Methoden mit einer zu hohen Parameteranzahl in der unmittelbaren Nachbarschaft des Knotens enthalten. Eine Erweiterung der IR um weitere Kanten ist entsprechend nicht notwendig.

Schlussendlich besteht die IR aus einem ungerichteten Graphen, in dem jeder Knoten vom selben Typen und jede Kante einer der Kantentypen *declares*, *child* oder *reaches* ent-

spricht. Diese Topologie bietet die Infrastruktur für die Propagierung der Node-Features durch das GNN. Die Auswahl der Node-Features wird im nachfolgenden Abschnitt erörtert.

6.2 Node-Features

Node-Features sind eine wichtige Komponente beim Lernen eines GNN. Sie werden auf Basis der Nachbarschaftsmatrix durch gelernte Aggregations- und Aktualisierungsfunktionen im Netz propagiert. Es ist also wichtig, dass die Node-Features relevante Merkmale des jeweiligen Knotens repräsentieren. Die Auswahl relevanter Merkmale hängt entsprechend von den zu identifizierenden Fehlern ab. Konkret sollen die Node-Features einen Knoten im Graphen beschreiben.

Für die Fehleranalyse verwendet PMD strukturelle und semantische Informationen. Dabei wird der AST sowohl auf Bezeichner wie z.B. Methoden- oder Variablennamen als auch auf strukturelle Informationen wie Knotentypen überprüft. In einem AST bilden die Blätter zumeist auf lexikalische Token wie Bezeichner, Schlüsselwörter oder Literale ab. Innere Knoten und der Wurzelknoten bilden in der Regel auf strukturelle Elemente wie z.B. Methoden- oder Variablendeklarationen ab. Entsprechend sollen die Node-Features für Wurzelknoten und innere Knoten aus dem Knotentyp bestehen. Hierdurch bleiben Merkmale über strukturelle Elemente in der IR enthalten. Das bedeutet, dass ein Knoten, der auf die Deklaration der Methode abbildet, diese Informationen in den Node-Features enthält und dies an seine Nachbarknoten propagiert. Bei den Knotentypen handelt es sich um eine feste Menge von Werten. Der von JDT generierte AST besteht aus insgesamt bis zu 120 verschiedenen Knotentypen, die jeweils auf Codeelemente abbilden.

Für die Blätter des ASTs wird der lexikalische Token als Node-Feature verwendet. Dabei handelt es sich hauptsächlich um Bezeichner, Literale und Schlüsselwörter. Während die Schlüsselwörter ebenfalls aus einer festen Anzahl von Werten bestehen, werden Literale und Bezeichner von den Entwickler:innen definiert. Ein Literal ist die Quellcoderepräsentation eines primitiven Werts, eines Strings oder null [23]. Das kann unter anderem beliebigen Zahlen oder Zeichenketten entsprechen. Ein solcher Wert ist für die Fehleranalyse jedoch nicht von Bedeutung. Denn wie bereits aufgezählt, werden für die Identifikation der Fehler nur die Informationen von Bezeichnern und strukturellen Elementen verwendet. Deshalb wird anstelle des Literalwerts der Literaltyp als Node-Feature verwendet.

Die Node-Features für Literale bestehen also aus den Literaltypen. In der Programmiersprache Java sind das folgende sieben Werte: *IntegerLiteral*, *FloatingPointLiteral*, *BooleanLiteral*, *CharacterLiteral*, *StringLiteral*, *TextBlock* und *NullLiteral*. Hierdurch werden unnötige Informationen entfernt und sowohl der Wertebereich der Node-Features als auch das Rauschen der Daten reduziert. Informationen über die Art des Literals bleiben auf diese Weise erhalten. Bezeichner enthalten unter anderem wichtige Informationen für die Identifikation von aktivem Debug Code oder dem Fehlen einer Implementation der *equals* oder *hashcode* Methode. Konkret geht es dabei um einige wenige Namen wie *System.out.println*, *printStackTrace*, *equals* oder *hashcode*. Das könnte den Entschluss nahelegen, alle Bezeichner, die keine Relevanz für die Fehleranalyse haben, auf einen einzigen Wert abzubilden. Jedoch enthalten Bezeichner auch nützliche Informationen über die beabsichtigte Semantik des Codes und sind eine Hauptquelle für konzeptuelle Informationen [13]. Das umfasst außerdem die Groß- und Kleinschreibung sowie die Verwendung der erlaubten Sonderzeichen _ und *$*. Aus diesem Grund werden die Bezeichner nicht normalisiert und so übernommen, wie sie im Quellcode stehen. Um das Prinzip zur Erstellung der Node-Features zu verdeutlichen, ist in Abbildung 6.4 die reduzierte IR des nebenstehenden Beispielprogramms abgebildet.

Weil die Node-Features als Eingabe in das GNN verwendet werden, müssen sie in numerischer Form vorliegen. Für die Kodierung von Texten in eine numerische Repräsentation haben sich unterschiedliche Verfahren etabliert. Eine in der NLP weit verbreitete Methode ist das Einbetten von diskreten Werten (Wörtern) in einen Vektorraum [54]. Diese Art der Vektorrepräsentation wird *Embedding* genannt. Diese Embeddings werden gelernt, sodass sich semantisch ähnelnde Werte im Vektorraum näher beieinander befinden. Man spricht von einer verteilten Darstellung, weil die Bedeutung des Werts über mehrere Vektorkomponenten verteilt wird. Der semantische Zusammenhang ist abhängig von dem verwendeten Lernverfahren. Gegenüber anderen Kodierungsmethoden haben die Embeddings den Vorteil, dass sie aus vergleichsweise kleinen Vektoren bestehen. Alternativen wie die One-Hot-Kodierungen bestehen aus Vektoren, die so groß wie das gesamte Vokabular sind. Das Vokabular besteht aus den möglichen Werten, die für das Lernen des Modells verwendet wurden, also aus dem Set aller Node-Features, die in den Lerndaten vorkommen. Durch die Verwendung von Bezeichnern steigt diese Menge also mit der Anzahl der verwendeten Trainingsdaten. Große Vektoren sind rechentechnisch ineffizienter und können zu Problemen bei der Optimierung des Modells führen. Aus diesem Grund sollen die Werte der Node-Features durch Embeddings dargestellt werden.

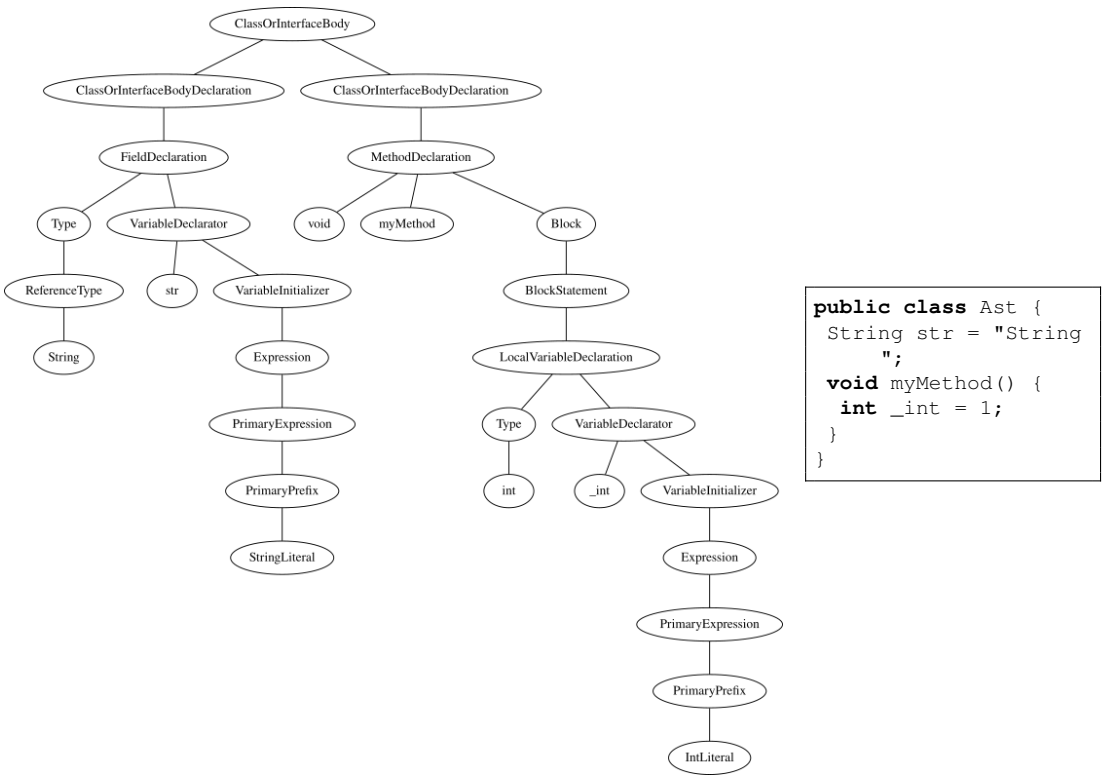

Abbildung 6.4: Reduzierter AST mit normalisierten Node-Features

Es gibt verschiedene Möglichkeiten, um Embeddings für einen Wert zu generieren. Das Lernen von Embeddings gehört zu den bereits in Kapitel 5 erwähnten Feature-Learning Methoden. Außerdem gibt es bereits eine Vielzahl von vorab gelernten Wort-Embeddings, die ohne eine weitere Verarbeitung direkt verwendet werden können. Diese wurden zumeist auf einem sehr großen Vokabular, bestehend aus mehreren Milliarden Wörtern einer bestimmten Sprache, gelernt. Quellcode enthält jedoch von Natur aus viele neologistische Ausdrücke, die Entwickler:innen während der Programmierung festlegen. Außerdem kann Java-Quellcode in verschiedenen Sprachen geschrieben werden, sodass vorab gelernte Embeddings einer bestimmten Sprache nicht ausreichen. Aus diesen Gründen werden in dieser Arbeit keine vorab gelernten Embeddings verwendet.

Eine weitere Möglichkeit ist es jedoch, die Embeddings durch einen entsprechenden ML-Algorithmus oder ein neuronales Netz in einem eigenständigen Modell zu lernen. Diese Algorithmen sind zumeist selbstüberwacht, sodass die Wörter auf Basis der Kontextwörter eingebettet werden [44]. Die Semantik der Embeddings ergibt sich also aus umliegenden Wörtern. Das Problem bei dieser Variante ergibt sich daraus, dass Node-Features

nicht als geordneter Text, sondern in Form einzelner Wörter vorliegt. Der ursprüngliche Quellcode als Text kann nicht als Basis für das Einbetten der Wörter verwendet werden, weil in diesem die Bezeichner der Knotentypen und Literale fehlen. Aus diesen Gründen wird auf das Vorablernen eines eigenständigen Sprachmodells verzichtet.

Schlussendlich können die Embeddings auch mit dem GNN zusammen gelernt werden. Das entspricht dem sogenannten End-To-End-Learning. Beim End-To-End-Learning der Embeddings wird dem GNN eine Embedding-Schicht hinzugefügt, der für jedes Node-Feature in den Trainingsdaten ein Embedding generiert und mit dem restlichen Netz zusammen lernt. Hierdurch werden die Embeddings nicht auf dessen Kontextwörter abgebildet, sondern im Endeffekt auf die zu prognostizierende Klasse.

Alle Modelle, die Wort-Embeddings verwenden, haben jedoch das Problem, dass die Eingabedaten früher oder später ein Node-Feature enthalten, das nicht in den Lerndaten vorhanden gewesen ist. Bei diesen unbekannten Eingaben spricht man vom sogenannten Out-Of-Vocabulary-Problem. Im Falle der Node-Features wird es sich bei unbekannten Eingaben um einen Bezeichner handeln, den das Netz zuvor noch nicht gesehen hat. Die Embeddings für Literale und Knotentypen sollten allesamt in einem ausreichend großen Testdatensatz vorhanden sein, sodass bei diesen keine Gefahr besteht. Die einfachste Lösung ist es, Eingaben, in denen ein unbekanntes Wort vorkommt, herauszufiltern. Das würde jedoch dazu führen, dass die Analyse einer Java-Datei von den gewählten Bezeichnern der Entwickler:innen abhängt. Bezeichner können jedoch in vielen Sprachen verfasst werden und enthalten oft neue Wortschöpfungen. Die Chance, dass unbekannte Bezeichner in den Eingabedaten enthalten sind, ist entsprechend hoch. Es gibt unter anderem zwei Möglichkeiten, dieses Problem zu umgehen. Zum einen können zufällig ausgewählte Node-Features der Lerndaten vorher durch einen Bezeichner, z.B. *<unbekannt>*, für unbekannte Werte ersetzt werden. Bei unbekannten Eingaben wird im Anschluss bei der Klassifizierung von neuen Eingabedaten das Embedding für den Wert *<unbekannt>* verwendet. Das hat den Nachteil, dass die Wörter im Lerndatensatz reduziert werden und die Chance auf unbekannte Node-Features in neuen Daten steigt. Die zweite Lösung besteht darin, den Durchschnitt aller Embeddings zu verwenden, die selten vorkommen. Bei unbekannten Node-Features wird es sich hoher Wahrscheinlichkeit nach um Bezeichner handeln, die nur selten vorkommen. Der Durchschnitt aller Embeddings von Node-Features, die z.B. nur einmal vorkommen, kann also eine angemessene Repräsentation für neue unbekannte Wörter sein. Die Annahme ist, dass die Embedding-Schicht allen Bezeichnern, die nur selten vorkommen, einen ähnlichen Wert zuweisen. Somit bieten entsprechend generierte Embeddings einen guten Schätzwert für das fehlende Embedding.

In diesem Abschnitt wurde der Inhalt der Node-Features erörtert. Ferner wurde bereits die Kodierung der Node-Features in eine numerische Form erläutert. Somit steht neben der Topologie der IR auch der zu propagierende Inhalt fest.

6.3 Das Netz

Nachdem die Topologie der IR und die Auswahl der Node-Features getätigt wurde, folgt in diesem Abschnitt der Aufbau des künstlichen neuronalen Netzes. Das umfasst die Schritte Kodierung, GNN und Klassifikation aus Abbildung 6.1. Die Entscheidung für einen End-To-End Lernprozess orientiert sich an der praxisorientierten Methodologie von Goodfellow et. al. [22]. Dort wird empfohlen, einen Prototypen stets durch einen End-To-End-Prozess zu implementieren.

Die Eingabe in das Netz besteht sowohl aus der Nachbarschaftsmatrix A der Größe VxV, den Node-Features X, einem Vektor der Größe V, als auch den Kantentypen R, einem Vektor der Größe E. Dabei ist V die Menge aller Knoten und E die Menge aller Kanten. Weil es sich um ein End-To-End Lernprozess handelt, bestehen die Node-Features aus Indices, bei denen jeder Index für genau ein Wort im Node-Feature-Vokabular steht. Diese Indices werden zunächst durch eine Embedding-Schicht in Embeddings der Dimension d transformiert. Die Embedding-Schicht speichert zu jedem Index einen d-dimensionalen Vektor, der während des Lernens optimiert wird. Die Wahl der Dimension d beeinflusst also unmittelbar die Kapazität des Netzes. Die Ausgabe der Embedding-Schicht X besteht aus einer Matrix der Größe Vxd.

Ein GNN besteht aus mehreren Schichten, bei denen jede Schicht eine Propagierungsiteration darstellt. Jeder Propagierungsschritt besteht im Wesentlichen aus der Aktualisierung des eigenen Hidden-States h_v (Formel 6.2) und der Aggregation der Nachbar-Hidden-States zu einer Nachricht $m^k_{N(v)}$ (Formel 6.1). Dabei steht k für die aktuelle Iteration und N für die Menge aller Nachbarn.

$$m^k_{N(v)} = AGGR^k\left(\left\{h^k_n, \forall n \in N(v)\right\}\right) \tag{6.1}$$

$$h^{k+1}_v = AKT^k\left(h^k_v, m^k_{N(v)}\right) \tag{6.2}$$

Die meisten GNNs unterscheiden sich also im Grunde nur durch diese beiden Funktionen [25]. Die Konzeption des GNNs besteht also aus der Auswahl entsprechender Aggregations- und Aktualisierungsfunktionen.

In den vergangenen Jahren hat sich das *Graph Convolutional Network* (GCN) von Kipf et. al. [36] als effektives GNN für eine Vielzahl an Aufgabenstellungen etabliert [25]. Das Aktualisieren wird beim GCN durch sogenannte *Self-Loops* realisiert. Dafür wird die Nachbarschaftsmatrix A mit der Identitätsmatrix I addiert, sodass jeder Knoten auf sich selbst zeigt. Die Aggregation der Nachbar-Hidden-States wird durch eine symmetrisch normalisierte Addition von $\{h_n, \forall n \in N(v) \bigcup \{v\}\}$ realisiert. Aufgrund der Self-Loops ist in den Nachbar-Hidden-States auch h_v enthalten. Das Ergebnis wird anschließend mit einer lernbaren Gewichtungsmatrix addiert. Ein Nachteil an diesem Modell ist die Vernachlässigung der Kantentypen.

Das *Relational Graph Convolutional Network* (RGCN) ist eine Variation des GCN durch Schlichtenkrull et. al. [57] und ermöglicht die Einbeziehung von Kantentypen. Im Grunde wird bei der Aggregierung für jeden Kantentyp $r \in R$ eine eigene Gewichtungsmatrix W_r gelernt (Formel 6.3). Dabei steht R für die Anzahl möglicher Kantentypen und $f()$ für eine Normalisierungsfunktion.

$$m^k_{N(v)} = \sum_{r \in R} \sum_{n \in N_{r(u)}} \frac{W^k_r h^k_n}{f(N(v))} \tag{6.3}$$

$$h^{k+1}_v = W_0 * h^k_v + m^k_{N(v)} \tag{6.4}$$

Die Aktualisierung erfolgt durch die Multipliaktion von h^k_v mit der Gewichtungsmatrix W_0 und der anschließenden Addition mit $m^k_{N(v)}$. Die Multiplikation von W_0 mit h^k_v ist das Äquivalent zur Self-Loop, wobei W die Gewichtungsmatrix des speziellen Self-Loop-Kantentyp ist. Durch die Einführung mehrerer lernbarer Gewichtungsmatrizen für jeden Kantentyp steigt die Kapazität des Netzes mit jeder GNN-Schicht und jedem Kantentypen. Der Einbezug mehrerer Kantentypen bringt jedoch den Vorteil mit sich, dass das Modell zwischen den einzelnen Kanten unterscheidet und somit lokale bzw. strukturelle Zusammenhänge anders verarbeitet als semantische Zusammenhänge. Aus diesen Gründen erscheint das RGCN als eine sinnvolle Option. Die Normalisierungsfunktion $f(N(v))$ wird mit der Anzahl der Knoten gleichgesetzt, sodass die Aggregation dem Durchschnitt entspricht. Hierdurch wird der Fokus der Aggregation auf die Node-Features und nicht auf die Struktur des Graphen gelegt. Die Normalisierung der Nachbarknoten reduziert den Einfluss des Grads eines Knotens. Bei einer einfachen Addition aller h_n würden Knoten mit vielen Nachbarn einen entsprechend hohen Wert erzielen. Durch die Normalisierung wird dieser Einfluss reduziert, sodass die Bedeutung der Node-Features eine höhere Gewichtung auf die resultierenden Node-Embeddings haben.

Die Anzahl der Iterationen K und somit die Anzahl der RGCN-Schichten, wird experimentell festgelegt. Weil die zu erkennenden Fehler unterschiedliche Charakteristiken haben und ihr Optimum möglicherweise nach einer unterschiedlichen Anzahl von Iterationen erreichen, wird die Aktualisierungsfunktion um sogenannte *Jumping-Knowledge-Connections* erweitert. Hierdurch wird die Knotenrepräsentation e_v nicht nur durch den letzten Hidden-State h_v^K gebildet, sondern durch die Konkatenation der Hidden-States aller Iterationen (Formel 6.5). Dabei ist K die Anzahl aller Iterationen.

$$e_v = h_v^k \oplus h_v^{k+1} ... \oplus h_v^K \tag{6.5}$$

Daraus resultieren die Knotenrepräsentationen E von der jedes e_v die Größe der Dimension d_hxK hat. Wobei d_h die Dimension der Hidden-States ist.

Zwischen den RGCN-Schichten wird die Aktivierungsfunktion ReLu verwendet. Aktivierungsfunktionen aktivieren die Ausgaben der Neuronen [22]. Durch eine nicht lineare Funktion werden die Ausgaben in einen bestimmten Wertebereich transformiert und an die darauffolgende Schicht weitergeleitet. Diese Nicht-Linearität ermöglicht dem Netz das Lernen von komplexen Mustern der Daten. Durch die ReLu-Funktion werden negative Werte mit 0 gleichgesetzt, während positive Werte unverändert weitergegeben werden. Durch ihre annähernde Linearität beinhaltet die ReLu-Aktivierungsfunktion viele Eigenschaften, die eine Optimierung des Modells durch das Gradientenabstiegsverfahren vereinfachen. Aufgrund der nachgewiesenen Effektivität hat sich die ReLu-Funktion zu einer Art Standard-Aktivierungsfunktion etabliert und wird aus diesem Grund auch in dieser Arbeit verwendet.

Nach den RGCN-Schichten hat jeder Knoten eine Knotenrepräsentation e_v mit einer Dimension von d_hxK. Dieser Stand wird in Abbildung 6.1 durch den Graphen, der auf die Box mit der Bezeichnung GNN folgt, dargestellt. Diese enthalten nun Informationen über die Nachbarschaft des Knotens, sodass die Knotenrepräsentation ausreichend Informationen für eine Klassifizierung der Fehler beinhaltet. Für eine Klassifikation der einzelnen Knoten fehlt lediglich noch eine weitere Schicht, die die Dimension der Knotenrepräsentation auf die Anzahl der zu prognostizierenden Klassen reduziert. Das geschieht durch die Ausgabeschicht, die eine lineare Abbildung lernt, anhand derer die Dimension von E auf die Anzahl der zu prognostizierenden Klassen K reduziert wird (Formel 6.6). Wobei W eine lernbare Gewichtungsmatrix ist und b der Bias.

$$s_v = e_v W^T + b \tag{6.6}$$

Weil jeder Knoten genau einer Klasse zugeordnet werden kann, wird auf die Ausgabe der Lineare-Schicht s_v die Softmax-Aktivierungsfunktion angewandt. Hierdurch wird eine Wahrscheinlichkeitsverteilung über die Komponenten des Vektors s_v erstellt. Das Resultat ist ein Vektor, der aufsummiert 1 ergibt, in dem jede Komponenten die Wahrscheinlichkeit für eine der Klassen aus K angibt. Die Vektorkomponente mit dem höchsten Wert ergibt die prognostizierte Klasse.

Wie bereits erwähnt, haben die Parameter wie die Embedding-Dimension d, die Hidden-State-Dimension d_h und die Anzahl der Kantentypen K einen großen Einfluss auf die Kapazität des Modells. Die Kapazität eines Modells sollte jedoch stets der Komplexität der Aufgabe angepasst werden. Ist diese zu gering, kann das Modell nicht die zugrundeliegende Verteilung der Eingangsdaten lernen (Unteranpassung). Daraus resultiert eine schlechte Performance auf den Test- und Validierungsdatensatz. Wenn die Kapazität zu groß ist, dann kann das Modell die Eingabeinformationen auswendig lernen (Überanpassung). Die Performance auf den Lerndaten wird besser während sich die Generalisierung verschlechtert. Weil beim RGCN bereits die Anzahl der Knotentypen eine zu große Kapazität erwirken kann, empfiehlt sich die Anwendung von Regularisierungsmethoden. Bei der Regularisierung handelt es sich um Verfahren, durch die der Generalisierungsfehler unter Hinnahme eines höheren Lernfehlers reduziert wird [22]. Dabei wird der Generalisierungsfehler durch die Modifikation des Netzes reduziert. Laut Goodfellow et al. [22] sind die besten Netze (in Bezug auf den Generalisierungsfehler) jene, welche eine hohe Kapazität besitzen und angemessen regularisiert wurden. Aus diesem Grund wird an dieser Stelle eine Dropout-Schicht zwischen die Ausgabe des RGCN und die Ausgabeschicht geschaltet. Der Dropout bezeichnet das Deaktivieren von Neuronen während der Trainingsphase. Die Neuronen werden zufällig nach einer vordefinierten Quote ausgewählt. Deaktivierte Neuronen werden während einer bestimmten Forward- und Backpropagation des Netzes nicht berücksichtigt.

7 Experimente

Das in Kapitel 6 entwickelte Konzept ist das Resultat verschiedener Designentscheidungen. Sowohl die Bewertung dieser Entscheidungen als auch die Überprüfung der Machbarkeit und allgemeinen Anwendbarkeit des Konzepts, soll auf Basis des in diesem Kapitel gelernten Prototyps geschehen. Zunächst wird der Aufbau des Lern-, Validierungs- und Testdatensatzes beschrieben. Das umfasst sowohl eine Erläuterung der Datenauswahl als auch eine Erklärung des Verfahrens zum Labeln der Daten. Durch das Labeln wird jedes Beispiel in den Daten mit der entsprechenden Klasse gekennzeichnet. Der somit erstellte Datensatz bildet die sogenannte *Ground-Truth*. Anschließend wird die Implementierung des Lernprozesses beschrieben. Das beinhaltet außerdem die manuelle Optimierung verschiedener Hyperparameter. Nachfolgend wird ein geeigneter Modellkandidat ausgewählt und auf dem gesamten Lerndatensatz gelernt. Die Performance des Prototyps wird danach auf dem Testdatensatz gemessen.

7.1 Lern-, Validierungs- und Testdaten

Das Ziel des überwachten maschinellen Lernens ist die Ermittlung eines Modells auf Basis von Ein- und Ausgabedaten, die im Anschluss auf neuen Daten Vorhersagen trifft [22]. Als neue bzw. unbekannte Daten werden Daten bezeichnet, die nicht in den Lerndaten enthalten waren [9]. Das Problem der Generalisierung bezieht sich darauf, wie gut ein erlerntes Modell auf neuen Daten Vorhersagen trifft [50]. Die Generalisierung kann durch den sogenannten Generalisierungsfehler gemessen werden. Er wird üblicherweise durch die Leistung des Modells auf neuen Eingangsdaten geschätzt [22]. Es ist gängige Praxis, während des Lernens die Performance des Netzes mit einem Validierungsdatensatz zu bestimmen [9]. Der Validierungsdatensatz bildet sich aus einer Teilmenge der Lerndaten und dient zur Anpassung der Hyperparameter des Netzes. Deshalb wird der Generalisierungsfehler auf Basis des Validierungsdatensatzes meist unterschätzt. Jedoch bietet dieser eine bessere Schätzung als die Validierung auf dem Lerndatensatz [22]. Diese

Schätzung dient als Grundlage für die Auswahl und Einstellung der Hyperparameter des Netzes. Ferner ist die Güte der Messung des Generalisierungsfehlers auf Basis von einem fest definierten Validierungsdatensatz stark abhängig von der Aufteilung der Daten. Aus diesem Grund wird bei der Aufteilung für Klassifikationsaufgaben die Stratifikation des Datensatzes empfohlen [50]. Das bedeutet, dass die Verteilung der zu prognostizierenden Klassen in Lern- und Validierungsdaten gleich sein muss. Eine zufällige Aufteilung führt zu einem verfälschten Verhältnis der Klassen im Datensatz und kann das Fehlen ganzer Klassen im Lerndatensatz bewirken [50]. Für die nachfolgenden Experimente wird der Lerndatensatz in 80 % Lerndaten und 20% Validierungsdaten stratifiziert aufgeteilt. Dies ist eine gängige Rate für die Aufteilung der Daten [22]. Das endgültige Modell wird nach der Optimierung der Hyperparameter auf Basis des Datensatzes, bestehend aus Lern- und Validierungsdaten, gelernt. Für die Überprüfung der Performance des fertig gelernten Modells wird ein Testdatensatz aufgebaut. Dieser überschneidet sich nicht mit den Lern- und Validierungsdaten und wird deshalb auch nicht für die Optimierung der Modellparameter verwendet.

Der Lern- und Validierungsdatensatz besteht aus insgesamt 59.643 Java-Dateien aus 17 zufällig ausgewählten Repositories des Filehosters GitHub. Durch die Werkzeuge des Programms PMD wird für jede Datei ein AST inklusive Symboltabelle generiert. Anschließend wird jeder AST durch ein Programm in die IR transformiert. Das entspricht dem Schritt *Datenvorbereitung* in der ML-Pipeline, dargestellt in Abbildung 6.1. Für diesen Prozessschritt wurde durch den Autor dieser Arbeit ein Skript in der Programmiersprache Kotlin[1] entwickelt. Das Skript befindet sich auf der beiliegenden CD. Im Grunde wird dabei der AST durchquert. Währenddessen werden für jeden Knoten die Node-Features und die entsprechenden Kanten ermittelt. Der daraus resultierende Graph wird anschließend auf zwei Dateien verteilt und im CSV-Format abgespeichert. Die Knoten werden in einer Datei gespeichert, in der für jeden Knoten der Dateiname der entsprechenden Java-Datei, Start- und Endzeile sowie Start- und Endspalte des abgebildeten Codeelements, der Knotentyp und die Node-Features verzeichnet werden. Durch den Dateinamen, die Spalten- und Zeilenpositionen und den Knotentyp hat jeder Knoten einen eindeutigen Identifizierer (Knoten-ID). Durch die Informationen in der Knoten-ID kann jeder Knoten auf einen Abschnitt im Quellcode zurückgeführt werden. Die Kanten werden in einer Datei gespeichert, in der für jede Kante der Dateiname der entsprechenden Java-Datei, der Typ der Kante sowie die Knoten-IDs der Start- und Endknoten eingetragen werden. Das Label jedes einzelnen Knotens wird durch die statische Codeanalyse des Programms

[1] https://kotlinlang.org/

PMD ermittelt. Das Ergebnis dieser Analyse wird ebenfalls in einer Datei im CSV-Format abgespeichert. Dort ist für jeden gefundenen Fehler der Dateiname der entsprechenden Java-Datei, Start- und Endzeile sowie Start- und Endspalte als auch einer Bezeichnung für den gefundenen Fehler abgespeichert. Die Bezeichnung für den jeweiligen Fehler ergibt sich aus dem Namen der Regel. Alle Regeln wurden in Kapitel 3 vorgestellt. Durch den Dateinamen und die Zeilen- und Spaltenpositionen kann jedem Knoten das entsprechende Label zugewiesen werden. Weil in einigen Fällen mehrere Knoten im AST auf dieselbe Position im Quellcode abbilden, kommt es vor, dass mehrere benachbarte Knoten durch einen einzigen Fehler im Quellcode gelabelt werden. Alle Knoten, für die kein Fehler während der Fehleranalyse gefunden wurde, sind fehlerfrei und werden mit der Klasse *Flawless* gelabelt. Das Resultat dieser Transformation ist ein Datensatz in dem jede Java-Datei in die IR transformiert und anschließend gelabelt wurde.

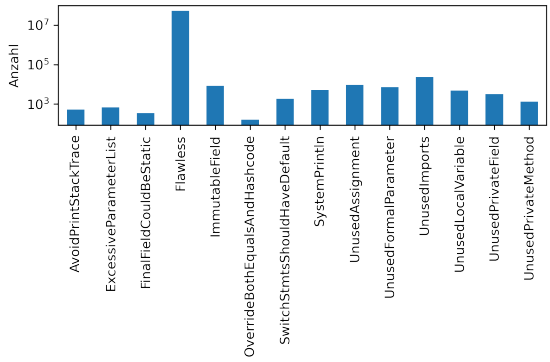

Abbildung 7.1: Labelverteilung in den Lerndaten

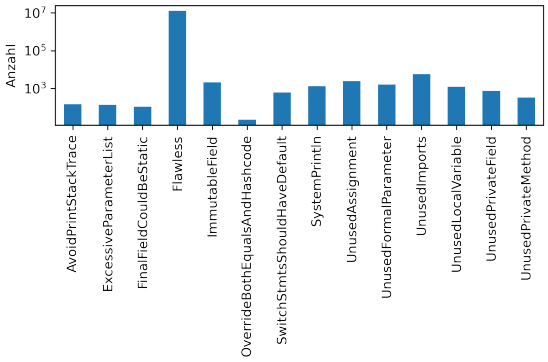

Abbildung 7.2: Labelverteilung in den Validierungsdaten

Nach der Vorbereitung der Daten bestehen Lern- und Validierungsdatensatz insgesamt aus 65.598.989 Knoten und 179.681.908 Kanten. In den Abbildung 7.1 und 7.2 ist die Verteilung der Label auf den Lern- und Validierungsdaten abgebildet. Aufgrund der Stratifizierung ist in beiden Datensätzen das Verhältnis der Label gleich. Weil die Verteilung extrem unausgeglichen ist, wird die Y-Achse logarithmisch dargestellt. Die Diskrepanz zwischen fehlerfreien Knoten *Flawless* und fehlerbehafteten Knoten ist offensichtlich. In den Lerndaten sind insgesamt 66.746 Knoten mit einem der zu identifizierenden Fehler und 52.162.526 Knoten, die als fehlerfrei deklariert sind. In den Validierungsdaten liegt dieses Verhältnis bei 15.604 fehlerbehafteten zu 13.354.113 fehlerfreien Knoten. In den Abbildungen 7.3 und 7.4 ist die Verteilung der Label ohne die fehlerfreien Knoten abgebildet. In dieser Darstellung ist ersichtlich, dass auch zwischen den fehlerbehafteten Knoten eine große Diskrepanz besteht. Während das Label *UnusedImports* mit 23.278

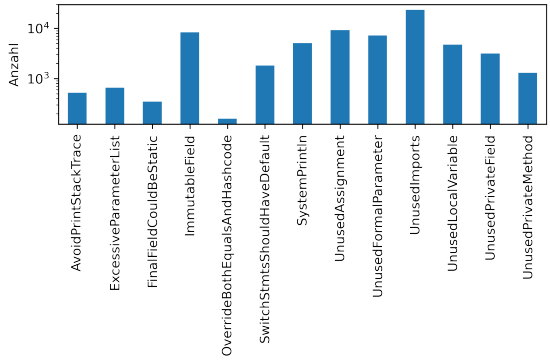

Abbildung 7.3: Labelverteilung in den Lerndaten ohne *Flawless*

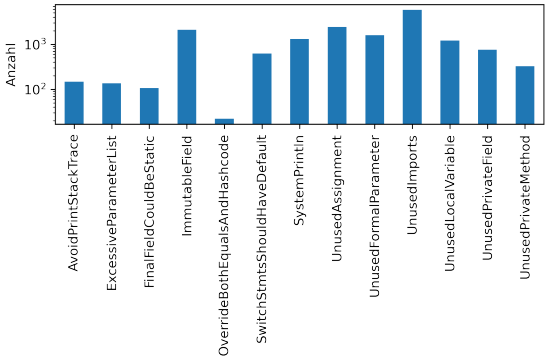

Abbildung 7.4: Labelverteilung in den Validierungsdaten ohne *Flawless*

Beispielen am häufigsten vorkommt, sind für die Klasse *OverrideBothEqualsAndHashcode* lediglich 141 Beispiele verzeichnet. In den Validierungsdaten liegt dieses Verhältnis bei 5.923 *UnusedImports* Beispielen zu 40 *OverrideBothEqualsAndHashcode* Beispielen. In Kapitel 4 wurde der F_1-Score bereits als geeignete Performance-Metrik vorgestellt. Die Aggregation des F_1-Score jeder Klasse zu einer einzigen Kennzahl muss entsprechend eine Normalisierung der Beispielanzahl beinhalten. Andernfalls wird die Performance zu sehr von den Klassen mit den meisten Beispielen beeinflusst. Aus diesem Grund wird der macro F_1-Score als Performance-Metrik verwendet. Der macro F_1-Score berechnet sich aus dem Durchschnitt des F_1-Scores jeder Klasse. Bei der Auswahl einer geeigneten Kostenfunktion sollte der unausgeglichene Datensatz ebenfalls beachtet werden. Insgesamt bestehen Lern- und Validierungsdatensatz aus 668 Knoten mit der Klasse *AvoidPrintStackTrace*, 795 Knoten mit der Klasse *ExcessiveParameterList*, 450 Knoten mit der Klasse *FinalFieldCouldBeStatic*, 65.516.639 Knoten mit der Klasse *Flawless*, 10.373 Knoten mit der Klasse *ImmutableField*, 179 Knoten mit der Klasse *OverrideBothEqualsAndHashcode*, 2.430 Knoten mit der Klasse *SwitchStmtsShoudlHaveDefault*, 6.366 Knoten mit der Klasse *SystemPrintln*, 11.605 Knoten mit der Klasse *UnusedAssignment*, 8.830 Knoten mit *UnusedFormalParameter*, 29.201 Knoten mit *UnusedImport*, 5.931 Knoten mit der Klasse *UnusedLocalVariable*, 3.895 Knoten mit der Klasse *UnusedPrivateField* und 1.627 Knoten mit der Klasse *UnusedPrivateMethod*.

Der Testdatensatz besteht aus insgesamt 9 zufällig ausgewählten Repositories mit 19.762 Java-Dateien. Keines der Repositories im Testdatensatz war in den Lern- und Validierungsdaten enthalten. Die IRs der Testdaten bestehen aus insgesamt 45.117.656 Kanten

und 16.319.556 Knoten. In den Abbildungen 7.5 und 7.6 ist die Labelverteilung der Test-

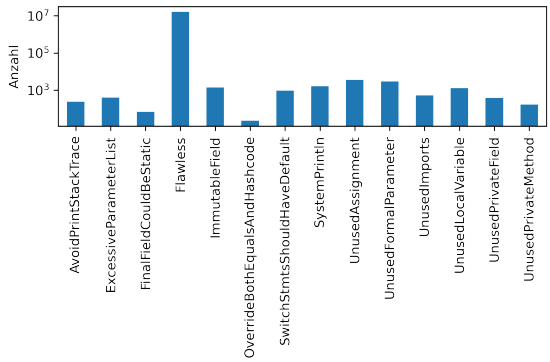

Abbildung 7.5: Labelverteilung in den Testdaten

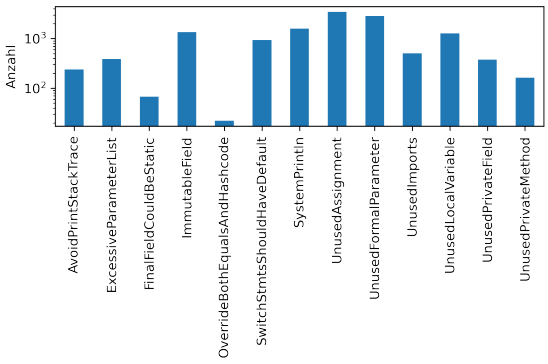

Abbildung 7.6: Labelverteilung in den Testdaten ohne *Flawless*

daten mit einer logarithmischen Y-Achse dargestellt. In absoluten Zahlen wurden 238 Knoten mit der Klasse *AvoidPrintStackTrace*, 390 Knoten mit der Klasse *ExcessiveParameterList*, 67 Knoten mit der Klasse *FinalFieldCouldBeStatic*, 16.306.335 Knoten mit der Klasse *Flawless*, 1.358 Knoten mit der Klasse *ImmutableField*, 22 Knoten mit der Klasse *OverrideBothEqualsAndHashcode*, 936 Knoten mit der Klasse *SwitchStmtsShoudlHaveDefault*, 1.582 Knoten mit der Klasse *SystemPrintln*, 3.462 Knoten mit der Klasse *UnusedAssignment*, 2.842 Knoten mit der Klasse *UnusedFormalParameter*, 5.088 Knoten mit der Klasse *UnusedImport*, 1.278 Knoten mit der Klasse *UnusedLocalVar*, 375 Knoten mir *UnusedPrivateField* und 163 Knoten mit der Klasse *UnusedPrivateMethod* gelabelt. Obwohl der Lern- und der Testdatensatz auf komplett unterschiedlichen Repositories basieren, haben die Labels der beiden Datensätze eine ähnliche Verteilung.

7.2 Das Modell lernen

Das in Kapitel 6 entwickelte Netz wird mithilfe von PyTorch[2] und dessen Erweiterung PyTorch Geometric[3] realisiert. PyTorch ist ein ML-Framework für die Programmiersprache Python[4]. Es bietet eine umfangreiche Funktionsbibliothek für das effiziente Lernen von künstlichen neuronalen Netzen. PyTorch Geometric erweitert das ML-Framework um

[2]https://pytorch.org/
[3]https://pytorch-geometric.readthedocs.io/
[4]https://www.python.org/

verschiedene Methoden für das Lernen von GNNs. In den Bibliotheken sind unter anderem bereits die Embedding-Schicht, die RGCN-Schicht und die Lineare-Schicht enthalten. Weiterhin beinhaltet das Framework die meisten Aktivierungsfunktionen und Implementationen für verschiedene Kostenfunktionen und Optimierer. Entsprechende Komponenten müssen für das Lernen des Modells lediglich verkettet werden. Das Skript zum Lernen des Modells wurde durch den Autor dieser Arbeit mithilfe der genannten Funktionsbibliotheken in der Programmiersprache Python verfasst. Das entsprechende Skript ist auf der beigelegten CD hinterlegt. Im Schaubild 6.1 der ML-Pipeline umfasst das somit gelernte Modell die Schritte *Kodierung*, *GNN* und *Klassifizierung*.

Beim überwachten Lernen wird das künstliche neuronale Netz gelernt, indem iterativ die Modellprognosen mit den Werten aus dem Faktenvorrat verglichen werden und die Gewichte der Neuronen im Netz so angepasst werden, dass in jeder Iteration der Fehler zwischen Modellprognose und Lerndaten reduziert wird [22]. Jede Iteration entspricht einem Optimierungszyklus, indem sich das Modell der Zielfunktion annähert. Dabei spricht man von einem Batch, in dem eine zuvor festgelegte Anzahl an Beispielen aus dem Faktenvorrat prognostiziert und mit der Ground-Truth verglichen wird. Die Optimierung basiert auf dem kombinierten Fehler aller Beobachtungen eines Batch. Wenn alle Daten einmal zum Lernen verwendet wurden, spricht man von einer Epoche [50]. Der Fehler, also der Unterschied zwischen Prognose und Wahrheit, wird durch eine Kostenfunktion berechnet. Die Anpassung entspricht einem Optimierungsproblem, dessen Optimum zumeist durch das Gradientenabstiegsverfahren genähert wird [22]. Entsprechend müssen für das Lernen des Modells noch Kosten- und Optimierungsfunktionen ausgewählt werden.

Seit geraumer Zeit ist die Cross-Entropy-Loss (CEL) der Standardansatz für die Berechnung des Fehlers [22][10]. Dabei werden die berechneten Wahrscheinlichkeiten mit dem tatsächlichen Label verglichen. Je größer der Abstand zwischen den Werten ist, desto höher ist der Fehler. Dabei wird der Fehler logarithmisch berechnet, sodass große Abstände durch einen höheren Fehler bestraft werden als kleine Abstände. Daraus ergeben sich kleinere Werte für ein gutes Modell während schlechte Modelle einen hohen Fehler haben.

$$\sum_{c=1}^{M} y_c - \log(p_c) \tag{7.1}$$

Der Formel 7.1 kann die Berechnung der Kreuzentropie entnommen werden. Dabei bildet die Summe des negativen Logarithmus aller Klassen die Kreuzentropie. Wobei M die Menge aller Klassen ist, y_c die tatsächliche Klasse und p_c der vorhergesagte Wahrscheinlichkeitswert. Durch die Anwendung des Logarithmus haben Fehlklassifizierungen einen großen Einfluss auf den resultierenden Fehler. Deshalb kann die CEL auch auf einem unausgeglichenen Datensatz angewendet werden.

Auf Basis des Fehlers folgt die Fehlerrückführung (Backpropagation) auf jedes Neuron im Netz. Durch die Optimierungsfunktion werden anschließend die Gewichte angepasst, um somit den Fehler zu minimieren. Das geschieht zumeist auf Basis des Gradientenabstiegsverfahren. Die ADAM-Optimierungsfunktion kann für diesen Zweck verwendet werden. Sie unterscheidet sich insofern von dem Gradientenabstiegsverfahren, als dass die Lernrate adaptiv für jedes einzelne Neuron angepasst wird [35]. Aufgrund seiner nachgewiesenen Effektivität im Gegensatz zu anderen Optimierungsverfahren [55][35] wird ADAM für das Lernen des Modells verwendet.

Die Anzahl der Optimierung pro Epoche wird durch die Batch-Size definiert [22]. Lerngeschwindigkeit und Lernstabilität werden durch diesen Parameter maßgeblich beeinflusst [10]. Eine kleinere Batch-Size (< 32) soll regulierend wirken und somit den Generalisierungsfehler reduzieren [42]. Durch häufigere Berechnungen benötigt eine kleine Batch-Size jedoch erhöhte Rechenleistung. Dieser Parameter wird im Rahmen der Modellierung durch den experimentellen Vergleich verschiedener Werte optimiert.

Beim Lernen von Netzen mit einer hohen Kapazität besteht die Gefahr, dass der Validierungsfehler nach einer gewissen Anzahl von Epochen zu steigen beginnt, während der Lernfehler weiterhin sinkt [22]. Ab einer bestimmten Anzahl von Epochen reagiert das Netz also mit einer Überanpassung. Das hat zur Folge, dass sich das Netz von dem Signal entfernt und das Rauschen der Daten erlernt. Durch die Regularisierungsmethode *Early-Stopping* wird das Lernen dann gestoppt, wenn der Validierungsfehler am geringsten ist. Im Gegensatz zu anderen Regularisierungsmethoden, wird beim Early-Stopping während des Lernvorgangs der optimale Wert ermittelt [22]. Für den Fall, dass der Validierungsfehler nicht nur ein Minimum hat, gibt es die *patience*. Diese gibt an, wie viele Epochen weitergelernt wird, nachdem der Validierungsfehler zu steigen begonnen hat. Sobald ein neues Minimum entdeckt wurde, wird die patience zurückgesetzt. Das Lernen endet, sobald der Validierungsfehler zu steigen beginnt und die patience erschöpft ist. Das Early-Stopping regularisiert das Netz, indem es den Parameterraum durch die Reduzierung von Lernschritten eingrenzt.

In den nachfolgenden Experimenten werden die Hyperparameter wie Propagierungsiterationen, Batch-Size und Embedding-Dimension manuell optimiert. Dabei wird das Modell auf Basis der nachfolgenden Konfigurationen aus Hyperparametern solange gelernt, bis das Early-Stopping das Lernen stoppt. Die patience wird fest auf 10 Epochen festgelegt. Dieser Wert ergibt sich aus den Erfahrungen der zwei vorbereitenden Arbeiten [59][60].

7.3 Ergebnisse

Zu Beginn der Experimente wird eine Standardeinstellung der Hyperparameter gewählt, deren Werte im Anschluss durch Stichproben einzeln optimiert werden. Zunächst wird die Batch-Size auf 32 gesetzt, die Dimension der Node-Feature-Embeddings und Knotenrepräsentationen auf 32 und die Anzahl der Iterationen auf 8. Diese Werte sind lediglich die Starteineinstellung und sollen nachfolgend durch stichprobenartige Experimente optimiert werden. Die Embedding-Dimension der Starteinstellung ergibt sich aus einer Faustregel[5]. Sie besagt, dass sich die optimale Dimension eines Embeddings aus der vierten Wurzel der Anzahl der einzubettenden Werte ergibt. Die Größe des Vokabulars der Node-Features im Lerndatensatz umfasst aufgerundet ca. 1 Mio. Werte. Somit ergibt sich aufgerundet ein Wert von $\sqrt[4]{1.000.000} = \lceil 32 \rceil$. Der Wert 32 für die Batch-Size stellt die Grenze zwischen vergleichsweise kleinen und großen Batch-Sizes dar [42] und wird deshalb für die Starteinstellung verwendet. Die Anzahl von 8 Iterationen in der Starteinstellung ergibt sich den Sichtungen einiger Beispiele des Lerndatensatzes. In diesen Beispielen waren relevante Informationen immer in maximal 8 Kanten vom korrespondierenden Knoten entfernt.

Im ersten Schritt wird der Einfluss der Batch-Size auf das Modell gemessen. Zu diesem Zweck wird die Performance des Modells unter Anwendung der Batch-Sizes b=8, b=16, b=32 und b=64 gemessen. In der Tabelle 7.1 sind die Ergebnisse der Durchläufe verzeichnet. Dabei wurde die beste Performance für b=8 mit einem macro F_1-Score von 0,813 nach 18 Epochen, bei b=16 mit einem macro F_1-Score von 0,822 nach 27 Epochen, bei b=32 mit einem macro F_1-Score von 0,826 nach 35 Epochen und bei b=64 mit einem macro F_1-Score von 0,820 nach 37 Epochen erreicht. Der Tabelle 7.1 kann entnommen werden, dass die insgesamt beste Performance unter Verwendung der Batch-Size 32 erreicht wurde. Aus diesem Grund wird für die nachfolgenden Experimente die Batch-Size von 32 verwendet.

[5] https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html

	b = 8 F_1-Score	b = 16 F_1-Score	b = 32 F_1-Score	b = 64 F_1-Score
SystemPrintln	0,998	0,999	0,990	0,993
ImmutableField	0,551	0,538	0,509	0,563
AvoidPrintStackTrace	0,899	0,911	0,866	0,930
UnusedFormalParameter	0,877	0,867	0,829	0,842
SwitchStmtsShouldHaveDefault	0,898	0,980	0,944	0,960
UnusedPrivateField	0,756	0,709	0,689	0,746
ExcessiveParameterList	0,755	0,773	0,807	0,707
UnusedImports	0,978	0,980	0,979	0,978
FinalFieldCouldBeStatic	0,858	0,812	0,830	0,818
UnusedLocalVariable	0,943	0,930	0,940	0,934
OverrideBothEqualsAndHashcode	0,350	0,514	0,650	0,470
UnusedPrivateMethod	0,784	0,783	0,794	0,809
UnusedAssignment	0,738	0,712	0,735	0,725
Flawless	0,999	0,999	0,999	0,999
Macro	**0,813**	**0,822**	**0,826**	**0,820**

Tabelle 7.1: F_1-Score pro Klasse für die Batch-Sizes b=8, b=16, b=32 und b=64

Durch die nachfolgenden Experimente soll die Auswirkung der Dimension d auf die Performance des Modells gemessen werden. Anhand von d wird die Dimension der Node-Feature-Embeddings d und der Knotenrepräsentation d_h gleichermaßen gesteuert. Das separate Steuern der Dimensionen d und d_h würde lediglich die Kapazität des Netzes erhöhen. Aus diesem Grund werden die beiden Hyperparameter nachfolgend gleichgesetzt und mit d beschrieben. In der Tabelle 7.2 ist die Performance unter Verwendung der Dimensionen d=16, d=32, d=64 und d=96 verzeichnet. Dabei wurde die beste Performance basierend auf dem macro F_1-Score bei d=16 mit einem Wert von 0,765 nach 21 Epochen, bei d=32 mit einem Wert 0,826 nach 35 Epochen, d=64 mit einem Wert 0,840 nach 17 Epochen und bei d=96 mit einem Wert 0,836 nach 11 Epochen erreicht. Die Dimension hat einen starken Einfluss auf die Kapazität des Netzes. Bei d=16 besteht das Netz aus insgesamt 10,667 Mio. optimierbaren Parametern, bei d=32 aus 21,959 Mio., bei aus d=64 aus 42,817 Mio. und bei aus d=96 aus 65,742 Mio.. Die insgesamt beste Performance wurde bei d=64 erreicht. Entsprechend wird nachfolgend mit einer Dimension von d=64 weitergearbeitet.

Zuletzt wird die Performance des Netzes unter Anwendung verschiedener Propagierungsiterationen k gemessen. In der Tabelle 7.3 ist die Performance des Modells unter der Anwendung von k=2, k=4, k=8 und k=12 Iterationen gemessen. Im Gegensatz zur Batch-

	d = 16 F_1-Score	d = 32 F_1-Score	d = 64 F_1-Score	d = 96 F_1-Score
SystemPrintln	0,996	0,996	0,999	0,998
ImmutableField	0,521	0,509	0,528	0,544
AvoidPrintStackTrace	0,915	0,866	0,915	0,983
UnusedFormalParameter	0,759	0,829	0,900	0,954
SwitchStmtsShouldHaveDefault	0,930	0,944	0,977	0,990
UnusedPrivateField	0,737	0,689	0,742	0,807
ExcessiveParameterList	0,718	0,807	0,829	0,727
UnusedImports	0,976	0,979	0,973	0,980
FinalFieldCouldBeStatic	0,372	0,830	0,904	0,855
UnusedLocalVariable	0,931	0,940	0,937	0,930
OverrideBothEqualsAndHashcode	0,360	0,650	0,500	0,438
UnusedPrivateMethod	0,733	0,794	0,814	0,794
UnusedAssignment	0,767	0,735	0,745	0,708
Flawless	0,999	0,999	0,999	0,999
Macro	**0,765**	**0,826**	**0,840**	**0,836**

Tabelle 7.2: F_1-Score pro Klasse für die Dimension d=16, d=32, d=64 und d=96

	k = 2 F_1-Score	k = 4 F_1-Score	k = 8 F_1-Score	k = 12 F_1-Score
SystemPrintln	0,998	0,996	0,999	0,998
ImmutableField	0,306	0,507	0,528	0,572
AvoidPrintStackTrace	0,534	0,944	0,915	0,929
UnusedFormalParameter	0,632	0,683	0,900	0,923
SwitchStmtsShouldHaveDefault	0,749	0,914	0,977	0,936
UnusedPrivateField	0,614	0,704	0,742	0,795
ExcessiveParameterList	0,725	0,801	0,829	0,772
UnusedImports	0,972	0,982	0,973	0,978
FinalFieldCouldBeStatic	0,640	0,805	0,904	0,666
UnusedLocalVariable	0,744	0,911	0,937	0,946
OverrideBothEqualsAndHashcode	0,000	0,457	0,500	0,472
UnusedPrivateMethod	0,233	0,805	0,814	0,811
UnusedAssignment	0,168	0,570	0,745	0,796
Flawless	0,999	0,999	0,999	0,999
Macro	**0,594**	**0,791**	**0,840**	**0,828**

Tabelle 7.3: F_1-Score pro Klasse für die Anzahl der Iterationen k=2, k=4, k=8 und k=12

Size oder Dimension, hat die Anzahl der Iterationen einen deutlichen Einfluss auf die Performance. Bei k=2 betrug der macro F_1-Score nach 30 Epochen 0,594, bei $k = 4$ nach 18 Epochen 0,791, bei k=8 nach 18 Epochen 0,840 und bei k=12 nach 20 Epochen 0,828. Im Vergleich zur Dimension hat die Anzahl der Iterationen einen geringen Einfluss auf die Kapazität des Netzes. Die Anzahl der zu optimierenden Parameter beträgt bei k=2 42,664 Mio., k=2 42,715 Mio., bei k=8 42,817 Mio. und bei k=12 42,919 Mio.. Bei einer Anzahl von k=8 Iterationen hat das Modell die insgesamt beste Performance erreicht.

Das endgültige Modell wird entsprechend der vorherigen Experimente mit b=32, d=64 und k=8 für 18 Epochen lang auf dem gesamten Datensatz bestehend aus Lern- und Validierungsdaten gelernt. Die anschließende Bewertung der Performance basiert auf dem Testdatensatz. In Abbildung 7.7 und der Tabelle 7.4 ist die finale Performance des Modells auf den Testdaten verzeichnet. Der finale macro F_1-Score beträgt 0,614, während eine deutliche Diskrepanz zwischen der Performance der einzelnen Klassen zu verzeichnen ist. Ferner ist die Performance im Vergleich zu den Validierungsdaten bei nahezu jeder Klasse geringer. Dabei ist die Differenz von Klasse zu Klasse unterschiedlich. Bei einer Gegenüberstellung der Werte ergeben sich die Differenzen zwischen Validierungs- und Testperformance von 0 für *Flawless* und *SystemPrintln*, von 0,271 für *ImmutableField*, von 0,036 bei *AvoidPrintStackTrace*, von 0,170 bei *UnusedFormalParameters*, von 0,108 bei *SwitchStmtsShouldHaveDefault*, von 0,458 bei *UnusedPrivateField*, von 0,615 bei *ExcessiveParameterList*, von 0,322 bei *UnusedImports*, von 0,474 bei *FinalFieldCouldBeSta-*

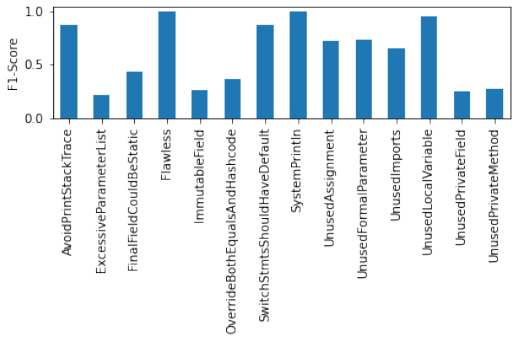

Abbildung 7.7: F_1-Score pro Klasse auf den Testdaten

Klasse	F_1-Score
SystemPrintln	1
ImmutableField	0,257
AvoidPrintStackTrace	0,879
UnusedFormalParameter	0,730
SwitchStmtsShouldHaveDefault	0,869
UnusedPrivateField	0,248
ExcessiveParameterList	0,214
UnusedImports	0,651
FinalFieldCouldBeStatic	0,430
UnusedLocalVariable	0,952
OverrideBothEqualsAndHashcode	0,371
UnusedPrivateMethod	0,270
UnusedAssignment	0,725
Flawless	0,999
Macro	**0,614**

Tabelle 7.4: F_1-Score pro Klasse den Testdaten

tic, von -0,015 bei *UnusedLocalVariable*, von 0,129 bei *OverrideBothEqualsAndHashcode*, von 0,544 bei *UnusedPrivateMethod* und von 0,020 bei *UnusedAssignment*.

In diesem Kapitel wurde die Implementation und das Lernen des Prototyps erläutert. Anschließend wurde die Performance auf Basis eines Testdatensatzes gemessen. Diese Ergebnisse bilden die Grundlage für die anschließende Analyse und Bewertung des Konzepts. Ferner soll aus den Erkenntnissen dieser Experimente eine Antwort auf die Frage, inwiefern sich die Techniken des maschinellen Lernens auf das Problemfeld der klassischen statischen Codenanalyse anwenden lassen, formuliert werden.

8 Analyse und Diskussion

Ziel der Arbeit ist die Untersuchung von ML-Methoden für das Lernen einer Fehleranalyse auf Quellcode. Zu diesem Zweck wurde zunächst eine Auswahl von Fehlern getroffen, die durch ein Modell prognostiziert werden sollen. Anschließend wurden die Methoden der klassischen Codeanalyse vorgestellt, mit denen diese Fehler bisher identifiziert wurden. Das beinhaltete die Erörterung von Merkmalen, die von klassischen statischen Codeanalyse-Programmen verwendet werden, um entsprechende Fehler zu identifizieren. Danach wurden gängige Methoden für das Lernen auf Quellcode vorgestellt. Das beinhaltete die Auswahl einer geeigneten Methode, dem GNN, für diese Problemstellung. Auf diesen Erkenntnissen wurde darauf folgend ein Konzept für das Lernen einer Fehleranalyse entwickelt. Nachfolgend wurde die Implementierung und das Lernen des Modells protokolliert. Die Performance des Modells wurde anschließend durch einen Testdatensatz bewertet. In diesem Kapitel soll das vorgestellte Konzept auf Basis der Ergebnisse analysiert und diskutiert werden. In der Analyse sollen die Ergebnisse aus dem Kapitel 7 analysiert und interpretiert werden. Das beinhaltet Aufarbeitung der Faktoren, die schlussendlich zur ermittelten Performance des Modells geführt haben. In der anschließenden Diskussion werden diese Faktoren diskutiert und mit Optimierungsvorschlägen versehen. Schlussendlich werden das Konzept und die Implementation in der Diskussion herangezogen, um eine Antwort auf die in dieser Arbeit behandelte Problematik zu formulieren.

8.1 Analyse

Die Tabellen 7.1, 7.2 und 7.3 zeigen deutlich den Einfluss der zugehörigen Hyperparameter auf die Performance des Modells.

Die Batch-Size hat Einfluss auf die Anzahl der Optimierungen pro Epoche. Deshalb ist es nicht verwunderlich, dass das Optimum bei niedrigeren Batch-Sizes nach vergleichsweise

weniger Epochen erreicht wurde. Weiterhin zeigt die Tabelle 7.1, dass die Batch-Size nur einen minimalen Einfluss auf den macro F_1-Score hat. Vielmehr scheinen die stärksten Schwankungen in der Performance bei der Klasse *OverrideBothEqualsAndHashcode* vorzuliegen. Das muss jedoch nicht zwangsläufig an der Batch-Size liegen. Es ist anzunehmen, dass diese Schwankungen durch einen unruhigen Lernverlauf ausgelöst werden können. Die Batch-Size hat also hauptsächlich einen Einfluss auf die Anzahl der zu lernenden Epochen. Ein regularisierender Effekt ist bei kleineren Batch-Sizes nicht zu erkennen.

Die Dimension hat maßgeblich Einfluss auf die Kapazität des Netzes. Laut den Ergebnissen in Tabelle 7.2 wirkt sich eine zu geringe oder zu hohe Dimension negativ auf die Performance des Netzes aus. Wobei eine großer Unterschied lediglich bei einer Dimension von $d = 16$ zu erkennen ist. Dabei scheinen auch nur die Klassen *FinalFieldCouldBeStatic* und *OverrideBothEqualsAndHashcode* wesentlich an Performance einzubüßen. Ab einer Dimension von d=32 sind die Unterschiede der Performance eher gering. Eine höhere Kapazität verlangsamt das Lernen und erhöht die Gefahr der Überanpassung. Deshalb ist es fraglich, ob bei der Auswahl zwischen d=32, d=64 und d=96 der macro F_1-Score tatsächlich die beste Metrik für die Entscheidung gewesen ist.

Die Anzahl der Propagierungsiterationen hat im Vergleich zur Dimension einen verschwindend geringen Einfluss auf die Kapazität. Der Tabelle 7.3 kann der Performanceunterschied zwischen den verschiedenen Iterationsanzahlen entnommen werden. Es ist ersichtlich, dass eine Anzahl von k=2 Iterationen für die meisten Klassen keine ausreichende Informationsverteilung bietet. Lediglich *SystemPrintln* und *UnusedImports* scheinen in dieser Konfiguration bereits ausreichende Informationen von den Nachbarknoten zu erhalten. Interessant ist jedoch auch die Tatsache, dass diese Klassen auch durch die Zunahme von Iterationen nicht an Performance einbüßen. Das Modell scheint hier also wichtige Merkmale über mehrere Iterationen beibehalten zu können. Das kann durch den Einbezug von Jumping-Connections erklärt werden. Hierdurch verwendet das Modell schließlich die Hidden-Embeddings jeder Iteration für die Erstellung der Knotenrepräsentation. Die beste Performance wurde mit k=8 Iterationen erreicht. Wobei der Unterschied zwischen k=8 und k=12 sehr gering ist. Es ist anzunehmen, dass durch mehr Iterationen tendenziell eine bessere Performance erzielt wird.

Die zuvor besprochenen Hyperparameter wurden in dieser Arbeit manuell durch Stichproben ermittelt. Es gibt jedoch auch verschiedene Methoden wie z.B. *genetische Alogrithmen*, *Random-Search* oder *Grid-Search*, um den Computer die optimale Kombination der Parameter bestimmen zu lassen. Diese Verfahren sind aber sehr rechenintensiv, da eine

Vielzahl an möglichen Kombination ausgeführt werden muss [22]. Sie können dennoch zu einer besseren Konfiguration des Modells führen.

Die Performance zwischen Validierungs- und Testdaten in Tabelle 7.4 und 7.3 weist eine große Diskrepanz auf. Vor allem die Klassen *ImmutableField*, *UnusedPrivateField*, *ExcessiveParameterList*, *UnusedImports* und *UnusedPrivateMethod* zeigen eine große Differenz. Klassische Gründe für eine schlechte Performance sind sowohl eine zu hohe Kapazität des Netzes, gefolgt von einer Überanpassung des Modells als auch die Unausgeglichenheit der Lerndaten.

Eine Überanpassung des Modells sollte sich in der Regel auf den Validierungsdaten abzeichnen. Wie bereits erwähnt, tendieren die Validierungsdaten dazu, den Generalisierungsfehler zu unterschätzen. Eine gewisse Diskrepanz zwischen Validierungs- und Testdaten ist daher zu erwarten. Die vorliegende Diskrepanz ist jedoch sehr hoch und somit wahrscheinlich auf verschiedene Faktoren zurückzuführen. Einer der Faktoren ist eine suboptimale Auswahl der Validierungsdaten. Mithilfe der Validierungsdaten soll einen Schätzwert für die Performance auf unbekannten Daten produziert werden. Die Validierungsdaten werden jedoch aus einem Teil der Lerndaten gebildet und stammen somit aus den gleichen Repositories wie die Lerndaten. Das kann zur Folge haben, dass sich Bezeichner und Programmstrukturen in Lern- und Validierungsdaten sehr ähneln. Hieraus kann ein geringer Validierungsfehler resultieren. Wenn sich Lern- und Validierungsdaten zu sehr gleichen, kann dies auch auf einen zu geringen Datensatz zurückgeführt werden. Ein größerer Datensatz beinhaltet eine größere Variation an Beispielen und erschwert somit das Anpassen an einige wenige Merkmale der Lerndaten.

Ein weiterer Faktor für die Diskrepanz kann die Unausgeglichenheit der Lerndaten darstellen. Wenn ein hoher Unterschied in der Anzahl der Beispiele für die verschiedenen Klassen vorliegt, dann spricht man von einem unausgeglichenen Datensatz. Das kann im Anschluss zu einer schlechten Performance führen. Für die Untersuchung des Zusammenhangs zwischen der Anzahl der Beispiele einer Klasse und der Performance ist in Abbildung 8.1 eine Gegenüberstellung der Labelverteilung der Lern- und Validierungsdaten mit der Performance des Modells auf den Testdaten dargestellt. Es ist ersichtlich, dass keine der Klassen mit einer hohen Diskrepanz durch eine verhältnismäßig geringe Anzahl an Beispielen auffällt. Es ist jedoch möglich, dass für die Komplexität der Klassen *ImmutableField*, *UnusedPrivateField*, *ExcessiveParameterList*, *UnusedImports* und *UnusedPrivateMethod* nicht ausreichend Lerndaten vorhanden gewesen sind. Ein Bezug

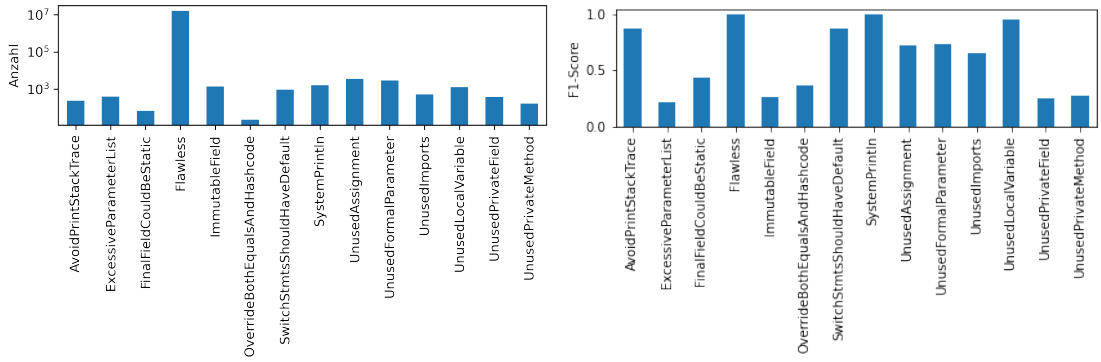

Abbildung 8.1: Gegenüberstellung Labelverteilung der Lerndaten mit der Performance auf den Testdaten

zwischen der Anzahl der Klassen und der resultierenden Performance ist also nur bedingt aussagekräftig.

Die Gründe, die bei einem Modell zu einer Klassifizierung geführt haben, sind nahezu unmöglich nachzuvollziehen. Die Verkettung vieler nicht-linearer Funktionen in vielschichtigen Deep Learning-Architekturen macht sie hochgradig intransparent und bieten diesbezüglich keine gute wissenschaftliche Fundierung. Das Modell ist eine Art *Black-Box*, die keine Rückschlüsse darüber zulässt, welche Merkmale der Eingabe ausschlaggebend für die Entscheidungsfindung waren. Erklärbarkeit von maschinellem Lernen ist nicht Gegenstand dieser Arbeit. Allerdings gibt es in diesem Bereich einige wissenschaftliche Ansätze, die diese Frage untersuchen [46].

Durch das Heranziehen von Stichproben sollen nachfolgend jedoch Erklärungsversuche für die Gründe einer entsprechenden Klassifikation formuliert werden. Ausgangspunkt für diese Analyse bildet die in Abbildung 8.2 dargestellte Confusion-Matrix der Testdaten. Sie stellt auf der Y-Achse die tatsächlichen Klassen und auf der X-Achse die vorhergesagten Klassen dar. Alle Zahlen in der Diagonale von oben links nach unten rechts stehen somit für korrekt klassifizierte Beispiele. Alle weiteren Zahlen stehen für eine Fehlklassifizierung. Diese Grafik stellt für jede Klasse dar, wie viele Knoten durch das gelernte Modell der korrespondierenden Klasse zugeordnet wurden und wie viele basierend auf der Ground-Truth tatsächlich zur Klasse gehören. Die True-Positives der Klasse *Flawless* werden in der Abbildung abgekürzt dargestellt. Der entsprechende Wert in der Grafik wird 16,30 M. steht für 16.300.369. Nachfolgend werden die einzelnen Klassen anhand der Kennzahlen precision und recall untersucht. Diese Hintergründe zu dieser Metrik

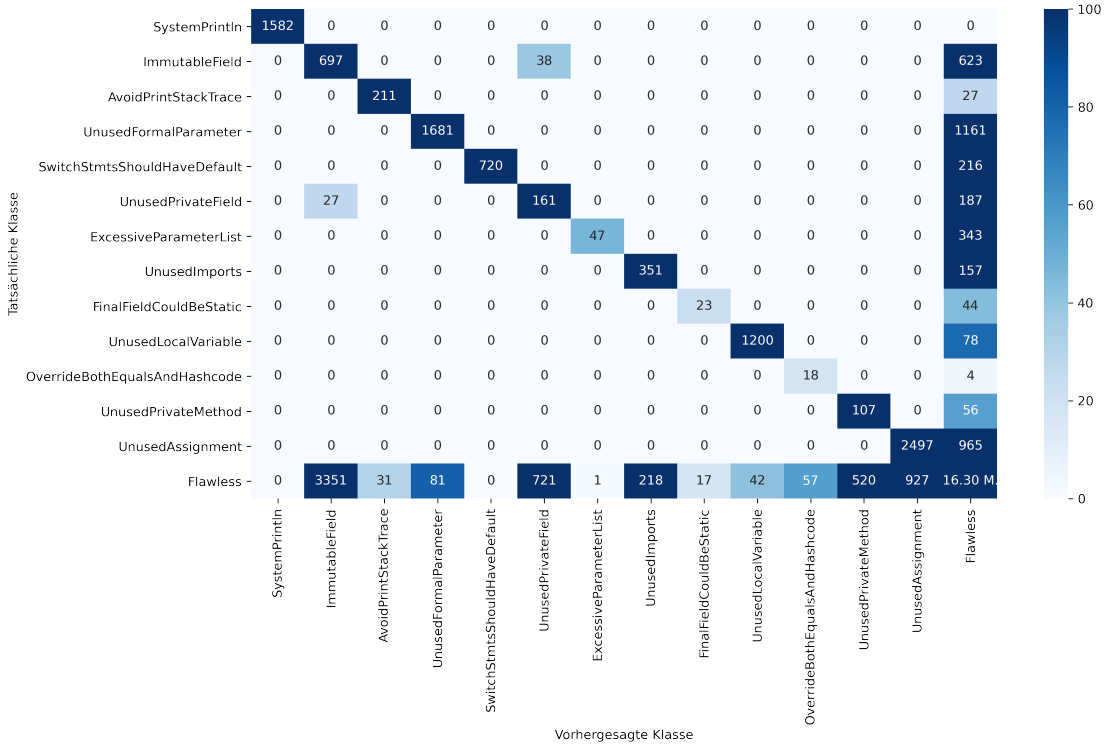

Abbildung 8.2: Confusion-Matrix der Testdaten

wurden bereits in Kapitel 4 erläutert. Sie bieten für eine detaillierte Analyse eine bessere Grundlage als der F_1-Score.

Auf eine detailliertere Untersuchung der Klasse *SystemPrintln* wird an dieser Stelle verzichtet. Alle Beispiele in den Testdaten wurden korrekt klassifiziert. Das Modell scheint für diese Klasse das Signal erkannt zu haben.

Weitaus geringer ist die Performance der Klasse *ImmutableField*. Von insgesamt 1.350 Knoten, die tatsächlich der Klasse *ImmutableField* zugeordnet sind, wurden 697 als *ImmutableField*, 38 als *UnusedPrivateField* und 623 als *Flawless* klassifiziert. Das entspricht einem recall von 0,53. Ferner wurden *3351* der tatsächlich mit *Flawless* annotierten Knoten von dem Modell als *ImmutableField* identifiziert. Das entspricht einer precision von 0,17. Außerdem hat diese Klasse eine Diskrepanz des F_1-Scores zwischen Validierungs- und Testdaten von 0,271. Die Fehlklassifizierung zwischen *UnusedPrivateField* und *ImmutableField* sind auf die Vereinfachung des Klassifizierungsproblems zurückzuführen. Eine Feldvariable kann schließlich beide Fehler aufweisen. Die Multi-Class-Klassifizierung kann jedem Knoten jedoch nur eine Klasse zuweisen, sodass es hier zu einer Fehlklassifi-

kation kommt. Das größte Problem bei dieser Klasse ist aber, dass Knoten, die *Flawless* sind, fälschlicherweise als *ImmutableField* deklariert wurden. Durch die Begutachtung einiger Stichproben der Testdaten konnten zwei Problemfaktoren bei der Klassifikation festgestellt werden. Um das zu verdeutlichen, sind in Listing 8.1 und 8.2 zwei Beispiele abgebildet.

```
public class UnvoteOnComment{
 private Long commentId;

 public Long getCommentId() {
  return commentId;
 }
}
```

Listing 8.1: *commentId* ist tatsächlich ein *ImmutableField*

```
public class UnvoteOnComment{
  @Data
  private Long commentId;

  public Long getCommentId() {
   return commentId;
  }
}
```

Listing 8.2: *commentId* ist kein *ImmutableField*

In beiden Codebeispielen klassifiziert das Modell die Feldvariable *commentId* als *ImmutableField*. Laut *PMD* ist jedoch nur die Feldvariable in Listing 8.1 tatsächlich eine *ImmutableField*. Die Regel von PMD zum Auffinden von *ImmutableField*-Fehlern ignoriert alle Feldvariablen, die unter dem Einfluss bestimmter Annotationen wie z.B. *Data* stehen. Dieser komplexe Zusammenhang wurde von dem Modell nicht gelernt. Eine weitere Kante, die den Abstand zwischen Annotation und Feldvariable reduziert, könnte an dieser Stelle zu einer Verbesserung der Performance führen. In den Testdaten weisen die meisten False-Negatives eine lesende Referenz auf die Feldvariable auf. Das lässt vermuten, dass das Modell den Unterschied zwischen verwendenden und schreibenden Referenzen auf Feldvariablen nicht erfassen konnte. Eine Aufteilung der *declares* Kante in zwei Kanten, bei denen eine für lesende und eine für schreibende Zugriffe steht, könnte dieses Problem reduzieren.

Das Modell hat von insgesamt 238 Knoten mit der tatsächlichen Klasse *AvoidPrintStackTrace* 211 korrekt und 27 als *Flawless* klassifiziert. Hinzu kommen 31 Knoten, die eigentlich *Flawless* sind, aber fälschlicherweise als *AvoidPrintStackTrace* identifiziert wurden. Das ergibt eine precision von 0,87 und einen recall von 0,88. Die Diskrepanz der Performance für diese Klasse zwischen Validierungs- und Testdaten liegt bei 0,036. Laut Goodfellow et. al. [22] kann in solchen Fällen das Vergrößern des Lerndatensatzes helfen. Es kann nämlich sein, dass im Lerndatensatz nicht ausreichend Beispiele vorhanden gewesen sind.

Die Ergebnisse für die Klasse *UnusedFormalParameter* führen zu einer precision von 0,95 und einen recall von 0,59. Die Diskrepanz des F_1-Scores zu den Validierungsdaten beträgt 0,170. Ein Erklärungsversuch für diese Performance wird durch den Beispielcode in Listing 8.3 demonstriert.

```
1 public class UnusedFormalParameter {
2  public boolean main(String input) {
3   return false;
4  }
5  public boolean check(String input) {
6   return false;
7  }
8  public void check(Integer input) { }
9 }
```

Listing 8.3: Beispielcode für die Klassifizierung von *UnusedFormalParameter*. Der unterstrichene Parameter wurde positiv klassifizert.

Während durch PMD jeder der Parameter als *UnusedFormalParameter* klassifiziert wurde, hat das gelernte Modell nur den unterstrichenen Parameter *input* in Zeile 5 als *UnusedFormalParameter* klassifiziert. Die Parameter *input* in Zeile 2 und 5 unterscheiden sich lediglich durch den Namen der umfassenden Methode. Das legt den Schluss nahe, dass das Modell die Bezeichner der Methoden für eine Klassifizierung verwendet. Die Parameter *input* in Zeile 5 und 8 unterscheiden sich lediglich durch den Rückgabewert der Methode. Auch hier kann vermutet werden, dass dieses Merkmal die Klassifikation des Modells beeinflusst hat. Der Einbezug von Bezeichnern als Node-Feature gewährt dem Modell die Freiheiten, diese als Merkmal für die Analyse zu verwenden. Dieses Verhalten bietet auch eine Erklärung für die Diskrepanz zwischen Test- und Validierungsdaten. Eine Verbesserung kann an dieser Stelle durch das normalisieren der Bezeichner vorgenommen werden. Ferner kann ein größerer Lerndatensatz dieses Problem reduzieren. Es bleibt jedoch festzuhalten, dass das Modell in 1.681 Fällen die richtige Entscheidung getroffen hat. Das könnte Vermuten lassen, dass nicht nur Bezeichner und Rückgabewert als Merkmal verwendet werden. Andernfalls würde die Analyse auf Basis der Bezeichner und Rückgabewerte eine überraschend hohe Trefferquote vorweisen.

Für die Klasse *SwitchStmtsShouldHaveDefault* ergibt sich auf Basis der Testdaten eine precision von 1 und ein recall von 0,76. Das Modell ist also in der Lage, nur Knoten als *SwitchStmtsShouldHaveDefault* zu klassifizieren, die auch tatsächlich der Klasse zugehören. Jedoch werden auch viele Knoten als negativ klassifiziert, obwohl sie eigentlich positiv sind. Die Überprüfung von Stichproben aus den Testdaten lässt auch hier wie-

der das Problem eingrenzen. In diesem Fall lässt sich das Problem nur an der IR der

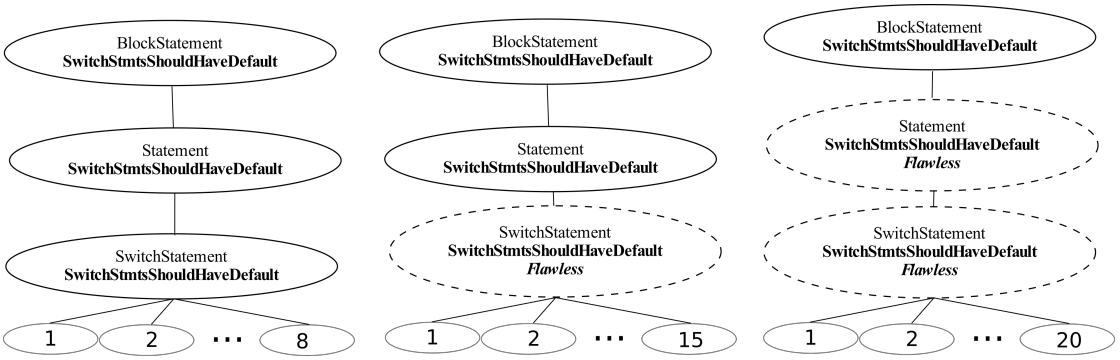

Abbildung 8.3: Skizze der Klassifizierung durch das Modell anhand einer Teildarstellung der IR

Codebeispiele erkennen. In der Abbildung 8.3 sind relevante Knoten der IR von drei Switch-Statements abgebildet. Ein Switch-Statement besteht in der IR mindestens aus drei Knoten mit den Node-Feautures *BlockStatement*, *Statement* und *SwitchStatement*. Unter den Node-Features ist in jedem der Knoten sowohl die tatsächliche Klasse als auch die vorhergesagte Klasse abgebildet. Wenn sich beide gleichen, dann werden tatsächliche und vorhergesagte Klasse durch ein Wort fettgedruckt dargestellt. Wenn ein Unterschied zwischen vorhergesagter und tatsächlicher Klasse besteht, dann wird der Knoten gestrichelt dargestellt und die vorhergesagte Klasse in kursiv und fett abgebildet. Die Knoten mit den Zahlen stehen für eine entsprechende Anzahl an Case-Einträgen. Das Labelverfahren führt dazu, dass dieser Fehler immer auf drei Knoten abgebildet wird. Es ist ersichtlich, dass bei einer erhöhten Anzahl von Switch-Einträgen weniger Knoten mit *SwitchStmtsShouldHaveDefault* klassifiziert werden. Durch eine Veränderung des Labelverfahrens kann dieses Problem reduziert werden. Wenn jeder Fehler immer genau auf einen Knoten abgebildet wird, dann reduziert das die Komplexität der Klassifikation und kann zu einer Erhöhung der Performance führen.

Die Performance des Modells für die Klasse *UnusedPrivateField* weist Parallelen zur Klasse *ImmutableField* auf. Mit einer precision von 0,175 und einem recall von 0,43 zeigt sich auch hier das Problem einer hohen False-Positive Rate. Die Klasse *UnusedPrivateField* wird während der Analyse von PMD in Abhängigkeit der Annotationen identifiziert. Wie bei der Klasse *ImmutableField* scheint das auch in diesem Fall zu Problemen geführt haben. Jedoch ist durch eine stichprobenartige Überprüfung der Testdaten ein weiteres Problem aufgefallen. In Listing 8.4 ist diesbezüglich ein Beispielcode zu sehen. Das Mo-

dell hat nur die Feldvariable *id* als *UnusedPrivateField* deklariert. In der Ground-Truth handelt es sich jedoch bei allen drei Variablen um nicht verwendete Feldvariablen.

```
1 class UnusedPrivateField {
2  private Integer id;
3  private Integer dummy;
4  private Integer dieserBezeichnerIstUnbekannt;
5 }
```

Listing 8.4: Codebeispiel für die *UnusedPrivateField* Klassifizierung

Der einzige Unterschied zwischen den Feldvariablen ist der Bezeichner. Das lässt vermuten, dass sich auch diese Klassifikation auf den Variablen-Bezeichner basiert.

Mit einer precision von 0,97 und einem recall von 0,12 erkennt das Modell nur wenige Beispiele, die in der Ground-Truth der Klasse *ExcessiveParameterList* zugeordnet sind. Bei den meisten korrekten Klassifizierung betrug die Parameteranzahl weitaus mehr als 10 Parameter. In der Ground-Truth sind jedoch alle Methoden mit einer Anzahl ab 10 Parametern als *ExcessiveParameterList* gelabelt. Das Modell scheint die Identifikation der Parameteranzahl nicht gelernt zu haben. Diese Eigenschaft kann auf die Auswahl der Normalisierungsmethode für das GNN zurückgeführt werden. Durch die Normalisierung bei der Aggregation der Nachrichten propagiert das Netz hauptsächlich inhaltliche Informationen.

```
public class ExcessiveParameterList {
 int methode1(int a, int b, int c, int d, int e, int f, int g, int h, int i, int j, int
      k, int l) {
  return a + b + c + d + e + f + g + h + i + j;
 }
 void method2(int a, int b, int c, int d, int e, int f, int g, int h, int i, int j, int
      k, int l) {
 }
}
```

Listing 8.5: Codebeispiel für die *ExcessiveParameterList* Klassifizierung

Die Begutachtung der Stichproben bietet jedoch weitere Anhaltspunkte. Dies soll durch ein Beispiel in Listing 8.5 veranschaulicht werden. Das Modell hat die Methode mit dem Namen *methode1* positiv und die Methode mit dem Namen *methode2* negativ klassifiziert. Es ist jedoch offensichtlich, dass beide Klassen zu viele Parameter haben und entsprechend positiv klassifiziert werden sollten. Der Unterschied zwischen den beiden

Methoden ist der Rückgabetyp von *methode1* und die Referenzierung der meisten Parameter im Methodenkörper. Es scheint, als ob die Klassifizierung auf dem Rückgabewert der Methode und den Parameterreferenzierungen basiert.

Mit einer precision von 0,61 und einem recall von 0,69 hat das Modell bei der Klassifizierung von *UnusedImports* gleichermaßen Probleme False-Positive und False-Negative vorhersagen. Die Diskrepanz zwischen Test- und Validierungsdaten liegt bei 0,322. Auf Basis verschiedener Stichproben hat sich abgezeichnet, dass sich die Klassifizierung von *UnusedImports* durch die Veränderung von Annotationen an Methoden in der Java-Klasse verändern lässt. Es liegt die Vermutung nahe, dass sich die Analyse sehr stark an die Programmkonstrukte der Lern- und Validierungsdaten angepasst hat. Das würde die Diskrepanz der Performance zwischen Test- und Validierungsdaten erklären und lässt auf eine suboptimale Auswahl der Validierungsdaten und einen zu kleinen Lerndatensatz schließen.

Bei der Analyse der Klasse *FinalFieldCouldBeStatic* ist ersichtlich, dass mit einer precision von 0,57 und einem recall von 0,34 generell Probleme bei der Klassifikation bestehen. Die Überprüfung von Stichproben zeigen, dass sich das Modell bei der Klassifizierung von *FinalFieldCouldBeStatic* auf die Referenzierung der korrespondierenden Feldvariable stützt. Dabei werden Feldvariablen, die von PMD als *FinalFieldCouldBeStatic* klassifiziert werden, von dem Modell als *Flawless* klassifiziert, weil sie in der Klasse referenziert werden. Das kann an einer zu geringen Lerndatenmenge für diese Klasse liegen. Die Diskrepanz zu der Performance auf den Lerndaten lässt sich ebenfalls hierdurch begründen. Das Modell hat das Rauschen der Lerndaten gelernt.

Die Performance auf der Klasse *UnusedLocalVariable* mit einer precision von 0,96 und einem recall von 0,933 ist auf den Testdaten besser als auf den Validierungsdaten. Das lässt darauf schließen, dass das Modell zumindest ein Teil des Signals erkennen konnte.

Die Klasse *OverrideBothEqualsAndHashcode* zeichnet auf Test- und Validierungsdaten ein gleiches Bild ab. Mit einer precision von 0,25 und einem recall von 0,81 produziert die Analyse eine hohe Anzahl an False-Positives. Durch eine stichprobenartige Untersuchung der Testdaten zeichnet sich das Bild ab, dass das Modell Java-Klassen in denen nur die *equals*-Methode vorhanden ist, zuverlässig als *OverrideBothEqualsAndHashcode* erkennt. Java-Klassen, in denen nur die *hashcode*-Methode vorhanden ist, wurden jedoch als Fehlerfrei angesehen werden. Dieses Problem kann ebenfalls auf eine zu geringe Anzahl an Lerndaten zurückgeführt werden.

Die Performance der Klasse *UnusedPrivateMethod* zeichnet eine große Diskrepanz zwischen Test- und Validierungsdaten. Auch durch die Analyse von Stichproben konnte kein Muster erkannt werden, nach dem das Modell die Codeelemente klassifiziert. Durch die hohe Diskrepanz kann auch hier darauf geschlossen werden, dass sich das Modell zu sehr an die Lerndaten angepasst hat. Entsprechend kann auch hier wieder die Vergrößerung des Testdatensatzes und ein besser Auswahl der Validierungsdaten Abhilfe leisten.

Die Performance auf der Klasse *UnusedAssignment* zeichnet mit einer precision von 0,73 und einem recall von 0,72 eine ähnliche Quote bei den False-Positives und False-Negatives. In der IR werden alle Knoten, die auf die rechte Seite einer Wertzuweisung abbilden, mit der entsprechenden Klasse gelabelt. Für einen entsprechenden Fehler im Quellcode werden also alle Knoten, die auf der rechten Seite einer Wertezuweisung stehen, mit dem Label *UnusedAssignment* versehen. Die Überprüfung von Stichproben der Testdaten haben ergeben, dass sich in den 927 False-Negative Klassifizierungen viele Beispiele befinden, in denen nur eine Untermenge dieser Knoten korrekt klassifiziert wurden. Wie bei der Klassifizierung von *SwitchStmtsShouldHaveDefault* kann auch hier die Reduzierung der gelabelten Knoten auf einen festen Knoten eine Verbesserung der Performance erwirken.

8.2 Diskussion

Die Analyse zeigt auf, dass das Konzept und die Implementation an mehreren Stellen Optimierungspotential aufweisen. Ein großer Aspekt lässt sich aus der Performance-Diskrepanz zwischen Test- und Validierungsdaten ableiten. Dabei zeichnet der Performanceunterschied der Klassen *ImmutableField*, *UnusedPrivateField*, *ExcessiveParameterList* und *UnusedImports* das Bild einer Überanpassung. Neben einer potenziell zu hohen Kapazität lässt sich dieses Ergebnis jedoch auch auf den zu kleinen Lerndatensatz zurückführen. Wenn der Datensatz zu klein ist, kann es passieren, dass sich das Netz an das Rauschen der Daten anpasst. Grund hierfür ist eine zu hohe Gleichheit der Beispiele, sodass das Modell die Beispiele auswendig lernt und nicht das zugrundeliegende Signal der Daten erkennt. Eine Vergrößerung des Lerndatensatzes führt zu mehr Beispielen mit unterschiedlichen Merkmalen. Das reduziert entsprechend die Wahrscheinlichkeit, dass sich das Modell an einige wenige Beispiele anpasst. Ferner scheinen sich Lern- und Validierungsdaten zu sehr zu gleichen. Ein größerer Lerndatensatz, aus dem die Validierungsdaten abgespalten werden, könnte auch hier durch eine größere Varianz in den Beispielen

zu einem besseren Schätzwert der Generalisierung führen. Neben der Vergrößerung des Lerndatensatzes gibt es verschiedene Regularisierungsmethoden, die einer Überanpassung entgegenwirken können. Bei der Regularisierung handelt es sich um Verfahren den Generalisierungsfehler unter Hinnahme eines höheren Lernfehlers zu reduzieren [22]. Im Bereich der Regularisierungen gibt es über das Early-Stopping und den Dropout hinaus eine Vielzahl an Methoden für die Optimierung der Generalisierung eines Modell.

Neben der Erhöhung der Lerndaten bieten die Erkenntnisse der Analyse weitere Ansatzpunkte für eine Optimierung des Modells. Dabei handelt es sich hauptsächlich um die Überarbeitung einiger Designentscheidungen. Unter anderem sind einige wichtige Merkmale in der IR zu weit entfernt von dem zu klassifizierenden Knoten. Bei den Klassen *UnusedPrivateMethod* und *ImmutableField*, in denen eine komplexe Abhängigkeit zum Konstruktor der Klasse und den Annotationen an der Feldvariable gelernt werden muss, lässt sich dieses Problem erkennen. Sowohl die Zusammenstellung der Kanten als auch die Auswahl der AST-Repräsentation beeinflussen dieses Problem maßgeblich. So scheinen die Kanten *child*, *declares* und *reaches* nicht ausreichende Verbindungen zwischen den Elementen im AST herzustellen. Das Hinzufügen mehrerer Kanten führt im Falle eines Multi-Relationen-GNN jedoch auch zur Erhöhung der Kapazität und somit zu einer erhöhten Überanpassungsgefahr. Neben den Kanten bestimmt die Auswahl der AST-Repräsentation die Abstände zwischen den einzelnen Knoten. So zergliedert JDT den Quellcode in insgesamt 120 unterschiedliche Knotentypen, während Alternativen wie JavaParser [61] lediglich aus 84 Knotentypen besteht. Das hat letztendlich Einfluss auf die Knotenanzahl der IR und somit auch auf die Abstände zwischen den einzelnen Knoten. Außerdem führen weniger Knoten zu weniger unnötigen Informationen und schlussendlich zu einer Reduzierung des Rauschens in den Daten. Neben der Topologie der IR haben die Node-Features maßgeblichen Einfluss auf die Performance. Dabei ist während der Analyse vor allem die Entscheidung, Bezeichner als Node-Feature zu verwenden, mehrmals negativ aufgefallen. Es hat schlussendlich dazu geführt, dass unter anderem die Erkennung des Fehlers *UnusedPrivateField* von dem Bezeichner der Variable beeinflusst wird. Der zuvor angesprochene Vorteil, komplexe Zusammenhänge in den Daten zu erkennen, scheint sich in diesem Fall als Nachteil herausgestellt zu haben. Ferner wird das Vokabular der Node-Features durch die Hinzunahme der Bezeichner extrem vergrößert, was sich im Endeffekt durch die Verwendung einer Embedding-Schicht auf die Kapazität des Modells auswirkt. In Verbindung mit einem zu kleinen Lerndatensatz verfügt das Modell dann über zu viel Freiheiten für eine Überanpassung. Neben dem Verbesserungspotential lässt die Testperformance des Modells aber auch positive Rückschlüsse zu. So

scheinen das Modell für die Klassen *SystemPrintln*, *AvoidPrintStackTrace* und *UnusedLocalVariable* das Signal der Daten einigermaßen erfolgreich erkannt zu haben. Das zeigt, dass das vorgestellte Konzept durchaus in der Lage ist, die Identifizierung verschiedener Fehler im Quellcode zu erlernen. Vielmehr wurde in der Analyse anhand verschiedener Stichproben nachgewiesen, dass sich die erkannten Fehler mithilfe die Knoten-ID auf die entsprechenden syntaktischen Elemente im Quellcode zurückführen lassen.

Neben der IR hat auch die Struktur des neuronalen Netzes einen Einfluss auf die Performance des Modells. Durch die Auswahl des GNNs wurde das induktive Lernen ermöglicht. Also die Klassifikation von Daten, die nicht im Lerndatensatz vorhanden gewesen sind. Das wird jedoch erst durch die Kompensation unbekannter Node-Feature ermöglicht. Während der Analyse sind die unbekannten Node-Features nicht als gravierendes Problem aufgefallen. Es scheint so, als ob das Vorgehen also ein valider Ansatz für die Kompensation von unbekannten Node-Features ist. Die Auswahl des RGCN basierte maßgeblich auf der Einbeziehung verschiedener Kantentypen. Die Bewertung des Einflusses der Kantentypen auf die Performance des Modells ist nicht Bestandteil dieser Arbeit. Deshalb bleibt für nachfolgende Arbeiten die Frage, inwiefern die Verwendung einen von Kantentypen einen Einfluss auf die Performance des Modells hat. Ferner bietet der Forschungsbereich der GNNs auch Alternativen. So kann z.B. durch die Verwendung von *Graph Attention Networks* [66] die Wichtigkeit der jeweiligen Verbindungen gelernt werden. Durch die Vereinfachung des Klassifizierungsproblems zu einer Multi-Class-Klassifizierung wurde das Modell auf die Identifikation eines Fehlers pro Knoten begrenzt. Das entspricht jedoch nicht einem realistischen Szenario. Deshalb bleibt für zukünftige Arbeiten die Überprüfung der Machbarkeit einer Multi-Label-Klassifikation für das vorgegebene Problem. Die Herausforderungen bei der Implementation einer Multi-Label-Klassifikation sind Bestandteil aktueller Forschungen. Herrera et. al. [28] bieten einen Überblick über aktuelle Methoden.

Der Lerndatensatz bietet, mal abgesehen von der Größe, noch weiteres Optimierungspotential. Während der Analyse ist aufgefallen, dass die Performance einiger Klassen unter dem Label-Mechanismus leidet. Schließlich führt dieser dazu, dass mehrere benachbarte Knoten durch einen einzelnen Fehler mit der gleichen Fehlerklasse belegt werden. Das liegt daran, dass in der AST-Repräsentation von JDT mehrere Knoten auf denselben Bereich im Quellcode abbilden. Ein Label-Mechanismus, der einen Fehler immer nur genau einen Knoten zuweist, könnte die Performance des Modells optimieren. Schließlich reduziert es die Komplexität der Klassifizierung.

Eines der zu Beginn der Arbeit angeführten Vorteile einer gelernten Codeanalyse besteht daraus, dass das Modell stetig von neuen Daten lernen kann. Das Weiterlernen eines künstlichen neuronalen Netzes sowie das Hinzufügen von neuen Klassen sind Bestandteil aktueller Forschung. Parisi et. al. [51] bieten einen Überblick über aktuelle Herausforderungen und Lösungen in diesem Bereich. Weiterhin wurde zu Beginn der Arbeit angeführt, dass ein gelerntes Modell komplexe und sinnvolle Zusammenhänge zwischen Fehlerindikatoren erkennen kann. Die Analyse des Prototyps lässt vermuten, dass tatsächlich komplexe Zusammenhänge gelernt wurden. Leider waren diese Zusammenhänge nur in den Lern- und Validierungsdaten gültig und haben somit zu einer schlechten Generalisierung geführt. Auslöser hierfür ist der zu kleine Lerndatensatz in Verbindung mit einer hohen Modellkapazität. Damit neue und sinnvolle Zusammenhänge in den Daten gelernt werden können, die tatsächlich eine Codeanalyse bereichern, müssen neue Label-Mechanismen etabliert werden. Die Verwendung von Codeanalyseprogrammen zum Labeln der Lerndaten bietet zwar die Möglichkeit, dass die Regeln verschiedener Programme in einem Modell gelernt werden. Aber für das Erkennen bisher unbekannter Zusammenhänge von Codeeigenschaften, müssen weitere Quellen verwendet werden. Wie Eingangs erwähnt können Methoden aus dem Forschungsgebiet *Mining Software Repositories*[1] hier Abhilfe leisten. So ist es möglich, auf Basis von Commit-Nachrichten oder Code-Kommentaren fehlerhafte Codeabschnitte zu annotieren und die gelernte Analyse somit durch vom Menschen verfasste Eingaben zu optimieren [34][41].

Ein bisher ungenannter Nachteil eines künstlichen neuronalen Netzes für die Analyse von Fehlern im Quellcode ist die intransparente Entscheidungsfindung. Wie bereits erwähnt, ist das Nachvollziehen der Faktoren, die zu einer Klassifizierung geführt haben, nahezu unmöglich. Es gibt in diesem Bereich zwar einige wissenschaftliche Ansätze, die diese Frage untersuchen [46], jedoch ist die klassische statische Codeanalyse in diesem Punkt stets im Vorteil.

Schlussendlich bietet das vorgestellte Konzept einen ersten Ansatz für das Lernen von Fehlern im Quellcode. Mit dem Konzept wurde ein Ansatz vorgestellt, der in seinem jetzigen Zustand für einige Fehlertypen bereits eine erfolgreiche Klassifikation von Fehlern im Quellcode durchführen kann. Durch die vorgestellte IR lassen sich die semantischen Zusammenhänge der Elemente im Quellcode modellieren. Vielmehr ermöglicht die Kombination der IR und die Klassifikation des Modells das Zurückführen der Fehler auf die entsprechenden Codeelemente. Ferner bietet das Konzept, unter der Voraussetzung einiger Optimierungen, das Potenzial, auch für weitere Fehlerklassen eine akzeptable Feh-

[1] http://www.msrconf.org/

lererkennung durchzuführen. Letztendlich ist jedoch auch die Datenvorbereitung für das Lernen einer Codeanalyse ein zeitintensiver Prozess der unter Beachtung jeder Fehlerklasse durchgeführt werden muss.

9 Fazit und Ausblick

Diese Masterarbeit hat sich der Problemstellung gewidmet, Fehler im Java-Quellcode durch die Methoden des maschinellen Lernens (ML) zu ermitteln. Speziell handelt es sich dabei um eine Auswahl von Bad-Coding-Practices (BCP), die während der Implementation des Quellcodes durch ein gelerntes Modell analysiert und identifiziert werden. Ziel der Arbeit ist eine Überprüfungen der Machbarkeit dieses Vorhabens durch die Entwicklung und Implementation eines Prototyps. Dabei wurde stets darauf geachtet, ein Konzept zu entwickeln, das eine große Bandbreite von unterschiedlichen Fehlern erkennen kann. Ferner sollen Fehler in beliebigen Elementen des Quellcodes erkannten werden. Thematisch befindet sich diese Arbeit in den Bereichen der statischen Codeanalyse und dem Program Language Processing (PLP). Mit dem Kapitel 2 *Fehler im Programmcode* beginnt die Arbeit mit einer detaillierten Einordnung und Abgrenzung des Fehler-Begriffs. Eine anschließende Auswahl von Fehlern grenzt das Vorhaben dieser Arbeit auf einen konkreten Problemfall ein, durch den die Machbarkeit der behandelten Problematik überprüft werden soll. Durch das Kapitel 3 *Statische Codeanalysen* folgt eine Einordnung der Thematik im Bereich der statischen Codeanalyse. Neben einer allgemeinen Einführung in die Methodik der statischen Codeanalyse wurden außerdem konkrete Vorgehensweisen für die Identifikation der zuvor getätigten Fehlerauswahl im Quellcode erläutert. Zu diesem Zweck wurden die Algorithmen der statischen Codeanalyse PMD analysiert. Das Resultat der Analyse besteht aus eine Aufzählung wichtiger Merkmale für die Identifikation der Fehler im Quellcode.

Aufbauend auf Kapitel 2 und 3 folgt in Kapitel 4 *Statische Codeanalyse als ML-Problem* die Formulierung der Fehleranalyse als ML-Problem. In diesem Rahmen wurde die Fehleranalyse als Multi-Class-Klassifikationsproblem formuliert, das durch die Anwendung von überwachten Lernverfahren auf einem Datensatz bestehend aus OSS-Projekten gelernt wird. Jedes syntaktische Element im Quellcode soll durch das Modell klassifiziert werden können. Als Performance-Indikator wurde die F_1-Score-Metrik festgelegt.

Kapitel 5 *Vom Programmcode lernen* beinhaltet eine thematische Einordnung dieser Arbeit in den Bereichen PLP und ML. Ferner wurde eine ausführliche Recherche gängiger Methoden für das Lernen von Quellcode durchgeführt. Das beinhaltet eine Auswahl geeigneter Methoden für die vorliegende Problemstellung. Dabei haben sich AST und Daten- bzw. Kontrollflussgraphen als geeignete Repräsentation des Quellcodes herausgestellt. Hieraus resultierend wurden verschiedene Methoden zum Lernen von Graphen auf ihre Vor- und Nachteile untersucht. Schlussendlich wurden Graph-Neural-Networks (GNN) als geeignetes Verfahren für die Klassifikation von Fehlern im Quellcode ausgewählt. Dabei stellten sich der Einbezug von semantischen Informationen und die Eigenschaft des induktiven Lernens als ausschlaggebende Vorteile heraus.

Aufbauend auf dieser Recherche wurde in Kapitel 6 *Konzept* ein Konzept für das Lernen einer Fehleranalyse konstruiert. Zu diesem Zweck wurde zunächst eine IR entwickelt, die relevante Merkmale für eine anschließende Analyse beinhaltet. Dabei handelt es sich um einen auf dem AST basierenden und durch semantische Kanten angereicherten Graphen. Jeder Knoten im Graphen wird durch ein Node-Feature repräsentiert, dessen Inhalt ebenfalls im Rahmen des Konzepts gestaltet wurde. Die Knoten werden durch drei verschiedene Kantentypen verbunden. Jeder Typ steht jeweils für eine eigene Art von Beziehung. Durch die Verwendung des ASTs als Basis wird jeder Knoten auf ein syntaktisches Element des Quellcodes abgebildet. Mit dem anschließend erstellten künstlichen neuronalen Netz, bestehend aus einer Embedding-Schicht, mehreren RGCN-Schichten und einer linearen Ausgabeschicht, wird jeder Knoten, und somit jedes syntaktische Element im Quellcode, klassifiziert.

Im nachfolgenden Kapitel 7 *Experimente* folgte die Implementation des Konzepts. Das umfasste sowohl den Aufbau der Lern-, Validierungs- und Testdatensätze als auch die Ermittlung der Hyperparameter zum Lernen des Modells. Die Datensätze wurden aus zufälligen Repositories mit Java-Dateien des Filehosters GitHub gebildet. Somit enthalten die Datensätze eine realistische Verteilung der Klassen und Merkmale und folglich ein wirklichkeitsgetreues Signal. Die rohen Daten wurden anschließend durch ein eigens entwickeltes Skript in die vorgestellte IR transformiert und durch PMD gelabelt. Aus dieser Vorbereitung resultieren ein Lern- und ein Testdatensatz. Der Lerndatensatz wurde anschließend stratifiziert in 80 % Lerndaten und 20% Validierungsdaten aufgeteilt.

Der Datenvorbereitung folgte die Implementation des künstlichen neuronalen Netzes mit PyTorch und PyTorch Geometric. Auf den Lerndaten wurden verschiedene Experimente mit variierenden Hyperparametern durchgeführt, deren Werte auf Basis der Performance

der Validierungsdaten angepasst wurden. Ziel dessen war die Annäherung an eine optimale Konfiguration des Modells. Ferner sollte der Einfluss verschiedener Hyperparameter auf die Performance des Modells untersucht werden. Es stellte sich heraus das die Dimension der Embeddings und die Anzahl der Iterationen den größten Einfluss auf die Performance des Modells haben. Beide Parameter haben, bis zu einer bestimmten Grenze, eine tendenziell bessere Performance bei höheren Werten erzielt. Mit der entsprechend experimentell erstellten Konfiguration wurde der finale Prototyp auf Basis des gesamten Lerndatensatzes (Lern- und Validierungsdaten) gelernt. In einer abschließenden Überprüfung der Performance des finalen Modells auf Basis der Testdaten haben sich einige Diskrepanzen zwischen der Performance auf den Test- und Validierungsdaten gezeigt.

In Kapitel 8 *Analyse und Diskussion* wurde infolgedessen eine Analyse der Testergebnisse mit einer kritischen Diskussion vorgenommen. Hier wurde pro Fehlerklasse eine Untersuchung potenzieller Problemfaktoren durchgeführt. Dabei hat sich unter anderem die mangelnde Größe des Lerndatensatzes und die damit einhergehende Ähnlichkeit von Lern- und Validierungsdaten als Problem herausgestellt. Diese resultierte für mehrere Klassen schlussendlich in einer Überanpassung und bieten eine Erklärung für die Diskrepanz der Performance zwischen Validierungs- und Testdaten. Schlussendlich konnte gezeigt werden, dass das Lernen eines Multi-Class-Klassifikators auf Basis eines GNN, der Fehler in beliebigen syntaktischen Elementen identifiziert, grundsätzlich funktioniert. Wenngleich diese Arbeit das Problem der Fehleranalyse durch das Lernen einer Multi-Class-Klassifizierung vereinfacht hat. Schließlich konnte das Signal einiger Fehlerklassen weitestgehend gelernt werden. Ein Großteil der Fehlerklassen boten jedoch entweder auf Validierungs- und Testdaten oder nur auf den Testdaten eine mangelhafte Performance. Trotz der mangelnden Interpretierbarkeit von Vorhersagen tiefer neuronaler Netze konnten einige Erkenntnisse und Rückschlüsse auf die Verhaltensweise des gelernten Modells gewonnen werden. Insbesondere auf potenzielle Problemfaktoren bei Fehlerklassen mit einer nicht ausreichenden Performance. Als wohl gravierendstes Defizit wurde die nicht ausreichende Größe der Lerndaten und die damit einhergehende Überanpassung ausgemacht. Ferner wurde sowohl eine, durch den Einbezug von Bezeichnern bedingte, suboptimale Ausdrucksstärke der IR als auch die mangelnde Vernetzung der Knoten in der IR als Problemfaktoren identifiziert. Einer der zunächst angepriesenen Vorteile, das Lernen von komplexen Mustern in den Daten, hat sich diesbezüglich also als Nachteil herausgestellt. Außerdem scheint der gewählte Label-Mechanismus sowie die Vereinfachung der Klassifizierungsaufgabe zu Problemen bei der Fehleridentifikation zu führen.

Diese Hypothese wird durch Bewertung verschiedener Stichproben während der Analyse bestärkt.

Eine verbesserte Performance könnte demnach sowohl durch die Vergrößerung des Lern- und Validierungsdatensatzes als auch einer Anpassung der IR erfolgen. Dabei kann maßgeblich durch die Normalisierung der Bezeichner und dem Hinzufügen weiterer relevanter Kanten eine Performancesteigerung erwartet werden. Das Verwenden weiterer Regularisierungsmethoden gegen die Überanpassung kann ebenfalls eine Verbesserung erzielen.

Mit dem Lernen einer Fehleranalyse leistet diese Arbeit einen Beitrag in den Bereichen der statischen Codeanalyse sowie dem PLP. Die statische Codeanalyse ist ein bereits seit den 70er Jahren aktives Forschungsgebiet, das sich aufgrund der wachsenden Bedeutung von Software und der steigenden Anzahl von Programmiersprachen stets an großer Beliebtheit erfreut. Das Verarbeiten von Programmen durch ML-Methoden (PLP) ist ein vergleichsweise junges Forschungsgebiet, das sich aus dem Erfolg von Sprachmodellen und tiefen künstlichen neuronalen Netzen entwickelt hat und eine zunehmende Popularität aufweist. Die steigende Popularität ist mitunter durch die Fülle der Anwendungsmöglichkeiten zu erklären. Eine stetig wachsende Anzahl an Arbeiten leistet mit neuen Modellen zum Lernen oder Generieren verschiedenster Codeeigenschaften zu diesem Forschungsgebiet einen Beitrag.

Die Analyse von Fehlern im Quellcode wurde in dieser Arbeit als Multi-Class-Problem definiert. Diese Vereinfachung reduziert den tatsächlichen Nutzen des Konzepts und stellt einen erheblichen Nachteil im Gegensatz zu etablierten statischen Codeanalysen dar. Schließlich kann ein Codeabschnitt auch mehrere Fehler aufweisen. Der Forschungsbereich des maschinellen Lernens bietet jedoch verschiedene Ansätze, um das Problem als Multi-Label-Problem zu formulieren und zu implementieren. Die Arbeit von Herrera et. al. [28] bietet diesbezüglich einen guten Überblick.

Das Labeln der Lerndaten durch ein bestehendes statisches Codeanalyseprogramm wie PMD bietet auf lange Sicht nur bedingt einen Mehrwert. Die Verwendung mehrerer Quellen für das Labeln der Lerndaten könnte hier Abhilfe schaffen. Relevante Quellen können unter anderem aus dem Forschungsgebiet *Mining Software Repositories*[1] entspringen. So ist es möglich auf Basis von Commit-Nachrichten oder Code-Kommentaren fehlerhafte Codeabschnitte zu annotieren und die gelernte Analyse somit durch weitere Eingaben zu optimieren [34][41].

[1] http://www.msrconf.org/

In dem hier entwickelten Konzept wurde das einmalige Lernen eines Klassifikators für eine feste Anzahl an Fehlern vorgestellt. Eine Erweiterung der Fehleranalyse um weitere Fehlerarten ist jedoch unerlässlich. Schließlich sollen auch neu entdeckte Fehler durch diese Analyse abgedeckt werden. Die Erweiterbarkeit des Modells ist somit ein essenzieller Faktor für den tatsächlichen Nutzen einer gelernten statischen Codeanalyse. Parisi et. al. [51] bieten diesbezüglich einen Überblick über aktuelle Herausforderungen und Lösungen in diesem Forschungsbereich.

Einer der größten Nachteile den tiefe künstliche neuronale Netze mit sich bringen, ist die Intransparenz der Entscheidungswege. Diese Eigenschaft hat auch im Rahmen dieser Arbeit zu Problemen bei der Analyse des Modells geführt. Das kann unter anderem die Akzeptanz und den praktischen Nutzen einer gelernten statischen Codeanalyse reduzieren. Für dieses Problem bietet das Forschungsgebiet der *Explainable AI* Lösungsansätze. Erklärbarkeit des maschinellen Lernens ist nicht Gegenstand dieser Arbeit. Molnar er. al. [46] bieten diesbezüglich einen Überblick über aktuelle Methoden.

Schlussendlich wurde in dieser Arbeit durch die Konzeption, Implementation und Analyse eines Prototyps aufgezeigt, dass das Lernen einer statischen Codeanalyse grundsätzlich möglich ist. Insbesondere wurden wichtige Komponenten thematisiert, Herausforderungen identifiziert und Lösungsansätze skizziert. Das Lernen einer statische Codeanalyse, die eine Fehleridentifikation auf beliebigen syntaktischen Elementen durchführen kann, ist also mit den vorgeschlagenen Verfahren möglich. Bis diese jedoch eine ernsthafte Alternative zu den klassischen Verfahren der statischen Codeanalyse bietet, müssen noch einige grundlegende Probleme gelöst werden.

Literaturverzeichnis

[1] Ahmed, Mahtab ; Samee, Muhammad R. ; Mercer, Robert E.: Improving tree-LSTM with tree attention. In: *2019 IEEE 13th International Conference on Semantic Computing (ICSC)* IEEE (Veranst.), 2019, S. 247–254

[2] Aho, Alfred V. ; Sethi, Ravi ; Ullman, Jeffrey D.: Compilers, principles, techniques. In: *Addison wesley* 7 (1986), Nr. 8, S. 9

[3] Allamanis, Miltiadis ; Barr, Earl T. ; Devanbu, Premkumar ; Sutton, Charles: A survey of machine learning for big code and naturalness. In: *ACM Computing Surveys (CSUR)* 51 (2018), Nr. 4, S. 81

[4] Allamanis, Miltiadis ; Brockschmidt, Marc ; Khademi, Mahmoud: Learning to represent programs with graphs. In: *arXiv preprint arXiv:1711.00740* (2017)

[5] Alon, Uri ; Zilberstein, Meital ; Levy, Omer ; Yahav, Eran: A general path-based representation for predicting program properties. In: *ACM SIGPLAN Notices* Bd. 53 ACM (Veranst.), 2018, S. 404–419

[6] Alon, Uri ; Zilberstein, Meital ; Levy, Omer ; Yahav, Eran: code2vec: Learning distributed representations of code. In: *Proceedings of the ACM on Programming Languages* 3 (2019), Nr. POPL, S. 40

[7] Alpaydin, Ethem: *Maschinelles Lernen.* Walter de Gruyter GmbH & Co KG, 2019

[8] Andrew, W A. ; Jens, Palsberg: Modern compiler implementation in Java. In: *ISBN 0–521–58388–8.* Cambridge University Press, 2002

[9] Brownlee, Jason: *Master Machine Learning Algorithms: discover how they work and implement them from scratch.* Machine Learning Mastery, 2016

[10] Brownlee, Jason: *Statistical methods for machine learning: Discover how to transform data into knowledge with Python.* Machine Learning Mastery, 2018

[11] Brownlee, Jason: Data Preparation for Machine Learning. (2020)

[12] Bui, Nghi D. ; Yu, Yijun ; Jiang, Lingxiao: InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees. In: *arXiv e-prints* (2020), S. arXiv–2012

[13] Butler, Simon ; Wermelinger, Michel ; Yu, Yijun ; Sharp, Helen: Exploring the influence of identifier names on code quality: An empirical study. In: *2010 14th European Conference on Software Maintenance and Reengineering* IEEE (Veranst.), 2010, S. 156–165

[14] Cai, Hongyun ; Zheng, Vincent W. ; Chang, Kevin Chen-Chuan: A comprehensive survey of graph embedding: Problems, techniques, and applications. In: *IEEE Transactions on Knowledge and Data Engineering* 30 (2018), Nr. 9, S. 1616–1637

[15] Chami, Ines ; Abu-El-Haija, Sami ; Perozzi, Bryan ; Ré, Christopher ; Murphy, Kevin: Machine learning on graphs: A model and comprehensive taxonomy. In: *arXiv preprint arXiv:2005.03675* (2020)

[16] Chess, Brian ; West, Jacob: *Secure programming with static analysis*. Pearson Education, 2007

[17] Compton, Rhys ; Frank, Eibe ; Patros, Panos ; Koay, Abigail: Embedding java classes with code2vec: Improvements from variable obfuscation. In: *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, S. 243–253

[18] Cui, Peng ; Wang, Xiao ; Pei, Jian ; Zhu, Wenwu: A survey on network embedding. In: *IEEE Transactions on Knowledge and Data Engineering* 31 (2018), Nr. 5, S. 833–852

[19] DeFreez, Daniel ; Thakur, Aditya V. ; Rubio-González, Cindy: Path-based function embedding and its application to specification mining. In: *arXiv preprint arXiv:1802.07779* (2018)

[20] Fischer, Peter ; Hofer, Peter: *Lexikon der Informatik*. Springer, 2011

[21] Frochte, Jörg: *Maschinelles Lernen: Grundlagen und Algorithmen in Python*. Carl Hanser Verlag GmbH Co KG, 2019

[22] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep learning*. MIT press, 2016

[23] Gosling, James ; Joy, Bill ; Steele, Guy L. ; Bracha, Gilad ; Buckley, Alex ; Smith, Daniel: *The Java Language Specification, Java SE 13 Edition. 1st.* 2019

[24] Grover, Aditya ; Leskovec, Jure: node2vec: Scalable feature learning for networks. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, S. 855–864

[25] Hamilton, William L.: Graph Representation Learning. In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14, Nr. 3, S. 1–159

[26] Heindl, Christoph: Graph Neural Networks for Node-Level Predictions. In: *arXiv preprint arXiv:2007.08649* (2020)

[27] Hellendoorn, Vincent J. ; Bird, Christian ; Barr, Earl T. ; Allamanis, Miltiadis: Deep learning type inference. In: *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, S. 152–162

[28] Herrera, Francisco ; Charte, Francisco ; Rivera, Antonio J. ; Del Jesus, María J: Multilabel classification. In: *Multilabel Classification*. Springer, 2016, S. 17–31

[29] Hindle, Abram ; Barr, Earl T. ; Gabel, Mark ; Su, Zhendong ; Devanbu, Premkumar: On the naturalness of software. In: *Communications of the ACM* 59 (2016), Nr. 5, S. 122–131

[30] Hoffmann, Dirk W.: *Software-Qualität*. Springer-Verlag, 2013

[31] Hölzl, Matthias ; Raed, Allaithy ; Wirsing, Martin: *Java kompakt*. Springer, 2013

[32] Huang, JC: *Software error detection through testing and analysis*. Wiley Online Library, 2009

[33] Johnson, Stephen C.: A tour through the portable C compiler. In: *Unix programmer's manual,* 2 (1979)

[34] Jung, Woosung ; Lee, Eunjoo ; Wu, Chisu: A survey on mining software repositories. In: *IEICE TRANSACTIONS on Information and Systems* 95 (2012), Nr. 5, S. 1384–1406

[35] Kingma, Diederik P. ; Ba, Jimmy: Adam: A method for stochastic optimization. In: *arXiv preprint arXiv:1412.6980* (2014)

[36] Kipf, Thomas N. ; Welling, Max: Semi-supervised classification with graph convolutional networks. In: *arXiv preprint arXiv:1609.02907* (2016)

[37] Knuth, Donald E.: Literate programming. In: *The Computer Journal* 27 (1984), Nr. 2, S. 97–111

[38] Larabel, Michael: *The Linux Kernel Enters 2020 At 27.8 Million Lines In Git But With Less Developers For 2019*. Januar 2020. – URL https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019. – Accessed: 12.10.2020

[39] Li, Yi ; Wang, Shaohua ; Nguyen, Tien N. ; Van Nguyen, Son: Improving bug detection via context-based code representation learning and attention-based neural networks. In: *Proceedings of the ACM on Programming Languages* 3 (2019), Nr. OOPSLA, S. 1–30

[40] Liggesmeyer, Peter: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Springer Science & Business Media, 2009

[41] Martinez, Matias ; Duchien, Laurence ; Monperrus, Martin: Accurate Extraction of Bug Fix Pattern Occurrences using Abstract Syntax Tree Analysis. (2014)

[42] Masters, D ; Luschi, C: Revisiting small batch training for deep neural networks. arXiv 2018. In: *arXiv preprint arXiv:1804.07612* (2019)

[43] McConnell, Steve: Code complete. (2004)

[44] Mikolov, Tomas ; Chen, Kai ; Corrado, Greg ; Dean, Jeffrey: Efficient estimation of word representations in vector space. In: *arXiv preprint arXiv:1301.3781* (2013)

[45] Mitchell, Tom M.: *Machine Learning*. New York : McGraw-Hill, 1997. – ISBN 978-0-07-042807-2

[46] Molnar, Christoph: *Interpretable machine learning*. URL https://christophm.github.io/interpretable-ml-book/, 2019

[47] Mou, Lili ; Li, Ge ; Zhang, Lu ; Wang, Tao ; Jin, Zhi: Convolutional neural networks over tree structures for programming language processing. In: *Proceedings of the AAAI Conference on Artificial Intelligence* Bd. 30, 2016

[48] Mueller, John P.: *Java ELearning Kit for Dummies*. John Wiley & Sons, 2014

[49] Novak, Jernej ; Krajnc, Andrej u. a.: Taxonomy of static code analysis tools. In: *The 33rd International Convention MIPRO* IEEE (Veranst.), 2010, S. 418–422

[50] Otte, Ralf: *Künstliche Inteligenz für dummies.* 1. 2019. – ISBN 978352771494

[51] Parisi, German I. ; Kemker, Ronald ; Part, Jose L. ; Kanan, Christopher ; Wermter, Stefan: Continual lifelong learning with neural networks: A review. In: *Neural Networks* 113 (2019), S. 54–71

[52] Perozzi, Bryan ; Al-Rfou, Rami ; Skiena, Steven: Deepwalk: Online learning of social representations. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, S. 701–710

[53] Pradel, Michael ; Sen, Koushik: Deepbugs: A learning approach to name-based bug detection. In: *Proceedings of the ACM on Programming Languages* 2 (2018), Nr. OOPSLA, S. 1–25

[54] Rao, Delip ; McMahan, Brian: *Natural language processing with PyTorch: build intelligent language applications using deep learning.* O'Reilly Media, Inc., 2019

[55] Ruder, Sebastian: An overview of gradient descent optimization algorithms. In: *arXiv preprint arXiv:1609.04747* (2016)

[56] Scarselli, Franco ; Gori, Marco ; Tsoi, Ah C. ; Hagenbuchner, Markus ; Monfardini, Gabriele: The graph neural network model. In: *IEEE transactions on neural networks* 20 (2008), Nr. 1, S. 61–80

[57] Schlichtkrull, Michael ; Kipf, Thomas N. ; Bloem, Peter ; Van Den Berg, Rianne ; Titov, Ivan ; Welling, Max: Modeling relational data with graph convolutional networks. In: *European semantic web conference* Springer (Veranst.), 2018, S. 593–607

[58] Schwaiger, Roland ; Steinwendner, Joachim: *Neuronale Netze programmieren mit Python.* Rheinwerk Computing, 2019

[59] Schädler, Florian: Datengetriebene statische Codeanalyse zur Ermittlungvon nicht verwedendeten Variablendeklarationen. (2019)

[60] Schädler, Florian: Datengetriebene statische Codeanalyse für Multi-Label Klassifikationsprobleme. (2020)

[61] Smith, Nicholas ; Bruggen, Danny van ; Tomassetti, Federico: JavaParser: visited. In: *Leanpub, oct. de* (2017)

[62] Sokolova, Marina ; Lapalme, Guy: A systematic analysis of performance measures for classification tasks. In: *Information processing & management* 45 (2009), Nr. 4, S. 427–437

[63] Steger, Angelika: *Diskrete Strukturen: Band 1: Kombinatorik, Graphentheorie, Algebra*. Springer-Verlag, 2007

[64] Tai, Kai S. ; Socher, Richard ; Manning, Christopher D.: Improved semantic representations from tree-structured long short-term memory networks. In: *arXiv preprint arXiv:1503.00075* (2015)

[65] Ullenboom, Christian: *Java ist auch eine Insel*. Galileo Press, 2011

[66] Veličković, Petar ; Cucurull, Guillem ; Casanova, Arantxa ; Romero, Adriana ; Lio, Pietro ; Bengio, Yoshua: Graph attention networks. In: *arXiv preprint arXiv:1710.10903* (2017)

[67] Wang, Song ; Chollak, Devin ; Movshovitz-Attias, Dana ; Tan, Lin: Bugram: bug detection with n-gram language models. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, S. 708–719

[68] Wang, Wenhan ; Li, Ge ; Ma, Bo ; Xia, Xin ; Jin, Zhi: Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* IEEE (Veranst.), 2020, S. 261–271

[69] Wilhelm, Reinhard ; Seidl, Helmut ; Hack, Sebastian: *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media, 2013

[70] Wirth, Niklaus: A brief history of software engineering. In: *IEEE Annals of the History of Computing* 30 (2008), Nr. 3, S. 32–39

[71] Wu, Xi-Zhu ; Zhou, Zhi-Hua: A unified view of multi-label performance measures. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70* JMLR. org (Veranst.), 2017, S. 3780–3788

[72] Xi, Hongwei: Dead code elimination through dependent types. In: *International Symposium on Practical Aspects of Declarative Languages* Springer (Veranst.), 1999, S. 228–242

[73] Zhang, Jian ; Wang, Xu ; Zhang, Hongyu ; Sun, Hailong ; Wang, Kaixuan ; Liu, Xudong: A novel neural source code representation based on abstract syntax tree. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* IEEE (Veranst.), 2019, S. 783–794

[74] Zhao, Haitao ; Lai, Zhihui ; Leung, Henry ; Zhang, Xianyi: *Feature Learning and Understanding: Algorithms and Applications*. Springer Nature, 2020

[75] Zheng, Alice ; Casari, Amanda: *Feature engineering for machine learning: principles and techniques for data scientists*. Ö'Reilly Media, Inc.", 2018

[76] Zhou, Jie ; Cui, Ganqu ; Zhang, Zhengyan ; Yang, Cheng ; Liu, Zhiyuan ; Wang, Lifeng ; Li, Changcheng ; Sun, Maosong: Graph Neural Networks: A Review of Methods and Applications. In: *arXiv:1812.08434 [cs, stat]* (2019), Juli. – URL http://arxiv.org/abs/1812.08434. – Zugriffsdatum: 2021-03-05. – arXiv: 1812.08434

A Anhang

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Masterarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Maschinelles Lernen für statische Codeanalysen

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original