Informatics

Predicting Bugs in Source Code

A Machine Learning Approach for Predicting Faults by utilizing Code & Change Metrics

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jodok Felder, BSc
Matrikelnummer 11775798

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Prof. Dr. Edgar Weippl
Mitwirkung: Dr. Alexander Schatten
Philip König, BA MSc

Wien, 22. Jänner 2025

Jodok Felder   Edgar Weippl

Informatics

Predicting Bugs in Source Code

A Machine Learning Approach for Predicting Faults by utilizing Code & Change Metrics

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Jodok Felder, BSc
Registration Number 11775798

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Dr. Edgar Weippl
Assistance: Dr. Alexander Schatten
Philip König, BA MSc

Vienna, January 22, 2025

Jodok Felder	Edgar Weippl

Erklärung zur Verfassung der Arbeit

Jodok Felder, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 22. Jänner 2025

Jodok Felder

Danksagung

Wie ich im Verlauf dieser Arbeit mehrfach betonen werde, ist diese Arbeit selbstverständlich nicht isoliert entstanden. Ich hatte das Privileg, auf die Expertise zahlreicher Menschen zurückgreifen zu können, und möchte an dieser Stelle all jenen danken, die mich während des Prozesses unterstützt haben.

Mein besonderer Dank gilt zunächst meinem Betreuer Univ.-Prof. Dr. Edgar Weippl sowie meinen Co-Betreuern Dr. Alexander Schatten und Philip König. Sie haben mich von der Entwicklung der ersten Ideen bis hin zur abschließenden Korrektur dieser Arbeit durchgehend begleitet. Ihre kontinuierliche Unterstützung, ihr konstruktives Feedback und ihre fachliche Expertise haben entscheidend dazu beigetragen, dass diese Masterarbeit in ihrer vorliegenden Form entstehen konnte.

Mein Dank richtet sich zudem an das Team des SAGE Projekts von SBA Research, das folgende Personen umfasst: Dennis Toth, Fabian Obermann und Zoe Herzig. Gemeinsam mit Dr. Alexander Schatten und Philip König arbeiteten sie an dem Projekt, das den Rahmen und den übergeordneten Kontext für meine Arbeit bildete. Ihre Arbeiten, insbesondere die Erstellung der Datensätze, legten die Grundlage für das Training meines Modells.

Weiters gebührt Sebastian Raubitzek von SBA Research mein Dank, der mir während der Entwicklung des Machine Learning Algorithmus stets mit Rat und Tat zur Seite stand. Seine Bereitschaft, Fragen zu beantworten, gemeinsame Brainstorming-Sitzungen vor der Implementierung abzuhalten und die Architektur sowie mögliche Algorithmen mit mir zu diskutieren, war von unschätzbarem Wert.

Abschließend möchte ich mich bei meinen Eltern bedanken, die mich während meines gesamten Studiums stets unterstützt haben. Mein Dank gilt außerdem meiner Familie sowie meinen Freund:innen, die mir während des Studiums zur Seite standen und mich auf meinem Weg begleitet haben.

Ohne die Unterstützung all dieser Personen wäre diese Arbeit in ihrem heutigen Umfang und ihrer Qualität nicht möglich gewesen. Dafür möchte ich ihnen allen meinen tief empfundenen Dank aussprechen.

Acknowledgements

As I will emphasize repeatedly throughout this thesis, this work was not created in isolation. I had the privilege of relying on the expertise of many individuals, and I would like to take this opportunity to express my gratitude to everyone who supported me during this process.

First and foremost, my deepest thanks go to my supervisor, Univ.-Prof. Dr. Edgar Weippl, and my co-supervisors, Dr. Alexander Schatten and Philip König. From the initial development of ideas to the final review of this thesis, they provided continuous guidance, support, and constructive feedback. Their expertise was instrumental in shaping this work into its final form.

I would also like to extend my gratitude to the members of the SAGE project at SBA Research, including Dennis Toth, Fabian Obermann, and Zoe Herzig. Together with Dr. Alexander Schatten and Philip König, they worked on the project that provided the framework and broader context for my thesis. Their contributions, particularly the creation of the datasets I used for training my model, were essential to my work.

Furthermore, I want to thank Sebastian Raubitzek from SBA Research, who generously made himself available throughout the development of the machine learning algorithm. His willingness to answer questions, engage in joint brainstorming sessions before implementation, and discuss architecture and algorithm selection proved invaluable.

Finally, I would like to thank my parents, who supported me throughout my entire studies. I am also deeply grateful to my family and my friends, who stood by my side and accompanied me on this journey.

Without the support of these individuals, this thesis would not have been possible in its current scope and quality. For this, I extend my heartfelt thanks to each and every one of them.

Kurzfassung

Die Erkennung von Fehlern ist ein zentraler Bestandteil des Software Engineerings und bietet Entwicklern und Unternehmen gleichermaßen großes Potenzial für Zeit- und Kosteneinsparungen. Mit der steigenden Menge an geschriebenem Code und der wachsenden Verfügbarkeit von Entwicklungsdaten wird die frühzeitige Vorhersage von Fehlern immer bedeutender. Diese Arbeit untersucht, wie sich Code-Metriken, insbesondere solche, die auf Code-Änderungen basieren, mit maschinellem Lernen kombinieren lassen, um fehleranfällige Dateien in Softwareprojekten effektiv zu identifizieren.

Basierend auf Daten aus 34 Open-Source-Projekten wurden über 37 verschiedene Code-Metriken eingesetzt – von einfachen Kennzahlen wie *Lines of Code* bis hin zu komplexeren Metriken, die Prinzipien der objektorientierten Programmierung berücksichtigen. Mithilfe eines CatBoost-Klassifikators wurde ein Modell entwickelt, das Dateien als fehlerhaft oder nicht fehlerhaft klassifiziert und einen Risiko-Score berechnet – ein numerischer Wert, der die Wahrscheinlichkeit von Fehlern in einer Datei angibt. Das Modell erreichte eine durchschnittliche Genauigkeit von 84,1% sowie eine Recall-Rate von 83%, was seine hohe Zuverlässigkeit und Leistungsfähigkeit bei der Fehlererkennung unter Beweis stellt.

Zusätzlich wurde analysiert, welche Code-Metriken den größten Einfluss auf die Vorhersagen des Modells hatten. Die Ergebnisse zeigten, dass Komplexitätsmetriken und der Bus-Faktor entscheidende Prädiktoren für fehlerhafte Dateien sind. Diese Erkenntnisse liefern wertvolle Hinweise darauf, welche Aspekte von Code maßgeblich zur Softwarequalität beitragen. Ein Vergleich mit einem Ansatz auf Basis logistischer Regression, der eine Genauigkeit von 61% erreichte, verdeutlicht die Vorteile nichtlinearer Modelle wie CatBoost bei der Fehlerprognose.

Insgesamt demonstriert diese Arbeit, wie maschinelles Lernen in Kombination mit codebasierten Metriken genutzt werden kann, um die Softwarezuverlässigkeit zu verbessern. Die Ergebnisse bilden eine solide Grundlage für zukünftige Forschung und Anwendungen, die darauf abzielen, Softwareentwicklungsprozesse effizienter und fehlerresistenter zu gestalten.

Abstract

Bug detection plays a critical role in software engineering, offering significant time and cost savings for organizations and developers alike. With the exponential growth in code volume and the availability of data surrounding its development, bug prediction has become increasingly important. This thesis focuses on combining code metrics, especially ones based on code changes, and machine learning techniques to address the challenge of identifying buggy software files.

The thesis leverages a dataset comprising 34 open-source projects and utilizes more than 37 code metrics, ranging from basic measures such as Lines of Code to advanced metrics rooted in object-oriented programming principles. A CatBoost classifier was employed to develop a predictive model capable of classifying files as buggy or non-buggy and assigning a corresponding Risk Score – a numerical indicator of the likelihood that a given file contains bugs. The model achieved an average accuracy of 84.1% and a recall rate of 83%, demonstrating its reliability and effectiveness in identifying buggy files.

The analysis further examined the importance of individual code metrics in driving the model's predictions. Feature Importance Analysis identified complexity metrics and the Bus Factor as the most influential in predicting buggy files, offering valuable insights into key contributors to software quality. Additionally, a Logistic Regression-based approach, which achieved an accuracy of 61%, was evaluated to contrast its performance with advanced non-linear models like CatBoost, demonstrating the latter's superior predictive capabilities for bug prediction.

This work contributes to the field of software engineering by demonstrating the efficacy of combining machine learning with metric-driven approaches for bug prediction. The results provide a foundation for future research and practical applications aimed at enhancing software reliability and development efficiency.

Contents

CHAPTER 1

Introduction

1.1 Problem Specification & Motivation

A central challenge emerges throughout all aspects of Software Engineering and various software projects: software decay, aging, and the loss of critical knowledge over time. Small bugs introduced today can snowball into significant errors and malfunctions in the future. This degradation often results in worse performance, increased error-proneness, and heightened security risks. The security implications, in particular, cannot be overstated, as even minor bugs can lead to severe vulnerabilities, potentially compromising the entire product.

This poses a tangible risk to software projects, often leading to escalating technical debt that can spiral out of control and result in dire consequences. According to the Consortium for Information & Software Quality, the cost of poor Software quality in the US was estimated to be above \$2 Trillion [1]. This underscores why bug detection and prediction are essential components of any software project aspiring to longevity. By investing in early detection of software decay, teams can better estimate maintenance costs, improve project reliability, and enhance security.

The overarching project conducted by the Complexity and Resilience Research Group at SBA Research addresses these challenges by collecting and calculating a wide range of metrics on code, commits, and other software-related data. These metrics are then analyzed to provide meaningful insights into code quality and reliable bug prediction. However, with so many metrics in play, each with its own purpose and focus, it can be challenging to a) maintain an overview of the most relevant metrics for specific cases and b) identify meaningful connections and correlations among them.

This is where machine learning proves invaluable. These models are specifically designed to uncover correlations and patterns that might elude human observation or intuition. By processing and interpreting the vast amount of data derived from various metrics, machine learning models can distill complex and overwhelming inputs into simple, actionable outputs. As a result, they offer a powerful tool for analyzing code and predicting potential issues, helping to ensure the reliability and sustainability of software projects.

This thesis aims to combine machine learning efforts with the overarching project and in turn, the objective is to generate a model that can reliably predict bugs in a given code. Before delving deeper into the aim of this work, it is helpful to briefly examine bug prediction, its definition, and its usefulness.

1.1.1 What is Bug Prediction & why is it important?

Naturally, bug prediction is about predicting faults in code. But what exactly does that mean and how is it done? In general, one can say that the aim of bug prediction is to spot bugs and faults as early as possible in the Software development lifecycle. The later on in this lifecycle a bug is detected, the more expensive it gets to fix, both time- and money-wise, with the worst-case being a bug that made its way to production.

Besides the obvious cost factor, it is also vital to note that buggy legacy code can serve as a foundation for much of later added code. As with traditional buildings, if the foundation is faulty, the whole structure is in danger. Therefore it is essential to spot errors as soon as possible and not build upon them.

This brief explanation, combined with the innate understanding and experience of every developer that bugs are inherently problematic, should sufficiently illustrate the importance of bug prediction. To further underscore this point, a concise list of high-profile & well-known software bugs and errors is provided, along with a short description of selected examples.

List of (in)famous bugs

Table 1.1: Significant Incidents and their Impacts

Incident	**Damage**	**Sources**
Boeing 737 MAX Groundings	346 deaths	[2], [3]
Aegis Combat / Iran Air Flight 655	290 deaths	[4]
2009 to 2011 Toyota Failures	>9 million cars recalled, alleged 89 deaths	[5], [6]
CrowdStrike Glitch	8.5 million systems crashed, damage of >$10 billion	[7], [8]
Knight Capital Glitch	$440 million lost	[9]
Ariane Flight V88	$370 million loss	[10]
NASA Mars Climate Orbiter	>$100 million cost of spacecraft	[11]
Heartbleed Vulnerability	Over 500k servers affected	[12]

Heartbleed, one of the most notorious vulnerabilities of the 2010s, was a security bug in the OpenSSL library, "a widely used implementation of the Transport Layer Security

(TLS) protocol." At the time of its discovery, an estimated 17% of all HTTPS servers on the internet were vulnerable, equating to approximately 500,000 systems globally [12].

In the case of Knight Capital, a glitch in their systems caused them to unintentionally purchase "150 different stocks at a total cost of around $7 billion". Lacking the funding to support these trades, they were forced to sell off the shares quickly, incurring a $440 million 'service fee' in the process [9].

The Mars Climate Orbiter serves as a cautionary tale of the importance of precision in software. A bug in the spacecraft's code caused a loss of communication, ultimately leading to its violent crash and the loss of hundreds of millions of dollars in project costs. Notably, the bug stemmed from a mismatch between measurement systems: one part of the software used metric units, while another used imperial units [11].

CrowdStrike's global internet outage, the most recent incident on this list, was triggered by a faulty update to a security product by the company. The error caused widespread outages affecting roughly 8.5 million systems globally. The financial damages were staggering, with losses estimated at over $10 billion [7] [8].

Lastly, the Boeing 737 MAX groundings marked a significant chapter in aviation history. A bug in the aircraft's software systems caused two planes to nosedive and crash, tragically claiming the lives of 346 people. Beyond the devastating human toll, the incidents resulted in the grounding of 387 aircraft and dealt Boeing a financial blow of approximately $20 billion in penalties, reparations, and associated legal expenses along with indirect losses exceeding $60 billion due to 1,200 canceled orders.[2] [3] [13]

By this point, it should be abundantly clear that bug prediction is a critical aspect of any software development project, regardless of its size or scope. Effective bug prediction not only saves time and money but can also, in extreme cases, prevent catastrophic outcomes where lives may be at stake.

After establishing the importance of addressing software bugs, the focus now shifts to the core objectives of this work. The next section delves into the specific goals of this project, detailing the methods and approaches employed to identify and predict bugs as early as possible. This approach aims to enhance software reliability, reduce maintenance overhead, and contribute to safer and more efficient development practices.

1.2 Research Questions

The goal of this work is to create a machine learning-based solution that is able to reliably predict whether a given code file contains a bug or not. To accomplish this, several code metrics are used to analyze the given files and serve as input to a supervised machine learning model. These metrics on their own are often already reliable indicators of whether code could contain faulty sections, and in this work, a number of them will be utilized in combination to maximize the knowledge gained from them and the degree of certainty with which a bug can be predicted.

To achieve this goal, the following research questions were formulated:

- **RQ1: How accurate is the developed Machine Learning model for Fault prediction?**
  This question evaluates the predictive performance of the developed model, focusing on metrics such as accuracy and recall to determine its effectiveness in identifying buggy files. The goal is to establish whether the model meets the reliability standards necessary for practical application in software engineering.

- **RQ2: Which metrics are best suited for Fault prediction and use in Machine Learning models?**
  This question seeks to identify the most influential code metrics that contribute to accurate fault prediction. By analyzing feature importance, the study highlights the metrics that have the strongest correlation with the presence of bugs, providing insights into their relevance and utility.

- **RQ3: How does a newly developed Supervised ML model perform in comparison to an Unsupervised one?**
  This question examines the differences in performance between supervised and unsupervised approaches to bug prediction. It aims to determine whether supervised models, which rely on labeled data, outperform unsupervised ones in terms of predictive accuracy and reliability.

- **RQ4: For the Machine Learning model, how does a Classifier approach fare versus a Linear Regression-based approach?**
  This question explores the comparative performance of classification and regression models for fault prediction. It investigates whether the non-linear classifier approach provides a more accurate and robust solution than the linear regression-based method.

1.3 Limitations of this work

The purpose of outlining these limitations is not just to identify what is beyond the scope of this research but to provide a clear understanding of its boundaries. This approach ensures transparency about the constraints of time, resources, and the specific objectives of this thesis. By doing so, readers can better align their expectations with the intended focus of this work.

The following discussion outlines these boundaries, explaining the context and reasoning behind the decisions that defined the scope of the thesis. While these limitations highlight areas for potential future research, they are essential for maintaining a focused and practical approach within the constraints of a Master's thesis.

1.3.1 Prototyped solution

It is crucial to emphasize that, while the developed model demonstrates high accuracy in predicting bugs and holds significant potential to aid software projects, it does not constitute a fully automated, out-of-the-box solution. In its current form, the accompanying codebase cannot simply be executed with raw source code as input to yield immediate bug or non-bug classifications.

This limitation arises because, as previously outlined, this thesis is part of a larger research endeavor. Within the broader project, essential preprocessing steps are undertaken to prepare the data used by the model. Specifically, the model does not analyze source code directly. Instead, it relies on metrics extracted and computed from the codebase and the associated commit history.

1.3.2 No claim to completeness

In machine learning, a wide variety of algorithms exist, each with unique strengths and trade-offs. Given this thesis's scope, testing all possible approaches was not practical. Instead, the selection of algorithms was guided by prior research, such as [14], [15], [16], [17] [18], and practical experience. The process began with a Random Forest algorithm, chosen for its simplicity and effectiveness as a baseline model. However, as the work progressed, it became clear that a Gradient Boosting approach, specifically the CatBoost algorithm, offered superior performance and accuracy.

The CatBoost algorithm was subsequently adopted as the primary model due to its ability to handle categorical data efficiently and its support for GPU acceleration, which significantly improved training speed and model refinement. This iterative approach allowed for the optimization of the model within the constraints of the thesis, ensuring a focus on practical and effective solutions.

Although this thesis does not claim to provide a comprehensive exploration of all available machine learning algorithms, the chosen methodology achieved a high level of accuracy, comparable to similar approaches in the literature. By leveraging CatBoost, this work demonstrates the utility of advanced machine learning techniques for bug prediction and lays the groundwork for further research into alternative algorithms and methodologies.

1.3.3 No comparison to State-of-the-Art

Finally, it is important to emphasize that this thesis does not offer a comprehensive state-of-the-art review, nor does it systematically place its findings in the context of, or directly compare them with, other approaches. While such an analysis is often the primary goal of similar research efforts, the focus of this thesis was shaped by its integration into a larger project on software aging and bug prediction. Within this context, aligning the work with the broader objectives of the overarching project was deemed more critical.

This is not to suggest that no comparisons or reviews were conducted. A limited review of the state-of-the-art is presented in the following Section, Related Work. However, the primary focus was not on benchmarking this solution against other methodologies. Instead, the work concentrated on analyzing the code metrics used in the dataset and evaluating their contribution to the model's performance within the project framework.

In summary, this thesis does not aim to serve as a state-of-the-art review. Rather, it seeks to develop a solution that is both relevant and applicable within the broader context of the associated research project.

1.3.4 Bugfix as a Proxy for Bugs in Code

Because it is virtually impossible to trace bugs in retrospect to their origins, in this dataset the target values are commits that are labeled as bugfixes, i.e. by having the word 'bug' or similar ones in their commit message. This in turn means that the model gets trained on the state (described by the metrics) of a file, at the point of the commit when the bug is fixed. A bugfix inherently indicates the existence of a bug, which limits the model's utility as a pre-commit checker for assessing whether new changes introduce potential risks. Rather, its advantages lie in the overall assessment of projects and files over time. This is a very important factor to keep in mind when reading this work and when its bug prediction abilities are mentioned.

CHAPTER 2

Related Work

To conclude this introductory chapter, this chapter reviews a selection of relevant literature in the field of bug prediction and software quality assessment. As outlined earlier, the motivation for this research is grounded in the long-standing challenges associated with identifying and addressing software bugs. Given the integral role of bug detection in software development, it is no surprise that this topic has been the subject of extensive investigation over the years.

In a similar vein, the desire to quantify the Quality of code using metrics goes back quite some time, with some of the most fundamental works on this topic being written 50 years ago like for example Thomas J. McCabe's "A Complexity Measure" in 1976 [19].

Much of the rise of machine learning can be attributed to the significant increase in computing power over the past 20 years, as well as the rapid growth of data publicly available, which led to more training data for models, its use for bug prediction, and detection naturally is not as old as the use of metrics. Nevertheless, it is not completely new and 'fresh' either, having played a role for at least 15 years now, if not more.

Although much of this work is focused on creating a fitting machine learning model with the aim of bug prediction, the code metrics, which make up the input of the model, play a crucial role as well, even if they are under-reported in this work, since they logically occur one step before this work. Therefore, this chapter will be split into two sections, one talking about Related Work in regard to code metrics and one detailing other machine learning approaches.

Because chronologically, as well as logically, the metrics occur first, starting with an examination of popular and related Literature in this domain.

2.1 Code Metrics

As mentioned, the need for code metrics has deep roots in the history of software development. From the early days of programming, there has been a drive to quantify aspects of code such as quality, complexity, and maintainability. The *Lines of Code* (LoC) metric emerged as one of the earliest measures, valued for its simplicity and ability to provide a rough estimate of a program's size. This early focus on metrics laid the groundwork for more sophisticated approaches to evaluating software attributes [20].

Hence, it should not be of much surprise that back in 1976 one of the fundamental works in this field was written, namely the already mentioned "A complexity measure" by McCabe [19]. In it, the author proposed an approach that visualized the flow of a program as a graph, on which mathematical operations from graph theory were executed and calculated, resulting in a score for what he called *cyclomatic complexity*. Through this method, simple mathematical analysis of programs was utilized to generate a score for the complexity of a given program, with the intuitive implication that a higher number is worse, and could imply higher error-proneness. As stated already, this work has been quite influential in this field and thus remains relevant to this day, almost 50 years later. This can also be seen in the selected metrics, from which some implement variations of McCabe's complexity measure.

Based on this complexity metric, another iteration was created just a few years ago by the team behind SonarQube [21], called *Cognitive Complexity*. In their 2018 Whitepaper "Cognitive Complexity: A New Way of Measuring Understandability", they emphasize the importance of complexity measures but argue that the original cyclomatic one has fallen out of date, mainly because of changes to programming languages and a mathematical model being unfit to adequately evaluate complexity [22]. To "remedy Cyclomatic Complexity's shortcomings and produce a measurement that more accurately reflects the relative difficulty of understanding, and therefore of maintaining methods, classes, and applications", they proposed their updated complexity metric, which aims to evaluate code based on its linear flow. This, in essence, means that they have implemented a system that assigns a higher complexity Score to code that is harder to read. For example, nested loops or if-statements slow code flow, and therefore increase the score.

Another older, yet very relevant work is "A metrics suite for object-oriented design" by Chidamber and Kemerer [23], worked out in the early 1990s. In it, they concerned themselves with the then newly upcoming object-oriented design approach of programming and created six metrics for it, which are still widely used today. Those metrics are: *Weighted Methods Per Class* (WMC), *Depth of Inheritance Tree* (DIT), *Number of children* (NOC), *Coupling between objects* (CBO), *Response For a Class* (RFC), and *Lack of Cohesion in Methods* (LCOM).

Descriptions for each of them would exceed the scope of this work; curious readers are referred to the original paper for further details. However, it should be mentioned that the metric *Number of Children*, has been used in this project.

Similarly to McCabe and the SonarQube team around Campbell, Hassan [24] created a metric to predict faults by evaluating complexity. In his work, he, however, chose to focus on code changes, instead of the whole corpus of code. For this, he created a metric called *Code Entropy* to evaluate the complexity of a given code change, with complex changes increasing the entropy more, and higher entropy being an indicator for faults. They found that "predictors based on our change complexity models are better predictors of future faults in large software systems in contrast to using prior modifications or prior faults".

Another way to predict errors in Software was proposed by Graves et al. [17]. Their model, similarly to Hassan's, did not look solely at the code itself for the prediction, but rather at the history of changes in it, in turn resulting in a new metric called *Code decay*. In it, they found that "the number of times code has been changed is a better indication of how many faults it will contain than is its length".

Two more higher-level reviews of a multitude of metrics have been conducted in recent years by Choudhary et al. [16] and D'Ambros et al. [18]. Choudhary et al. focused their work on change metrics, which include the likes of Hassan and Graves et al., proposing new change metrics in it and finding that "high-performance models can be built with several change metrics". They also found significant increases in recall in their models when using their newly proposed change metrics, which for example included *LOC-WORKED-ON* and *AVG-CODECHURN*, which focused on each developer's impact.

D'Ambros et al. presented a benchmark for defect prediction and used it to evaluate and compare well-known bug prediction approaches, as well as new ones they developed. The tested metrics include ones already touched upon in this section like ones by Chidamber & Kemerer [23], Hassan [24] or Graves et al. [17]. Their findings showed "that the best performing techniques are WCHU (Weighted Churn of source code metrics) and LDHH (Linearly Decayed Entropy of source code metrics), two novel approaches that we proposed". Furthermore, they stated that "past defects and source code metrics are lightweight alternatives with overall good performance" and "prediction techniques based on a single metric do not work consistently well".

Those two studies are highly relevant, as part of this work aims to determine which metrics are best suited for fault prediction, see RQ2 in section 1.2, using feature importance analysis on the trained model.

2.2 Machine Learning approaches

Machine learning has a well-established history of being utilized as a tool to detect bugs in software. During the literature review, one of the earliest notable and relevant studies in this field was identified, dating back to 2006. This highlights the long-standing interest in leveraging machine learning techniques to enhance software quality and reliability. Researching related works for this part presented some obstacles. During the review, numerous papers were encountered that appeared to closely align with this approach. However, these often relied on datasets that were either of low quality or not readily comparable to this one. This issue likely arises from the absence of standardized, central datasets in this domain, which continues to hinder consistent benchmarking and comparison across studies.

Another problem was that some works used different measurements for their results, for example not using accuracy, or achieving artificially high accuracies by using very imbalanced datasets. To elaborate briefly on this issue, naturally most Commits do not contain bugs, so if the datasets are not balanced, a model could simply guess all commits to be bug-free and achieve a high accuracy with that strategy. Despite these obstacles, several interesting and relevant papers were identified that align with the aim of this work. One such paper is the 2006 study by Kim et al. [25], which aimed to classify software changes as either clean or buggy. To achieve this, they used a machine learning classifier, just like in this work, to assess whether "a new software change is more similar to prior buggy changes, or clean changes". Using this approach, "predictions can be made immediately upon completion of a change", in contrast to the one proposed in this work, which looks at files instead of changes and needs metrics to be calculated on them first.

In "Reducing Features to Improve Bug Prediction" [26], two of the original authors from the previous Paper, together with Shivaji and Akella, improved and iterated upon that work, by proposing a feature selection technique for their classification problem, to improve performance. They found improvements in quality and performance in comparison to the old work.

Giger et al. created a machine learning approach for bug prediction with precision and recall of 84% and 88% respectively, by focusing on each method individually, instead of a whole file or class [14]. They argued that the finer granularity is preferable since it helps developers save time in finding bugs, especially since typically larger files are more bug-prone. For their models, which included a Random Forest algorithm, also used in one of the initial models in this work, they utilized change and source code metrics as input data. Their results confirmed earlier findings, specifically that change metrics outperform source code metrics.

In their work titled "A Developer Centered Bug Prediction Model", Di Nucci et al. [15] proposed a bug prediction model that emphasizes the role of developers in the software development process. As the title suggests, their approach focuses on the developers' perspective, particularly on their behavior during code changes. A central concept introduced in their work is *scattering*, which refers to the extent of a developer's focus dispersion while making modifications. This insight highlights how developer-related factors can significantly impact bug prediction models. Said *scattering* value was computed by looking at Where changes happened and whether they were close to each other (in file structure), as well as What changes occurred, namely whether they were semantically similar or not. Using these newly proposed predictors, they found a noticeable improvement in bug prediction models, where they combined them with regular predictors.

The last two machine learning approaches involve examining two papers in which the authors utilized a slightly different ML technique compared to the previously mentioned works, namely Deep Learning. These two works are "DeepBugs: A Learning Approach to Name-Based Bug Detection" by Pradel and Sen [27], and "Automatically Learning Semantic Features for Defect Prediction" by Wang et al. [28]. In the former, the authors employ natural language elements within code, such as variable and function names, to train a Neural Network for bug detection. This technique, referred to as *name-based bug detection*, represents a relatively new approach to bug prediction. Unlike other machine learning studies reviewed thus far, their methodology involves artificially introducing bugs by altering names or identifiers within the code. While this approach limits the scope of their model to detecting only three specific types of faults and makes their results incomparable to other studies, it offers a novel solution to the persistent issue of data imbalance in bug prediction research. Similarly to DeepBugs, Wang et al. also used Deep Learning to learn from semantic representations from source code. They used traditional Natural Language Processing techniques to generate Features for their model. They found that "learned semantic features could significantly improve both within-project and cross-project defect prediction compared to traditional features". Those two works and their findings could present Deep Learning as a possible technique for bug prediction in the future, but more research is needed to see whether they outperform traditional ML approaches, for which again a standard dataset would be integral.

Lastly, it is important to highlight a paper that does not fit neatly into the two categories detailed above. Specifically, "Root Cause Analysis of Software Aging in Critical Information Infrastructure" by König et al. [29]. This work, authored by members of the team collaborating on the overarching project at SBA Research, laid the foundation for the broader research initiative, and by extension, this thesis. The paper focuses on a critical issue in software engineering: the gradual decline in software quality over time, known as software aging. It delves into the root causes of this phenomenon, identifies factors contributing to it, and proposes methods for assessing its impact. Through their analysis, the authors offer valuable insights into the challenges of maintaining software reliability and performance in long-lived projects. The findings presented in this work served as an inspiration for the overarching project, which aims to tackle software aging by developing methods and tools to counteract its effects. Consequently, this thesis builds upon that groundwork by addressing one specific facet of the problem, namely the prediction and detection of software bugs as a means of mitigating quality degradation.

CHAPTER 3

Methodology

To address the research questions outlined in the previous chapter, construct a reliable bug prediction model, and document the findings effectively, this thesis followed a structured and methodical approach. This chapter details the methodology employed throughout the project.

The process began with an extensive review of existing literature in the field. This step was crucial for building a strong foundational understanding of relevant concepts, identifying gaps in the research, and pre-selecting approaches worth exploring. Following this, a thorough review of the ongoing project and an induction phase were undertaken. This facilitated gaining familiarity with the context, objectives, and prior work conducted as part of the broader research initiative.

Building upon this foundation, a prototype for a supervised machine learning (ML) model for bug prediction was developed. This served as a starting point for iterative experimentation, where the results of the initial prototype were analyzed to identify strengths and weaknesses. The model was progressively refined by experimenting with different algorithms and hyperparameter configurations to enhance its predictive accuracy and robustness. The main factor that was used as a measurement and tested for at this stage was the accuracy of the model.

Once a high-performing model was achieved, lessons learned from the iterative process were consolidated, and the best parameters and algorithms were applied to construct the final model. This model was then subjected to a feature importance analysis, providing insights into which metrics contributed most significantly to its predictions, thereby addressing one of the core research questions (RQ2).

As part of the model evaluation, a Feature Importance Analysis was conducted to identify which code metrics contributed most significantly to the predictions. This analysis provided valuable insights into the decision-making process of the machine learning models, highlighting the key features that played a critical role in determining whether a file was buggy. The findings from this analysis were also instrumental in addressing RQ2, which sought to identify the most reliable metrics for fault prediction.

The final steps of the methodology focused on comprehensive result analysis, validation, and comparison. These efforts aimed to answer additional research questions (RQ1 and RQ4) and ensure the reliability and applicability of the model. Throughout the entire process, meticulous documentation of findings was maintained, culminating in their presentation and synthesis within this thesis.

This structured methodological framework – which is visualized in Figure 3.1 – not only ensured a systematic approach to tackling the research objectives but also facilitated a clear and concise presentation of the results and their implications.

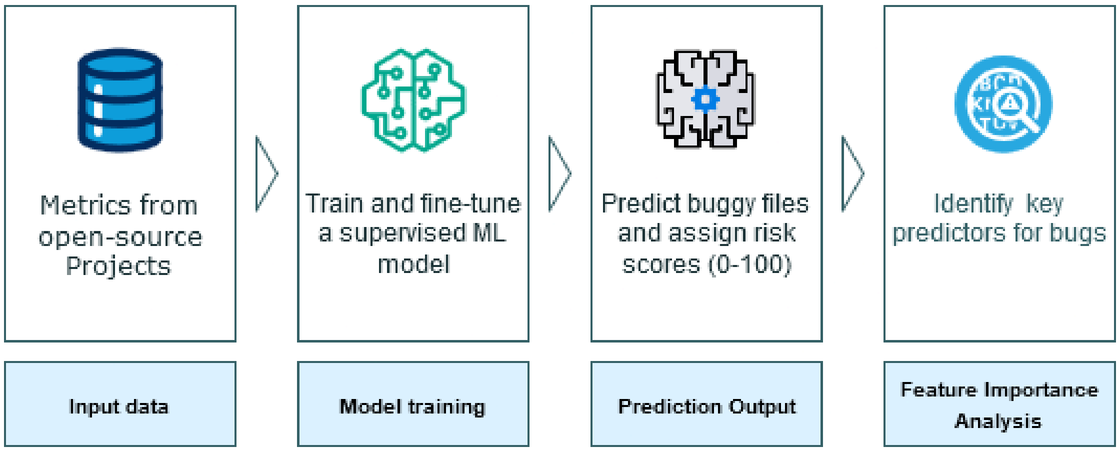
Metrics from open-source Projects
Train and fine-tune a supervised ML model
Predict buggy files and assign risk scores (0-100)
Identify key predictors for bugs
Input data
Model training
Prediction Output
Feature Importance Analysis

Figure 3.1: A Flowchart explaining the workflow.

The next sections provide an overview of the tools, techniques, and strategies used to develop, evaluate, and refine the solution, ensuring it is both effective and grounded in rigorous processes. The overview begins with an introduction to the dataset central to this research. The data is analyzed to understand the underlying projects from which it was sourced, as well as the specific metrics selected as features for model training. This foundational step is crucial to grasping the context in which the solution operates and the data it relies upon. Lastly, the discussion delves into machine learning, starting with a concise review of its principles and mechanics. This introduction serves as a springboard into a discussion of the particular machine learning methods and models applied within the work. Each relevant component and use case is detailed, illustrating how these techniques were tailored to address the unique needs of the project.

It is essential to note that this thesis was developed in collaboration with the Complexity and Resilience Research group from SBA Research [30], contributing to a larger project on Software aging and its impact on quality. This context is reflected in the dataset used here, which had undergone preprocessing by team members prior to its application in this work. Specifically, earlier phases of the project analyzed the source projects within the dataset, calculating various metrics that form the foundation of the dataset.

With these preliminaries in mind, this chapter begins by examining the dataset, exploring its structure, and analyzing the projects and metrics it encompasses.

3.1 Dataset

At the foundation of this project and its machine learning pipeline lies the dataset, serving as the cornerstone for training and evaluating the model. Understanding the composition of this dataset is crucial, as it provides insights into the factors and data points that collectively form the training data.

In general, the dataset consists of data derived from commits across various open-source software projects. This information is stored in a single `.csv` file, which can be found in the project folder under `DATA/dataset_unsplit.csv`. Each entry in the file contains the following fields: the name of the project, the commit hash, the commit date, a file ID, the file name, a label indicating whether that entry represents a bug fix or refactoring, and a collection of code metrics.

Details regarding the specific projects and metrics used will be discussed in subsequent sections; therefore, they are omitted here for brevity.

It is important to note how this input is utilized by the model. The `isBugfix` column serves as the target variable, which the model aims to predict. To achieve this, the model uses the values of the code metrics as input, applying machine learning techniques to identify patterns and relationships between these metrics and the target variable within the training data. This enables the model to make accurate predictions on new data based solely on the provided code metrics.

Crucially, the target value is `isBugfix`. This means that whether a commit was a bugfix is used as a proxy for the presence of a bug. In large datasets like this one, it is virtually impossible and computationally unfeasible to find the root commits for bugs. As a result, Natural Language Processing is applied to commit messages to identify words such as 'bug' or 'bugfix,' enabling the detection of bugs and, in turn, the training of a model to recognize them.

However, the process of training the model and utilizing the dataset will be discussed in detail in the implementation section. However, before delving into these specifics, it is crucial to introduce the individual metrics and projects that constitute the dataset, starting with the code metrics.

3.1.1 Description of Metrics

The code metrics that make up the dataset of this work have been mentioned frequently already, but without much detail or proper explanation. This section addresses that by first providing an overall description of what a code metric is, then listing the metrics used, and finally analyzing and explaining one example in greater depth.

What are Code Metrics

Simply put, metrics are measurements over code, often either on file- or class-level, and aim to express a particular value or state of said code. They can range from quite simple, like for example the *Lines of Code* metric, which simply counts the number of code lines in a file, to more complex values, for instance about class cohesion or coupling, as proposed by Chidamber & Kemerer [31]. They can be used to express several different things, so most importantly, it is only to know that they give code a numerical value regarding metric-specific criteria. Code metrics have been around for a long time and are continuously worked on and improved on, as well as new ones being proposed. Many of them serve as a measure of code Quality in popular IDE Plugins or external programs like SonarQube [21].

As previously mentioned, these metrics are sometimes capable of indicating whether a bug is present on their own. However, before exploring how they are utilized in this project and the role machine learning plays in this context, it is important to review the various metrics used in this work.

Metrics used

In total, 37 different code metrics were used for the dataset in this project. The relatively high number was chosen to be able to feed the model a maximum amount of information on the one hand and make it less reliant on one singular metric on the other. The complete set of metrics utilized in this thesis is outlined in the following table:

Table 3.1: Overview on the Metrics used in the Dataset.

Metric Type	**Metric Name**	**Description**
Social	dokSum	Sum of Degrees of Knowledge.
	dokAvg	Average of Degrees of Knowledge.
	dokMed	Median of Degrees of Knowledge.
	dokStd	Standard deviation of Degrees of Knowledge.
	dokMax	Maximum Degree of Knowledge.
	dokMin	Minimum Degree of Knowledge.
	dokSumChurnPartial	Partial sum of Degrees of Knowledge weighted by churn.
	dokMinChurnPartial	Partial minimum of Degrees of Knowledge weighted by churn.
	dokSumChurnWeighted	Weighted sum of Degrees of Knowledge based on churn.

Metric Type	Metric Name	Description
	bus1	Bus factor (i.e. number of people with crucial knowledge of a file).
	bus0.9	Bus factor with lower DoK.
	bus1.1	Bus factor with higher DoK.
	projectBus	Bus factor on project-level.
	projectBus0.9	Project bus factor with lower DoK.
	projectBus1.1	Project bus factor with higher DoK.
Process	isBugFix	Indicator for bug-fix commits.
	isRefactor	Indicator for refactoring commits.
	totalRefactors	Total count of refactoring commits.
	Revision	Revision number of the code.
	Age	Age of the code in terms of time.
	Number of Authors	Total number of authors who contributed.
	totalLoc	Lines of code in total.
	locAdded	Lines of code added.
	locRemoved	Lines of code removed.
	locReplaced	Lines of code replaced.
	locModified	Lines of code modified.
	relativeLocAdded	Percentage of lines added.
	relativeLocRemoved	Percentage of lines removed.
	relativeLocReplaced	Percentage of lines replaced.
	relativeLocModified	Percentage of lines modified.
Code Change Entropy	Hcpf1	History Complexity Period Factor 1.
	Hcpf2	History Complexity Period Factor 2.
	Hcm1	History of Complexity Metric 1.
	Hcm2	History of Complexity Metric 2.
	Edhcm	Exponentially Decayed History of Complexity Metric.
	Ldhcm	Linearly Decayed HCM.
	Lgdhcm	Logarithmically decayed HCM.

The **Degree of Knowledge** metric measures how familiar a developer is with a particular file or module by analyzing their past contributions. A lower value indicates less familiarity, which may correlate with a higher likelihood of bugs due to reduced context or understanding. The version of it in this dataset is a lightly adapted one of the original [32]. **Churn** quantifies the amount of code that has been modified within a specific timeframe. High churn can signal instability or frequent rework, often associated with increased bug density [33]. The **Bus Factor** evaluates the risk of knowledge loss by estimating the minimum number of developers whose absence would critically impair project progress. A low bus factor indicates that knowledge is concentrated in very few individuals, which can lead to maintainability issues [34]. The **History Complexity Period Factors** assess the complexity of changes made to a file over a specified period, focusing on how often and how extensively a file has been modified. Files with a high history complexity are often more error-prone due to frequent modifications [24]. Finally, the **Exponentially Decayed History of Complexity Metric** prioritizes recent changes by assigning higher weights to recent modifications and lower weights to older ones. This metric captures the evolving complexity of a file, providing insights into its current maintainability and potential bug risks [24].

3.1.2 Description of Projects

In order to accomplish optimal training conditions and provide a variety of data from different software development environments and real-world applications, a wide-ranging set of Software projects is used as input for the model. A diverse dataset comprising 34 open-source projects is utilized, spanning multiple domains, including medical software, integrated development environments like Eclipse, and various other categories, with a wide spread in age and size.

In this list, all projects along with their domains are presented:

Table 3.2: Overview of the Projects used in the Dataset.

Project Name	**Domain**
eclipse-mylyn/org.eclipse.mylyn	Development Tools
eclipse-equinox/equinox	Development Tools
pallets/flask	Web Development
eclipse-jdt/eclipse.jdt.core	Development Tools
apache/flink	Data Processing
apache/lucene	Data Processing
vercel/next.js	Web Development
eclipse-pde/eclipse.pde	Development Tools
PaperMC/Paper	Other (Game Server)
TanStack/query	Web Development

Project Name	**Domain**
termux/termux-app	Other (Mobile App)
ehrbase/ehrbase	Healthcare
openmrs/openmrs-core	Healthcare
apache/ctakes	Healthcare
Bahmni/bahmni-core	Healthcare
b2ihealthcare/snow-owl	Healthcare
synthetichealth/synthea	Healthcare
informatici/openhospital-core	Healthcare
keycloak/keycloak	Security
apache/tomcat	Web Development
zaproxy/zaproxy	Security
cve-search/cve-search	Security
intelowlproject/IntelOwl	Security
yeti-platform/yeti	Security
cuckoosandbox/cuckoo	Security
smicallef/spiderfoot	Security
SigmaHQ/sigma	Security
honeynet/honeyscanner	Security
mitre/caldera	Security
alibaba/fastjson	Data Processing
google/nomulus	Other (Domain Management)
google/error-prone	Development Tools
google/j2cl	Web Development
AzureAD/microsoft-authentication-library-for-android	Security

The statistics in Table 3.3 provide an overview of the above-mentioned projects, their contents & various project details.

As can be seen in these statistics, the used dataset truly is diverse and covers a wide range of fields, use cases, languages, activity, age, and other factors. Through this, it is ensured that the training data is as unbiased as possible, with the aim of high accuracy for the model when it encounters new data.

Having presented and analyzed the dataset that lays the foundation for this thesis, the focus now shifts to machine learning, its relevance today and in the given context, as

Table 3.3: Summary of Project Characteristics

Category	**Subcategory**	**Number of Projects**
Project Age	Mature (older than 10 years)	11
	Growing (5 to 10 years)	15
	Young (5 years or younger)	8
Contributor Base	Large (more than 150 contributors)	13
	Moderate (51 to 150 contributors)	14
	Small (50 or fewer contributors)	7
Activity Level	Highly active (>20k commits)	7
	Moderately active (5k-20k commits)	11
	Low activity (<=5k commits)	16
Release Frequency	Frequently released (>100 releases)	4
	Moderately released (10-100 releases)	15
	Infrequently released (<=10 releases)	15
Language	Java	23
	Python	8
	JavaScript	2
	TypeScript	1
Thematic Category	Development Tools	5
	Web Development	5
	Data Processing	3
	Healthcare	7
	Security	10
	Other	4

well as the decisions that shaped the type of model ultimately used.

3.2 Machine Learning in the given Context

Although machine learning has been briefly introduced in earlier sections, its role, principles, and broader implications have not yet been thoroughly explored. This section delves into the fundamentals of machine learning, outlining its theoretical underpinnings, its growing significance in contemporary technology, and the distinct advantages it brings to the table. By establishing this foundational understanding, the application of machine learning within this work is contextualized.

The discussion begins with an explanation of why machine learning was chosen as the core of the solution, addressing both its strengths and the challenges it presents, such as common pitfalls and limitations that must be considered during development. Additionally, an overview of the different types of machine learning approaches is provided, with a focus on those relevant to this project, and an explanation of how these methods align with the project's objectives.

Following this, the focus shifts to the approach and actual implementation of the solution.

3.2.1 Relevance of Machine Learning for this problem setting

Since the early 2010s, machine learning has steadily gained momentum, with its usage expanding significantly across an increasing variety of fields. The rapid advancements in computing power, combined with the vast and growing amounts of data available, have made machine learning a viable option for many tasks, often offering the most effective solution. Through this rise in feasibility and hence popularity, machine learning is now used in a wide range of application scenarios, including Self-Driving Cars like for example the Robo-Taxi Company Waymo [35], tumor diagnosis in cancer patients with significantly higher accuracy than even seasoned professionals [36], or increasingly complex trading algorithms for Investment Banks [37]. With the rapid rise of AI tools such as OpenAI's ChatGPT [38] and Google's Gemini [39], as well as the shift by major smartphone manufacturers like Apple and Google to integrate on-device AI models into their products, machine learning has become increasingly prominent in daily life. By leveraging AI assistants and tools as unique selling points — for example, Apple Intelligence [40] — these companies highlight the central role that machine learning now plays in various aspects of modern technology.

3.2.2 Advantages of Machine Learning & Appeal for this Application

Before discussing why machine learning is particularly suitable for this use case, it is important to consider, in broad terms, why ML has become applicable across such a wide range of diverse scenarios. At its core, the strength of machine learning lies in its ability to recognize patterns in large datasets that are representative of the problem at hand. Leveraging a variety of algorithms and techniques, ML models can analyze data to uncover complex relationships that may be imperceptible to humans due to the sheer number of interacting variables. This ability to handle and process high-dimensional data enables ML models to provide accurate predictions and classifications across a variety of domains and applications.

That is precisely the point of transition to the task at hand. As mentioned earlier, the datasets consist of a large number of data points – different files from various projects at different points in time – assessed through several code metrics, totaling 37 variables or features. The aim of using a machine learning model to analyze them is the following: It is known that there are some metrics out there that somewhat reliably indicate whether a bug is more or less likely to occur. However, nearly all of these metrics come with their specific drawbacks. Many are essentially just more complex versions of the *Lines of Code* metric [19][41], which, while it can be a useful bug indicator, has limitations. After all, a file with 100 lines of code is statistically more likely to contain bugs than one with just 10 lines. Yet, a large file does not necessarily imply poor code quality. For example, think of a file with only the following command, repeated 100 times: print("Hello World"). Whether this is done in a for-loop or copied and pasted a hundred times changes nothing, but the *Lines of Code* metric would imply it does, which is exactly where it can fall short sometimes. So to get back to the point, the aim of the use of an ML model in this work is to extract the best and most important information from many metrics and cross-referencing them to avoid their pitfalls.

To illustrate the importance of leveraging multiple metrics, consider the following example: One file is under analysis with ten metrics, including a key metric for this example, called Metric A. Metric A has an accuracy score of 70%, which means it provides useful – but not definitive – insight into whether the file contains a bug. However, not all metrics contribute equally to the prediction. Some metrics may capture patterns that are highly relevant to the presence of bugs, while others may provide little to no meaningful signal.

For instance, in diagnosing a complex problem, such as identifying a brain tumor, measurements like bone density or blood sugar levels are unlikely to contribute significantly to the outcome. Similarly, in bug prediction, certain code metrics might provide critical information about software reliability, while others may not be as informative. Therefore, the goal is not merely to increase the number of metrics used but to ensure that the chosen metrics are relevant and complementary.

By carefully selecting and combining metrics that offer meaningful contributions to the problem, the aim is to capture diverse perspectives on software quality. This approach enhances the model's ability to identify trends and correlations that individual metrics might miss, ultimately improving the overall predictive performance.

3.2.3 Pitfalls

Like many tools, machine learning is not a one-size-fits-all solution; there are several critical considerations and potential pitfalls that need to be addressed to ensure successful outcomes. In this section, two common pitfalls in machine learning will be discussed — overfitting and data imbalance — that were also encountered during this work. These issues are not unique to this project; they are well-known challenges that often require thoughtful mitigation strategies to avoid negatively impacting model performance and generalizability. Addressing these pitfalls was essential for achieving reliable results and serves as an important lesson for future applications.

Overfitting

First and foremost it is important to not overfit the model. This is a concern that is prevalent in (almost) every machine learning solution. Going back to the line fitting example, the problem would be if the Model defines the line in such a way that every single data point is exactly on the line (or curve). Of course, when testing with the same data, it would lead to an Accuracy score of 100%. But if then afterward the same model is fed new data, chances are very high that it will mislabel or misclassify a bunch of it, and the accuracy score will fall rather sharply. To combat this, the training data is split into a training and a test set. The model is first trained and configured on the training set. Once training is complete, it is then evaluated on the test set — data it has not previously encountered, keeping it 'fresh' for the model. The accuracy score is calculated based on this evaluation. That is the rough idea of it, how exactly it was done in this project will be described further in chapter 4, 'Approach'. To summarise, the aim is to be as specific to the training set as possible, without a loss in generalization which would lead to the mislabelling of fresh data.

Data Imbalance

Another problem to look out for, which is more specific to this particular use case, is Data Imbalance. When looking at the dataset of course most of the files do not contain bugs. It would indicate a poorly managed software project to observe an even split between buggy and non-buggy files. In reality, the dataset, which consists of multiple different projects to better average the values, shows that around 18.5% of files contain bugs.

To illustrate what exactly the problem with that is, the line-fitting example is revisited. For simplicity's sake, the dataset is assumed to be even more lopsided and contains 99% bug-free files. Naturally, a machine learning model would look at this and then in turn simply guess for all files that they are bug-free. Assuming a random split of train and test data, this would lead to the model being accurate in 99% of the cases. At first, that score looks incredibly good, but in reality, the model would be useless, since it never correctly predicts a bug. To address this, the focus was laid on Oversampling and Undersampling, techniques commonly used in the industry. How exactly those were implemented, what they achieved, and how they work will be discussed in chapter 4 later on. For now, it suffices to know that the objective of both approaches is to balance the dataset, meaning a 50/50 split of bugs to non-bugs would be achieved.

3.2.4 Supervised vs. Unsupervised

One of the Questions of this thesis, RQ3, was a comparison of a supervised and an unsupervised machine learning model for the task at hand. In the end, it was decided to not pursue that further, the reasoning is explained in the following. But before doing that, it is important to quickly recap and highlight what in fact are the differences between the two approaches.

Differences

If you look online and in various publications you will find that more or less everybody agrees on the main difference: the labeling of data (see e.g. Google or IBM's definition [42] [43]). Supervised Learning works with labeled in- and output data, whereas Unsupervised does not. This in turn makes one or the other more suited for differing use cases. For example for tasks like clustering or exploratory data analysis, Unsupervised Learning is generally regarded as more suitable, while Supervised Learning often is the choice for classification or regression tasks. In general, there is also a difference in how the model interacts with the data. Google has put it quite well in their comparison of those two: "With supervised learning, an algorithm uses a sample dataset to train itself to make predictions, iteratively adjusting itself to minimize error. These datasets are labeled for context, providing the desired output values to enable a model to give a 'correct' answer. In contrast, unsupervised learning algorithms work independently to learn the data's inherent structure without any specific guidance or instruction. You simply provide unlabelled input data and let the algorithm identify any naturally occurring patterns in the dataset." [42]

Furthermore, there are other differences as well of course, such as the Goals, Applications, or Complexity. For example, Supervised Learning can be more resource-extensive, since it requires more user input and labeling of the data beforehand, while "unsupervised learning methods can have wildly inaccurate results unless you have human intervention to validate the output variables." [43]

Naturally, there are more differences and nuances between the two approaches, especially when looking at commonly used Algorithms for each, but for the scope of this work, this level of understanding suffices.

Decision for Supervised Learning

Having those differences in mind and taking a step back and looking at the dataset described above, it is obvious that for this task, namely bug prediction through the analysis of multiple metrics, only a Supervised approach makes sense. First of all, the data has of course been labeled and that plays a big part in the analysis of it and the prediction made by it. And secondly, the task at hand is clearly a classification task, in that the aim is to classify the data into either bug or Non-bug. Given these reasons, it is clear that adopting an unsupervised approach offers no advantages, and does not align well with the problem, therefore, RQ3 was not explored further.

3.2.5 Classification vs. Regression

Now that the main, high-level differences between supervised and unsupervised learning have been established, along with the reasoning behind the choice of the former for this thesis, the focus shifts to a deeper analysis of the two main types of problems it addresses: classification and regression. This involves examining the definition and use cases of both, as well as reviewing a selected number of popular algorithms for each type that were considered for this work. Following this analysis, the emphasis placed on one type during this work should become evident, concluding with a brief rationale for the choice made.

Classification

In classic classification tasks, the model aims to analyze input data and predict which labels are assigned to which data points, ergo classifying them. This can be a simple binary classification like in this task, bug vs. Non-bug, or for example, a visual model deciding whether the animal in a given image is a cat or a dog, or for several labels, like in this study by LeCun et al. [44] where the model is tasked with digit recognition from handwritten input.

To achieve this, the models are provided with labeled input data, meaning data that includes the correct classifications, so they can train on it. In the training phase, they then try to find similarities between same-labeled instances, which allows them to make assumptions about given data points. Moreover, it is also important to find distinguishing factors between the data points that make out the different labels.

With these two considerations in mind, the study by LeCun et al. on handwritten digit recognition is revisited. To do this, the model could, for example, analyze that examples with one or more circular parts (think for example like the two halves in a '3') tend to be very seldom labeled as a '1' or '4', whereas they are often labeled '0', '3' or '8'. This then can be remembered and in combination with several different rules or patterns be applied to data to make predictions. Importantly, the feature referred to as 'circular parts' represents a combination of multiple variables. For instance, if the model treats the black/white value of each pixel as a separate variable, the feature is derived from the relationships between these individual pixel values, rather than being a single independent variable.

Applying this concept to the dataset at hand, it could be observed that if a code file is simultaneously old, lengthy, and has many contributors, it is more likely to be buggy compared to a file where only one of these factors is present. This demonstrates how combining multiple variables can yield more predictive power than analyzing them in isolation.

These examples provide a simplified, high-level view of how classification tasks work. In practice, machine learning models employ advanced mathematical techniques and algorithms to identify patterns and relationships in data. These patterns often involve higher-dimensional combinations of variables, sometimes incorporating 30 or more features, which are far more complex than the simplified scenarios described here. However, explaining these workings in detail would exceed the scope of this work and is not its primary aim, so this level of understanding is sufficient.

Several techniques can be employed in classification. Providing detailed explanations and use cases for all of them would exceed the scope of this work. Instead, a few are listed below, along with a reference to an excellent resource for in-depth explanations. The focus then shifts to an analysis of the two approaches selected for this project. A non-exhaustive list of classification methods includes: Decision trees, Perceptrons, Bayesian Networks, Instance-based learning, and Support Vector Machines.

For detailed reviews of these techniques and their respective advantages and disadvantages, "Supervised Machine Learning: A Review of Classification Techniques" by Kotsiantis et al.[45] is recommended. Furthermore, those approaches of course can also be further divided down and implemented in different ways and Algorithms. For this work, Random Forest and CatBoost classifiers were chosen from popular Python libraries, which both represent some sort of implementation of Decision trees. Both methods are now examined to establish a basis of understanding for the 'Approach' chapter later on.

In the beginning, an approach using the Random Forest library by scikit [46] was chosen. According to its inventor, Leo Breiman, Random Forests "are a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest" [47]. This means that the algorithm randomly samples Decision Trees and through that, paired with some optimization, tries to find the best tree/solution.

This proved to be effective in the beginning and, especially with the great documentation from scikit, served as a good stepping stone. But with time problems arose, namely performance issues and GPU support. For that reason, CatBoost was tried next. Since it provided better results it then quickly took over as the main approach in this thesis.

CatBoost uses gradient boosting for decision trees to iteratively improve results. Gradient boosting simply means that "the learning procedure consecutively fits new models to provide a more accurate estimate of the response variable. The principle idea behind this algorithm is to construct the new base-learners to be maximally correlated with the negative gradient of the loss function", as explained by Natekin and Knoll [48]. The idea goes back to 1999 by Jerome H. Friedman [49], but the exact mathematical details are not of concern to us.

CatBoost was developed by Dorogush et al.[50] and is nowadays one of the most frequently used machine learning frameworks, as can be seen on Kaggle's Developer survey [51]. One of its major advantages is its handling of categorial features, hence the 'Cat' in its name, which proved to be quite useful for the use case at hand. As already mentioned, major performance increases were witnessed, provided by the GPU support of CatBoost, as well as simply better overall results (higher accuracy), which led to the focus on a CatBoost-based approach. More on that in the next chapter.

Regression

Regression is used to predict continuous values, unlike classification, which assigns discrete labels. In this work, two regression techniques were considered: ElasticNet and Logistic Regression, both implemented using the scikit-learn library [52, 53].

Logistic regression, despite its name, is widely used for binary classification tasks and was chosen as one of the approaches due to its suitability for the given problem. These techniques differ primarily in how they apply penalties to their functions, which influences their handling of multicollinearity and regularization. Further details on their application are discussed in the next chapter.

To conclude this section, it is important to reflect on the overall machine learning approach taken in this thesis. While regression techniques were considered and applied for specific purposes, the primary focus remained on classification, given the nature of the problem.

Because the core aim of this work is to analyze code files on whether they contain bugs or not, ergo wanting to classify them into two categories, it clearly represents a classification task. That is why the main focus of this thesis has been spent on the implementation of such an algorithm, in fact, multiple ones. As already mentioned above, in the end, a CatBoost classifier was chosen.

Resources were also allocated to developing a regression-based solution for two main reasons. First, to explore how well an adapted version of a regression algorithm would perform for this task. Secondly, to enable a comparison between the two approaches and establish an important baseline for the classification solution.

Furthermore, there was also interest in the calculation of a Risk Score for given code files, a Score from 0 to 100 on how likely a bug is to occur in it. Since the output values of that task are continuous, this seemed like more of a regression task at heart and provided another reason for incorporating such an Algorithm as well.

CHAPTER 4

Approach

With the theoretical groundwork established in the previous chapter, this chapter shifts to the practical implementation of the solutions developed for fault prediction. The aim is to provide a detailed explanation of the methodologies employed, the steps taken, and the rationale behind the choices made.

At its core, this chapter follows a structured workflow that applies machine learning techniques to classify files as buggy or non-buggy and to assign Risk Scores for prioritizing code reviews. The process begins with **data loading and preprocessing**, laying the foundation for subsequent steps. This is followed by **model selection and training**, where the key machine learning algorithms – Decision Trees, Random Forest, ElasticNet, Logistic Regression, and CatBoost – are explained conceptually and implemented practically. Each algorithm builds on the foundational principles introduced earlier, with their selection motivated by specific use cases and comparative advantages.

The workflow can be summarized as follows:

1. **Data Preparation**: Preprocessing, feature engineering, and train-test splitting to ensure the model is trained and evaluated on clean, unbiased data.

2. **Algorithm Implementation**: Iterative development and parameter tuning for each machine learning approach, starting with simpler models and progressing to more complex ones.

3. **Model Evaluation**: Rigorous testing using accuracy, confusion matrices, and additional performance metrics such as precision and recall.

4. **Risk Score Calculation**: Extending the classification model to generate a continuous numerical Risk Score for further insights.

The chapter is structured to mirror the chronological order of development. It begins with foundational elements common to all solutions, such as data preprocessing, and progresses through the iterative improvements made to each algorithm. The **Random Forest Classifier** serves as the starting point, providing insights into early-stage challenges and solutions. The focus then shifts to the **CatBoost Classifier**, which forms the core of the work and demonstrates superior performance. After, the **Risk Score** methodology and its integration with visualizations are presented, showcasing the flexibility and interpretability of the developed solution. The chapter also covers the regression-based approaches (ElasticNet and Logistic Regression) and their role in benchmarking the primary classification model. Then, the methodology of how the models were tested and evaluated is explained. Finally, the chapter is rounded off with sections about implementation details, like model testing code, and some obstacles that were encountered during the whole process

By the end of this chapter, the reader will have a comprehensive understanding of the step-by-step process undertaken to develop, implement, and refine the solutions, as well as the obstacles encountered and the strategies employed to overcome them.

4.1 Foundational Elements

This section outlines the foundational steps common to all solutions developed in this work, including data loading, preprocessing, training, and evaluation. These elements provided the groundwork upon which each algorithm-specific approach was built.

Data Loading and Preprocessing

The dataset was loaded and prepared using the Python library *pandas* [54]. Columns irrelevant to fault prediction, such as *project* or *commit hash*, were dropped to avoid introducing unnecessary noise. The features (X) and labels (y) were separated, ensuring the model could not 'see' the correct label during training. An 80/20 train-test split was applied using the *train_test_split* function from *scikit-learn* [46], reserving 20% of the data for evaluation. This step was crucial to ensure that the calculated accuracy reflected the model's performance on unseen data.

Training and Evaluation

Over 100 models were trained throughout this work, leveraging systematic hyperparameter tuning and testing. The training phase involved fitting the model to the training data, while the evaluation phase required the model to make predictions (y_{pred}) on the test set (X_{test}). The predictions were then compared to the actual labels (y_{test}) to calculate the accuracy score, which provided a percentage of correctly predicted labels. This evaluation metric, along with other measures such as confusion matrices, precision, and recall, played a pivotal role in assessing the quality of each model and guiding further optimization.

The model predictions on unseen data ensured robust evaluation and helped prevent overfitting. By splitting the data and iterating over different test/train splits, variability in the results was reduced, and the models' generalizability was enhanced. This practice was consistent across all developed solutions.

Post-Run Outputs

After training and evaluation, the following outputs were generated for each run:

- **Model Saving:** Trained models were saved to facilitate reuse and quick retraining with new datasets.
- **Confusion Matrix:** A confusion matrix was created to visualize true positives, false positives, true negatives, and false negatives. This helped quickly assess the model's strengths and weaknesses, particularly its ability to balance precision and recall.
- **Results Documentation:** Key metrics and parameters, including the best model's accuracy, hyperparameters, and classification report (precision, recall, and F1-score), were saved in a structured results file for future reference.

A confusion matrix is a tabular summary of a classification model's performance, where the X-axis represents predicted labels, and the Y-axis represents actual labels. It breaks down predictions into four key categories: true negatives (correctly classified non-bugs) in the top-left quadrant, true positives (correctly classified bugs) in the bottom-right quadrant, false positives (incorrectly classified non-bugs as bugs) in the top-right, and false negatives (incorrectly classified bugs as non-bugs) in the bottom-left. The objective was to maximize the values of true positives and true negatives while minimizing errors from false positives and false negatives. During model development, confusion matrices, along with accuracy scores, offered a concise and visual means to assess predictive performance after each run.

Iterative Process

Throughout the development, the iterative nature of hyperparameter tuning and model optimization was key. Each run tested variations in parameters such as tree depth, number of iterations, and learning rate, among others, enabling a systematic comparison of configurations. Insights from over 100 trained models helped refine the approaches, leading to the selection of the best-performing solutions.

These foundational steps laid the groundwork for exploring the specific implementations and optimizations of each algorithm, beginning with the Random Forest classifier, after a short introduction to a key concept, Decision Trees, in the next section.

4.2 Decision Trees

Although Decision Trees were not directly employed as a standalone solution in this work, they form the conceptual foundation for more advanced models like Random

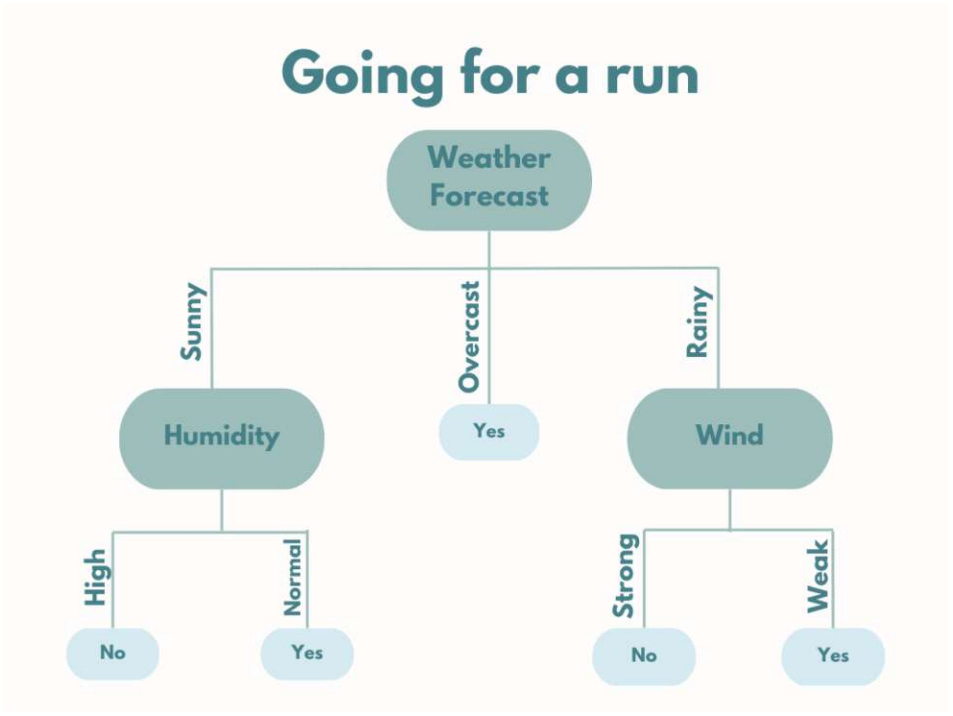

Figure 4.1: Simple Example on Decision Trees.

Forest and CatBoost, both of which build upon the tree-based structure. Understanding how Decision Trees operate is essential for comprehending the enhancements introduced by these ensemble methods. For instance, Random Forest combines multiple Decision Trees to reduce overfitting, while CatBoost incorporates gradient boosting to improve performance in handling complex, high-dimensional datasets.

What Defines Decision Trees?

Decision Trees are a fundamental and widely-used machine learning algorithm that operates by recursively splitting the dataset into subsets based on feature values. Each split is guided by a criterion, such as Gini Impurity or Information Gain, to maximize the separation of the target classes at each node. The tree structure comprises nodes, branches, and leaves, where nodes represent decisions based on feature values, branches denote the outcomes of those decisions, and leaves provide the final prediction [55].

How They Work

The algorithm begins at the root node, where it evaluates all possible splits of the dataset using a splitting criterion. The goal is to select the feature and threshold that best separates the target classes. This process is repeated recursively, creating a tree-like structure as the data is divided further. The algorithm continues until a stopping criterion is met, such as reaching a maximum tree depth, a minimum number of samples per leaf node, or a perfect classification of the training data. This process is visualized in

Figure 4.1 for easier understanding. There, the decision of whether to go for a run depends on a few factors regarding the weather forecast, ending in the leaf nodes with Yes or No as an answer.

Mathematically, the splitting criterion measures how well a split improves the homogeneity of the target variable within the resulting subsets. For example, Gini Impurity, a popular criterion, is computed as:

$$G = 1 - \sum_{i=1}^{n} p_i^2$$

where p_i represents the proportion of samples belonging to class i in a subset. A lower Gini Impurity indicates greater purity and, therefore, a better split [55].

4.3 Random Forest Classifier

4.3.1 Random Forest Algorithm

Random Forest is an ensemble learning algorithm commonly used for both classification and regression tasks. It operates by constructing a collection of decision trees during training and aggregating their outputs to make final predictions. The algorithm is based on the principle of *bagging* (Bootstrap Aggregation), which combines the predictions of multiple models to improve accuracy and robustness while reducing the risk of overfitting [47]. In Figure 4.2, the process is illustrated, showing how multiple decision trees contribute to finding different solutions. The final prediction is then derived by either averaging the individual predictions or applying majority voting, as depicted in the example.

Key Characteristics

One of the defining features of Random Forest is its ability to handle high-dimensional data and capture non-linear relationships between features. Each tree in the forest is trained on a random subset of the data, and at each node, a random subset of features is selected for splitting. This randomness enhances the diversity of the trees, improving the model's overall generalization performance [47].

Additionally, Random Forest provides feature importance metrics, making it an interpretable algorithm for identifying which input variables contribute most to the predictions. Its simplicity, efficiency, and ability to handle both numerical and categorical features make it a widely used algorithm in machine learning.

How It Works

Random Forest builds multiple decision trees by recursively splitting the dataset based on the feature that provides the best separation of the target variable. Each tree is trained on a bootstrap sample of the dataset, which is created by sampling with replacement. At

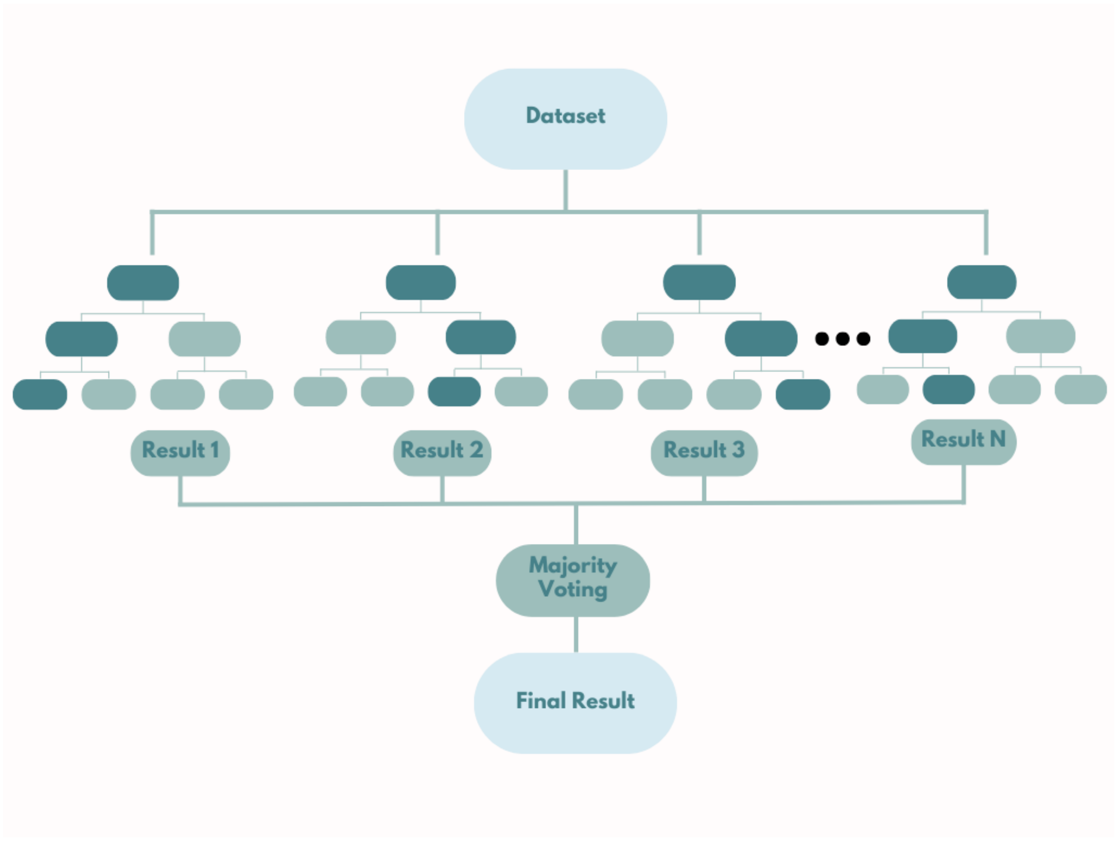

Figure 4.2: A simple Random Forest example, made up of multiple Decision Trees.

each split, only a random subset of features is considered, ensuring that the trees are not overly correlated.

During prediction, the model combines the outputs of all trees. For classification tasks, this involves majority voting, where the most frequently predicted class is selected. For regression tasks, the average of the outputs is taken as the final prediction.

Why Random Forest Was Chosen Initially

Random Forest was selected as the starting point for this work due to several reasons:

1. **Ease of Implementation and Familiarity:** The algorithm is straightforward to use and has well-documented implementations in libraries like scikit-learn [46], making it a convenient choice for initiating the fault prediction task.

2. **Baseline Performance:** Random Forest is known to provide solid baseline performance across a wide range of tasks [56]. It served as a benchmark to evaluate the feasibility of applying machine learning to the given dataset.

3. **Feature Importance Insights:** The ability of Random Forest to rank feature importance was helpful in gaining initial insights into which code metrics were likely to contribute most to bug prediction.

However, as the project progressed, it became evident that Random Forest was not the optimal solution for this task. While it delivered reasonable results, its performance lagged behind more advanced techniques like CatBoost. Specifically, Random Forest struggled with the large and diverse dataset, particularly in handling categorical features effectively and achieving the desired accuracy.

Transition to CatBoost

The limitations of Random Forest, particularly in terms of accuracy and scalability, prompted a shift to CatBoost. The latter offered superior handling of categorical features, higher predictive accuracy, and better GPU utilization, making it a more suitable choice for the project's objectives.

4.3.2 Implementation Details

The Random Forest approach was an early part of this project and served as an initial step before transitioning to CatBoost. The Random Forest classifier implementation included iterative hyperparameter tuning using Bayesian Search through the *skopt* library's `BayesSearchCV` function [57]. This iterative process aimed to identify optimal values for key hyperparameters such as tree depth and the number of estimators. Importantly, the Bayesian Search was only conducted when hyperparameter optimization was selected for a run. Otherwise, a standard Random Forest classifier was used, leveraging reasonable guesses for hyperparameters based on prior runs.

A key challenge encountered during this phase was the model's high rate of false negatives, where over 50% of the buggy files were often mislabeled as non-buggy. This significantly impacted the model's utility for fault prediction. Attempts to address this issue included oversampling techniques, which are explained later in the thesis. Despite these efforts, the results did not meet the performance criteria necessary to justify further exploration of this approach.

From a technical perspective, Random Forest's computational requirements, particularly for hyperparameter optimization, made GPU support a critical consideration. Early implementations relied on CPU-based calculations, which proved inefficient and time-consuming. To improve performance, RapidsAI's cuML libraries were explored for GPU-accelerated Random Forest training [58]. While this method was ultimately not adopted, the insights gained from setting up GPU acceleration informed subsequent implementations, particularly with CatBoost.

Ultimately, the Random Forest approach provided valuable insights and served as a foundation for early experiments. However, due to the combination of high false negatives and computational inefficiencies, it was deemed unsuitable for this project's goals. Consequently, the focus shifted to CatBoost, which is introduced in the next section.

4.4 CatBoost Classifier

The CatBoost implementation, being the main focus of this work's development, evolved significantly over time, as reflected in the various files within the project. This section first describes the CatBoost algorithm itself, followed by a focus on the implementation in `Catboost_Classifier.py`, which was used for the majority of the development and the experiments whose results are discussed later.

4.4.1 CatBoost Algorithm

CatBoost is a gradient-boosting algorithm specifically designed for handling categorical data effectively, making it a powerful choice for supervised machine learning tasks such as classification and regression. Gradient boosting itself is an ensemble technique that builds a sequence of decision trees, where each subsequent tree only makes very few assumptions about the data (therefore called 'weak') and corrects the errors of the previous ones. The final model is the weighted sum of all individual trees, optimized to minimize a specific loss function. For a better understanding of this process, it is visualized in Figure 4.3.

What sets CatBoost apart from other gradient-boosting methods like XGBoost or LightGBM are its unique features for managing categorical features, as well as its efficiency in avoiding overfitting [50]. Although CatBoost was initially designed for categorical features, there are many exemplary use cases where CatBoost also outperforms other machine learning algorithms on only numerical features, which is part of the reason why it was tried for these approaches in the first place [59, 60].

Key Characteristics

CatBoost stands out for its innovative approach to handling categorical data. Instead of requiring extensive preprocessing (e.g., one-hot encoding), CatBoost encodes categorical features internally using a technique called "order statistics." This encoding relies on the distribution of categories in the dataset, which helps preserve important information and improves training efficiency. Additionally, CatBoost implements symmetric tree structures, where each tree has the same depth and a fixed number of splits, leading to faster training times and more robust performance.

Another critical feature of CatBoost is its ability to avoid overfitting. It achieves this through techniques such as *ordered boosting*, where the model builds trees sequentially on different permutations of the training data, reducing target leakage and improving generalization [50].

How It Works

At a high level, CatBoost operates by iteratively building decision trees, with each tree aiming to reduce the residual error of the ensemble at that stage. For numerical features,

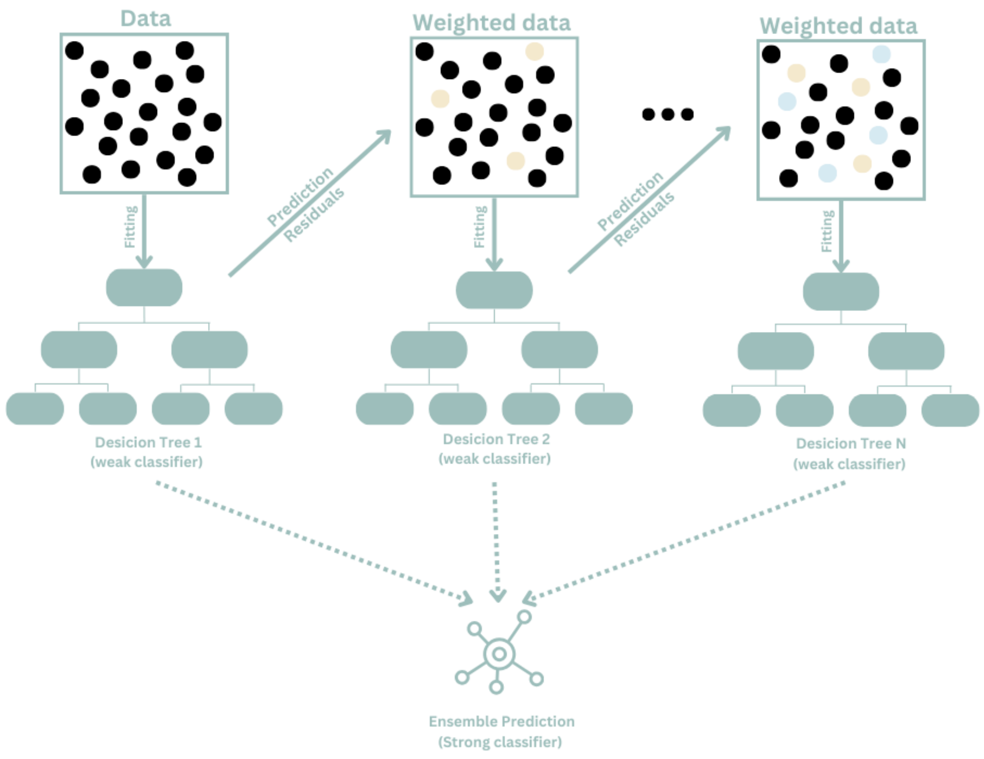
Data
Fitting
Prediction Residuals
Weighted data
Fitting
Prediction Residuals
Weighted data
Fitting
Desicion Tree 1 (weak classifier)
Desicion Tree 2 (weak classifier)
Desicion Tree N (weak classifier)
Ensemble Prediction (Strong classifier)

Figure 4.3: Visualized example of how Gradient Boosting works.

the algorithm performs splits based on thresholds, while for categorical features, it uses its internal encoding to determine splits. The model optimizes a custom loss function during training, which can be tailored to the specific task (e.g., log loss for classification).

The algorithm applies regularization techniques, including *L2 regularization* and *shrinkage*, to prevent overfitting. Additionally, it incorporates features such as *bagging* and *feature importance ranking*, further enhancing its robustness and interpretability.

Why CatBoost Was Chosen

CatBoost was selected as the primary algorithm for this work due to several key advantages:

1. **High Predictive Accuracy:** CatBoost consistently demonstrated superior accuracy in comparative tests during the development phase. Its ability to learn complex, non-linear patterns in the data contributed to its strong performance.
2. **Efficient Training on GPUs:** CatBoost supports GPU acceleration, significantly reducing training times – a critical advantage given the iterative nature of hyperparameter tuning in this work. Unlike many other non-neural network algorithms,

which often require additional setup or third-party libraries for GPU utilization, CatBoost offers integrated GPU support out of the box. This ease of use, combined with its efficiency, made it a compelling choice for this project.

By combining these features, CatBoost emerged as the best-performing algorithm in this thesis, achieving the highest accuracy and recall rates among all tested approaches. Further details on its performance and evaluation can be found in the Results chapter.

4.4.2 Implementation Details

The CatBoost classifier, utilized as the core algorithm in this work, was implemented using the `catboost` library [61]. One notable feature of CatBoost is its native GPU support, which significantly enhances training performance. Tests conducted during development indicated up to a 30x speed improvement when utilizing a GPU, due to its optimized architecture for machine learning tasks.

Key hyperparameters for the CatBoost classifier, such as `depth`, `iterations`, and `learning_rate`, were defined through a parameter grid. This grid was crucial for systematic hyperparameter tuning, as it allowed the identification of optimal configurations that maximized model performance. The parameter tuning process will be elaborated further in the next section.

After initialization, the classifier was trained on preprocessed data, followed by evaluation using separate test datasets. This process ensured reliable accuracy measurements while mitigating overfitting risks. These foundational steps, alongside additional measures such as undersampling techniques, contributed to the robust performance of the CatBoost model.

4.4.3 Optimization via Parameter Tuning

To automate the search for an optimal model, the entire program was wrapped in a loop, enabling the training of multiple models with different parameters in a single run. This in turn led to easier documentation and analysis of those parameters, which is a central part of machine learning. For each iteration of this loop, random parameters were generated for the above-mentioned parameter grid for the CatBoost model.

To better understand the structure of a parameter grid in this use case, attention is now directed to the following code:

```
parameter_grid = {
    'depth': [8, 9, 10],
    'iterations': [5000, 7500, 10000],
    'learning_rate': [0.005, 0.01, 0.05, 0.1, 0.2, 0.3],
    'l2_leaf_reg': [1, 3, 5, 7, 9],
    'border_count': [128, 256],
    'bagging_temperature': [0, 1]}
```

Normally, there is exactly one value for each of those variables, not an array of values. But in order to automate finding the optimal model with the optimal values for each of those parameters, a simple random parameter selection function was created, which takes such a grid with arrays as input and outputs randomly picked values from each array.

This approach enabled the creation of multiple models per run, each with different parameters, allowing for training and subsequent comparison. Without this approach, all of these tasks would have to be performed manually. By automating much of the repetitive work, it became much easier to document and analyze the results.

4.4.4 Undersampling

Revisiting the pitfalls of machine learning, one major issue that affected this implementation was the imbalance in the dataset. Specifically, the number of non-bugs was nearly ten times greater than the number of bugs, leading to complications during model training that required attention.

In earlier versions, an ADASYN-like approach was employed for oversampling the data [62]. However, this method did not yield the expected improvements and also slowed performance, prompting the adoption of its counterpart. Said counterpart, named Undersampling, works in the way that it randomly samples a select number of instances of the dominant (i.e. more frequent) class, to balance the data. Generally, this number is set to around 90% of the minority class, but it can vary a bit.

This solution was implemented by using the RandomUnderSampler from the *imblearn* library [63]. Since this allowed the model to train with only a subset of the actual data it improved performance (in Oversampling it uses a Superset, so increases the size and therefore training duration) and resulted in a higher Accuracy Score.

4.4.5 Full Pseudocode of the Application

With the individual components of the algorithm explained, a pseudocode representation of the complete CatBoost-based implementation is now presented.

Before that, however, one important thing to note is that the splitting of train and test data has also been put inside of a loop which normally was executed 5-10 times, dependent on prior time constraints. This was done to better be able to produce an average score for a model, since otherwise, a model could just have gotten lucky with its split, which would have led to an artificially high Accuracy Score.

As a side note, the Python file with the complete Implementation can be found in the associated Repository and is called `Catboost_Classifier.py`. In algorithm 4.1 below, the full pseudocode of it is listed.

Algorithm 4.1: Pseudocode for the Machine Learning Model Training with Random Parameter Selection and Undersampling

```
   Input  : Dataset $\mathbf{D}$, Parameter grid $\mathbf{P}$, Scalars $n_{\text{models}}$, $n_{\text{ttsplit}}$, $n_{\text{testUnder}}$
   Output : Best model with associated parameters and scores
   /* Data Loading and Preparation */
 1 Load dataset $\mathbf{D}$;
 2 Drop irrelevant columns from $\mathbf{D}$;
 3 Separate features $\mathbf{X}$ and target $\vec{y}$;
 4 Identify categorical and numerical features in $\mathbf{X}$;
 5 Define preprocessing transformer for numerical data;
   /* Model Training Loop with Random Parameter Selection */
 6 Initialize best model variables;
 7 for $k \leftarrow 1$ to $n_{models}$ do
 8     $\mathbf{p} \leftarrow$ random_parameter_selection($\mathbf{P}$);
 9     Initialize CatBoostClassifier with parameters $\mathbf{p}$;
10     for $i \leftarrow 1$ to $n_{ttsplit}$ do
11         $\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{test}}, \vec{y}_{\text{train}}, \vec{y}_{\text{test}} \leftarrow$ train_test_split($\mathbf{X}, \vec{y}$);
12         Scale training and test sets;
13         $\mathbf{X}_{\text{train}}^{\text{undersampled}}, \vec{y}_{\text{train}}^{\text{undersampled}} \leftarrow$ fit_resample($RandomUnderSampler$,
            $\mathbf{X}_{train}, \vec{y}_{train}$);
14         Train model on undersampled data: fit($\mathbf{X}_{train}^{undersampled}, \vec{y}_{train}^{undersampled}$);
15         Initialize test accuracies list;
16         for $j \leftarrow 1$ to $n_{testUnder}$ do
17             $\mathbf{X}_{\text{test}}^{\text{undersampled}}, \vec{y}_{\text{test}}^{\text{undersampled}} \leftarrow$ fit_resample($RandomUnderSampler$,
                $\mathbf{X}_{test}, \vec{y}_{test}$);
18             $\vec{y}_{\text{pred}} \leftarrow$ predict($\mathbf{X}_{test}^{undersampled}$);
19             Compute accuracy and update best model if necessary;
20             Append accuracy to test accuracies list;
21         end
22         Calculate and store average accuracy for this train/test split;
23     end
24     Calculate model's overall average accuracy and update best model;
25 end
26 Generate and save confusion matrix;
27 Save best model and results to files;
```

4.5 Risk Score

While the primary objective of this work was the classification of files as buggy or non-buggy, an additional goal was to provide a more nuanced perspective on the likelihood of bugs, better understanding, and visualization opportunities. This motivated the development of a Risk Score, which quantifies the risk associated with a file rather than simply assigning a binary label.

The Risk Score assigns each file a value between 0 and 100, where 0 indicates a file is highly unlikely to contain bugs, and 100 suggests a high likelihood of a bug. These values can also be interpreted as percentages, offering an intuitive representation of the model's confidence in the presence of bugs.

4.5.1 Calculation of the Risk Score

The Risk Score is calculated using the same CatBoost model as the classifier but with slight modifications. Instead of producing binary classifications, the model outputs raw probabilities that a file is buggy. These probabilities, ranging from 0 to 1, are scaled to a range of 0 to 100 to produce the Risk Score. Mathematically, the Risk Score for a given file can be expressed as:

$$\text{Risk Score} = \text{Probability(Bug)} \times 100$$

where *Probability(Bug)* represents the probability predicted by the model that a file contains a bug.

The process involves:

1. Loading the pre-trained CatBoost model.
2. Generating probability predictions for the test dataset.
3. Mapping the probabilities to the [0, 100] range.
4. Appending the Risk Score to the dataset as a new column for further analysis.

4.5.2 Interpretation of the Risk Score

The Risk Score provides a ranking of files based on their likelihood of containing bugs. Files with higher scores are prioritized for further review, while those with lower scores can be addressed later. This numerical representation allows developers to focus their debugging efforts on high-risk areas, optimizing time and resources.

It is important to note that the Risk Score is influenced by the binary nature of the training data, where files are labeled as either buggy or non-buggy. While the score provides valuable insights, it should not be interpreted as an absolute measure of risk but rather as a relative ranking within the dataset.

4.5.3 Testing and Evaluation of the Risk Score

The Risk Score was tested using the same methodology employed for the classifier. Specifically:

- Accuracy was calculated by thresholding the Risk Score (e.g., scores above 50 were considered buggy) and comparing the results to the ground truth labels.
- Confusion matrices were generated to evaluate true positives, false positives, true negatives, and false negatives.
- Precision and recall metrics were computed to assess the balance between correctly identified buggy files and false positives.

One thing that should be noted, however, is that the labels of the training data for the Risk Score model are binary, a file either contains a bug or not. Therefore the calculation of a Risk Score by the model with those prerequisites should be taken with a grain of salt. It should be seen as a side-arm of the project without a claim to complete precision and correctness. The Risk Score is intended to occasionally provide valuable insights into which files, identified as buggy by the model, carry a higher level of risk than others.

4.5.4 Visualization of Risk Scores

The Risk Score results were utilized to create several visualizations to facilitate further analysis and understanding of the bug prediction patterns across the dataset. These visualizations included:

- A heatmap showcasing the temporal evolution of Risk Scores, enabling the identification of patterns and correlations between risk spikes and bug occurrences.
- A treemap that provided a hierarchical view of Risk Scores distributed across files and folders, helping to pinpoint high-risk areas within the project structure.
- A line chart tracking the Risk Scores over time, offering insights into fluctuations and trends.

These visualizations can help in highlighting patterns within the dataset, identifying high-risk files, and providing actionable insights for prioritizing debugging efforts. The corresponding figures are presented and discussed in detail in the 'Discussion' chapter, where their implications are further analyzed.

4.6 Regression-based Approach

To evaluate the performance of the CatBoost model, it was compared against a regression-based solution. Naturally, a regression-based approach is expected to perform worse, as it is inherently ill-suited for binary classification tasks, which involve discrete labels rather than continuous values. Regression models struggle to capture the discrete nature of classification problems, resulting in reduced accuracy and effectiveness.

However, comparing the regression-based approach to the classification model can still provide valuable insights. It serves as a baseline for evaluating the classification model's quality, highlighting the advantages of using a method specifically tailored to the problem and emphasizing the importance of capturing non-linear relationships. As a result, research question 4 emerged, leading to the development of a solution based on regression.

In the following subsections, the two regression-based algorithms explored in this work, ElasticNet and Logistic Regression, are introduced and explained. These explanations cover their key characteristics, mechanisms, and motivations for selection. Once these conceptual foundations are laid out, the specific implementation details are presented, discussing how these algorithms were utilized in practice.

4.6.1 ElasticNet

ElasticNet is a linear regression algorithm that combines the penalties of Lasso (L1 regularization) and Ridge (L2 regularization). This combination allows ElasticNet to effectively handle multicollinearity (highly correlated features) and perform automatic feature selection by shrinking less important coefficients toward zero. This dual capability makes it particularly suited for datasets with a large number of correlated or noisy features and more complex relationships between the features compared to both Lasso and Ridge [64].

How It Works

ElasticNet combines the strengths of two well-known techniques: Lasso regression, which emphasizes simplicity by selecting only the most important features, and Ridge regression, which stabilizes the model by handling correlated features. By blending these two approaches, ElasticNet strikes a balance between reducing model complexity and maintaining robustness. Its behavior is controlled by two key hyperparameters, λ_1 and λ_2, which determine the relative emphasis on L1 (Lasso) and L2 (Ridge) penalties, respectively. The ElasticNet loss function can be expressed as:

$$\mathcal{L}(\beta) = ||y - X\beta||_2^2 + \lambda_1||\beta||_1 + \lambda_2||\beta||_2^2$$

Here, β represents the feature coefficients, y is the target variable, and X is the feature matrix. The L1 penalty encourages sparsity by reducing less significant feature coefficients to zero, while the L2 penalty helps manage multicollinearity among features. By tuning these parameters, ElasticNet adapts to prioritize sparsity, robustness, or a balance between the two, depending on the dataset and the problem at hand. This flexibility allows ElasticNet to uncover meaningful relationships, even in cases where features are interdependent or noisy [64].

Why ElasticNet Was Chosen

ElasticNet was selected as the first regression-based approach due to its conceptual strengths in managing collinearity and its feature selection capabilities. Given the diversity and interdependence of the metrics in the dataset, ElasticNet was a logical choice to explore relationships between features and outputs, providing insights into the data's underlying structure. Further, compared to standard linear regression, Lasso and Ridge, it is the most sophisticated approach for addressing complex problems that are still a linear regression, meaning, it fits coefficients to the individual features.

Challenges and Limitations

While ElasticNet proved effective in identifying relationships and handling feature selection, it was inherently ill-suited for the binary classification problem of bug prediction. As a regression algorithm, ElasticNet is designed to predict continuous values rather than discrete labels, making its application to classification tasks challenging. This limitation prompted the transition to Logistic Regression, which, while still a regression algorithm, is better suited for binary classification tasks.

4.6.2 Logistic Regression

Logistic Regression is one of the simplest and most widely used algorithms for binary classification tasks. Despite its name, it is fundamentally a classification algorithm rather than a regression method. Logistic Regression models the probability that a given input belongs to a particular class by fitting a sigmoid (logistic) function to the data.

Key Characteristics

The core feature of Logistic Regression is its use of a linear model combined with a logistic function. This function maps the output of the linear model (a real-valued number) to a probability between 0 and 1. Predictions are then made by applying a threshold – commonly 0.5 – to this probability, assigning the input to one of two classes.

Logistic Regression is computationally efficient and interpretable, with coefficients that provide insights into the relative importance of each feature, similar to a linear model. It performs well on linearly separable data but struggles with complex, non-linear relationships unless combined with feature engineering or extensions like polynomial regression [65].

How It Works

The algorithm optimizes a cost function called the *log-loss* (logarithmic loss), which penalizes incorrect predictions. During training, the coefficients of the model are adjusted iteratively using techniques such as gradient descent to minimize this loss. The result is a set of weights for each feature that determines its contribution to the classification decision [65].

Why Logistic Regression Was Chosen

Logistic Regression was chosen for this project as a baseline model for comparing classifier- and regression-based approaches. It offered several advantages:

1. **Simplicity:** Logistic Regression provided a straightforward implementation to test the viability of a classification-based approach for bug prediction.
2. **Interpretability:** The coefficients of the logistic model allowed for insights into how individual metrics influenced predictions.
3. **Transition from ElasticNet:** After ElasticNet demonstrated limited success in this context, Logistic Regression was introduced as a more classification-oriented alternative. However, the penalty terms from ElasticNet were incorporated into the Logistic Regression approach, utilizing the L1 and L2 terms to address feature correlation and enhance the model's ability to capture complex relationships among individual features.

Looking ahead to the results, Logistic Regression, while still outperformed by CatBoost, delivered reasonably adequate outcomes for the conducted experiments. More importantly, it served as a straightforward baseline – a standard interpretable linear classifier optimized through its coefficients. This made it both explainable in the classical sense and indicative of linear relationships between the studied features.

Comparison to ElasticNet

The primary difference between Logistic Regression and ElasticNet lies in their objectives: Logistic Regression is specifically designed for classification tasks, while ElasticNet is a regression method that combines Lasso and Ridge penalties to address collinearity and perform feature selection. While both approaches may incorporate penalty terms, the key distinction is that Logistic Regression includes an "activation function" (sigmoid function) following the linear regression step, which makes it particularly effective for classification tasks. Mathematically, while ElasticNet produces a linear combination of features with coefficients and penalty terms, Logistic Regression applies the sigmoid function to its linear output, enhancing its suitability and performance for classification problems.

ElasticNet was initially chosen to explore regression-based solutions, but its conceptual focus on predicting continuous values made it less suited for the classification-oriented nature of the problem. This prompted the shift to Logistic Regression as a more appropriate approach for binary classification.

4.6.3 Implementation Details

The regression-based approach initially utilized the ElasticNet algorithm from the *scikit-learn* Python library [52], which combines Lasso and Ridge penalties for regression.

However, test runs revealed that this method was unsuitable for the task, with accuracy scores only slightly above 50%, comparable to random guessing. As a result, the focus shifted to Logistic Regression, which, while still a regression method, incorporates classification capabilities by applying a sigmoid function to the output, allowing for binary decision-making.

Both ElasticNet and Logistic Regression were implemented within the same file, `LinReg_Classifier.py`, with a boolean variable (*lr*) used to switch between the two methods during runtime. This allowed for flexibility in testing and comparing their performances. Key hyperparameters for both algorithms, such as `l1_ratio` and `alpha` for ElasticNet, and `C` (regularization strength) for Logistic Regression, were tuned to optimize results.

Predictions for both approaches were made on undersampled test data, consistent with the methodology employed throughout this work. To classify predictions, a threshold of 0.5 was applied to the continuous output of the regression models, mapping results to binary values (0 for non-bug, 1 for bug). The remainder of the evaluation process, including accuracy calculation and result saving, followed standard procedures.

Although this approach offered valuable insights, particularly as a baseline comparison for the classification-based CatBoost model, it struggled to handle the dataset's complexity and non-linear relationships. Detailed results and performance comparisons are provided in chapter 5.

4.7 Testing and Evaluation

An integral part of the development process in this work was the systematic testing and evaluation of the machine learning models. This section provides an overview of how the performance of the models was assessed, as well as the techniques employed to identify the best-performing configurations.

4.7.1 Metrics for Evaluation

The primary metric used to evaluate the models was the accuracy score, which indicates the proportion of correctly classified data points. However, accuracy alone is insufficient to provide a comprehensive understanding of model performance, especially for imbalanced datasets. Therefore, confusion matrices were also generated for each run to analyze true positives, false positives, true negatives, and false negatives. From these, additional metrics such as precision, recall, and the F1-score were calculated to better understand the trade-off between identifying bugs (recall) and avoiding false alarms (precision).

These metrics played a crucial role in determining which parameter configurations warranted further tuning. For example, high recall values for buggy files ensured the model captured as many bugs as possible, while precision indicated its ability to minimize false positives. Together, these metrics provided a balanced view of the model's performance.

4.7.2 Train-Test Splitting

To ensure robustness and shield accuracy calculations from outliers, a train-test splitting approach was employed during model training. This involved dividing the dataset into separate training and testing subsets multiple times. For most models, 3 to 5 splits were used, enabling the evaluation of each model's performance across multiple iterations. The results were averaged to obtain a more reliable accuracy score, reducing the impact of any single, potentially anomalous split. This splitting methodology helped validate the generalizability of the models, ensuring that their performance metrics were not overly influenced by specific partitions of the data.

4.7.3 Documentation of Results

Each model run was meticulously documented in two Excel files: `Param_Comparisons` and `Runtime_Comparisons`. These files recorded not only the accuracy scores but also additional details such as runtime, the configuration of hyperparameters, and the number of iterations. Furthermore, summaries of the results, containing accuracies, confusion matrices, and run details, as well as all the chosen parameters for each iteration of a model per run were saved in `.txt` files. This process was done for over 100 models. By systematically logging this data, the evaluation process was streamlined, and trends across multiple runs became evident.

4.7.4 Parameter Tuning

The parameter tuning process, described above already, was automated, enabling multiple configurations to be tested within a single run. The results of each parameter iteration were compared, and averages and maximum values of accuracy were calculated to reduce the impact of outliers. This allowed for a more robust evaluation of which parameter settings performed best over time.

To better understand the impact of specific parameters on accuracy, correlations between hyperparameters (e.g., `depth`, `iterations`, `learning_rate`) and performance metrics were calculated. This analysis provided insights into how individual parameters influenced the model's effectiveness.

4.7.5 Insights from Runtime Comparisons

In addition to performance metrics, runtime data was carefully analyzed to balance model accuracy with computational efficiency. For example, the trade-off between increasing the number of iterations and the resulting runtime was a critical consideration. Early iterations of the model used fewer iterations and smaller depths, resulting in shorter runtimes but lower accuracy.

The systematic testing and evaluation process underscored the importance of balancing accuracy with practical considerations like runtime and resource usage. The use of train-test splitting, confusion matrices, correlation analysis, and detailed documentation allowed for an iterative improvement of the model and ensured that each decision was data-driven.

4.8 Helper Classes & Miscellaneous

Several helper classes and files were implemented throughout the project to support various tasks. While not central to the core methodology, these components played an important role in facilitating different aspects of the workflow. This section highlights some of these helpers and examines parts of their code to illustrate their impact on the project and provide a more comprehensive understanding of the workflow and effort involved in obtaining the results, beyond the primary approaches.

4.8.1 Model Testing

The models were tested immediately after training within their respective files, as described earlier, to ensure results were available immediately after a run, which sometimes lasted over 10 hours. However, it was equally important to have the ability to test a given model independently. Decoupling the training and testing processes proved beneficial for two main reasons: first, training is a time-intensive process, as previously mentioned and further discussed in the Results chapter, making it practical to separate the two stages. Second, the ability to test a model trained on dataset A using a different dataset B added significant flexibility and value. As a result, the creation of a model testing file was a logical step. Multiple testing files were developed for the different approaches explored. The code snippet below focuses on the primary file for testing CatBoost classifier models, `CB_Classifier_ModelTest.py`:

```python
while i < n_ttsplit:
    i+=1
    X_train, X_test, y_train, y_test =
        train_test_split(X, y, test_size=0.2, random_state=(i*42)**2)
    rus = RandomUnderSampler(sampling_strategy=0.9)
    j=0
    test_accuracies =[]
    while j < n_testUnder:
        j+=1
        X_test_undersampled, y_test_undersampled =
            rus.fit_resample(X_test, y_test)
        y_pred = model.predict(X_test_undersampled)
        accuracy = accuracy_score(y_test_undersampled, y_pred)
        classification_report_str =
            classification_report(y_test_undersampled, y_pred)
        test_accuracies.append(accuracy)

    average_accuracy_tt = np.mean(test_accuracies)
    tt_accuracies.append(average_accuracy_tt)
    conf_matrix = confusion_matrix(y_test_undersampled, y_pred)
avg_accuracy = np.mean(tt_accuracies)
```

In short, the model was loaded and evaluated on different randomly selected test data to obtain an average score. Often this score ended up being a bit higher than prior ones for this model since it was very likely that some of the trained data now was in the test set.

An important addition to this file, absent in the original training file, was the extraction of feature importance. Using this data, combined with the matplotlib library, a simple bar chart was generated to easily analyze which metrics had the greatest impact on determining whether a file contained a bug.

4.8.2 Loading and Retraining of Models

For a similar reason as mentioned earlier regarding the use of the model testing file, the file `LoadAndRetrain_Model.py` has proven to be very useful, particularly for retraining the models with new data. Instead of having to start training from scratch again when an update to the dataset was released, the model could simply be iteratively improved.

Another advantage was the option to change the parameter grid and test that with an already trained model, to perhaps quickly see whether those changes would lead to improvement in accuracy or could gladly be discarded.

4.8.3 Timed Runs

One challenge encountered during the numerous runs conducted to find the best model was the difficulty in accurately estimating the duration required for each run. Although it was rather obvious after a few tries that the combination of a high depth and a high number of iterations for the model during its training phase would lead to an increase in execution time, it was still hard to pinpoint the runtime exactly. Also, since the parameters were drawn randomly for each model, it could happen that one run would have an overrepresentation of models using such time-consuming parameters.

To address this issue, the file `Timed_CB_Classifier.py` was created, where the number of models to be trained (from which the best is selected) was replaced with a runtime specification. After training each model, the code checks whether the specified runtime has been exceeded. If so, it skips the training of additional models and proceeds directly to saving the results.

Apart from this small change in the while loop's decision criteria, the rest of the code remains unchanged, so it is not shown here.

4.8.4 CatBoost Regression

Similarly to the ElasticNet and Logistic Regression implementations discussed in section 4.6, a CatBoost Regressor solution was created for the sake of comparison and curiosity. Its differences from the classifier file mainly lie in the model initialization and the addition of a filter to map the predictions to 0 or 1 in order to be able to calculate the accuracy, just like before. The Code for it can be found in the file `CB_Regressor.py`. There is not much difference between the ElasticNet and Logistic Regression implementations, naturally only that the CatBoostRegressor is used in their stead.

4.9 Obstacles Encountered

Throughout the development of the solution described above, numerous challenges and obstacles were encountered that required careful consideration. Some had to be circumvented, others avoided, and a few addressed with entirely alternative methods. These challenges significantly influenced the final solution, even though they have not been extensively discussed thus far. This section highlights the key issues that shaped the work and the problems faced during the process.

4.9.1 Environment Setup

One of the biggest time sinks in this project, with minimal impact on the final product, was setting up the environment. In particular, the installation of the necessary Python packages proved to be immensely time-consuming, which can be quite often the case with Python. A notable challenge arises when two necessary dependencies conflict with each other, making it difficult to resolve compatibility issues. For example, Dependency A requires Python 3.8 or lower, while Dependency B requires Python 3.9 or higher. A similar issue occurred in this project, particularly with RapidsAI requiring newer versions while other dependencies needed older ones.

In general, the switch from a CPU- to a GPU-based approach led to many hours invested just to try to make it work. Installing Nvidia's CUDA Toolkit, which was needed to utilize GPU acceleration [66], proved to be quite extensive and time-consuming. A problem that played a big part in that was that Anaconda, an environment/package solver for Python [67], took very long for some packages to install, and often failed after considerable time. To address this, the libmamba solver was used. Although it initially posed challenges during installation, it significantly sped up the dependency installation process afterward. One reason for this problem could be that the solution was developed inside Ubuntu on the WSL (Windows Subsystem for Linux) and that an implementation on a native Linux system could have helped avoid some of those challenges, as well as saving time and energy.

Note: The linked guide was used for installing the necessary tools for GPU development, should any reader wish to run the code on their machine [68].

4.9.2 Time of runs & Processes killed

As briefly mentioned earlier about the timed runs, predicting their duration proved challenging due to the randomness of the parameter selection. This has already been explained, so it will not be reiterated. However, one aspect that has not been addressed is that, presumably due to the long execution times, some processes were occasionally terminated by the host machine. As one might imagine, this could be incredibly frustrating, especially after a run had already taken several hours, only to be abruptly aborted without any results. This could again point to a potential issue with not using a native Linux system. However, both of these issues have occurred to many programmers on various projects using Linux as well, so it is likely only a part of the reason, if at all. A more detailed understanding of WSL and its workings would be needed for a fair assessment.

4.9.3 Documentation & Parameter Pre-Selection

Another time-consuming aspect of the project was documenting the results. This is common in machine learning projects, so it is nothing unusual. However, it is worth mentioning because a significant amount of time was dedicated to documentation and analysis behind the scenes.

The main part of that work was focused on documenting various details about the run, like run time or parameters used, for easier analysis after the fact on the one hand, and for improving parameter pre-selection for the next runs on the other. Since runs could take quite some time, it was essential to at least be able to limit the parameter space to only generate favorable pairings and not spend unnecessary time trying out known bad parameters. As can be seen in the Documentation files, this was not always an easy task since some datasets reacted differently than others, and also sometimes there was no clear correlation between singular parameters and the resulting accuracy. However, without the extensive time invested in documentation and analysis, even more time would have been required to optimize the model and identify the best-performing parameters.

4.9.4 Dominant Metrics

The last of the *Obstacles encountered* luckily only appeared a handful of times but still proved challenging. Namely, the problem was when one metric (feature) was too dominant within the model. This, for example, happened once with the metric *totalBugfix*. With it, the model achieved an accuracy score of almost 90%, but when analyzing the Feature Importance function from CatBoost, it revealed that *totalBugfix* had a score of 78, while the next highest had a score of only 3. This meant that one metric alone had almost 80% impact on the decision-making of the model. Since this was too dominant, the metric was removed from further tests. This led to a drop in accuracy of about 10% at first. After a bit of training and new data being added to the dataset this was mitigated a bit, but the nearly 90% accuracy could not be reached again. Fortunately, this only occurred once.

With this, chapter 4 concludes, outlining the methodology and key steps taken to address the problem at hand. The following sections focus on the results achieved using these methods, examining the outcomes, analyzing their implications, and evaluating how effectively the objectives have been addressed. This transition marks a shift from the 'how' of the approach to the 'what' – the tangible results and insights produced by these methods.

CHAPTER 5

Results

This section presents the key findings from the experiments and evaluations, highlighting the performance of the proposed machine learning model and the integration of code metrics. The results address the research questions outlined in this thesis, focusing on the accuracy of the developed models (RQ1), the suitability and reliability of various code metrics in machine learning models (RQ2), and the comparative performance of supervised and unsupervised approaches (RQ3). Furthermore, the analysis examines how a classifier-based approach compares to a linear regression-based one (RQ4).

The evaluation emphasizes the strengths and limitations of the models, providing insights into the trade-offs between different methodologies and metrics. These findings offer practical guidance for fault prediction in real-world software projects. In addition to validating the proposed classifier, the results contribute to a broader understanding of fault prediction techniques within the field of software engineering.

5.1 RQ1 – How accurate is the developed Machine Learning model for Fault prediction?

The core focus of this work is the development of a machine learning algorithm to predict bugs in code with the highest possible accuracy. Over 100 models were trained and evaluated using systematic hyperparameter tuning and optimization. This process included testing different algorithms, such as Random Forest and Logistic Regression, before ultimately selecting CatBoost due to its superior performance.

The best-performing model achieved an accuracy score of 84.1%, demonstrating the effectiveness of the approach in predicting bug-prone files.

To evaluate the performance of different approaches, Table 5.1 highlights the accuracy achieved by Logistic Regression, Random Forest, and CatBoost classifier on the final dataset. While Random Forest and Logistic Regression provided reasonable baselines, their accuracy scores of **74.3%** (*on a prior dataset) and **61.4%** respectively were significantly lower than the **84.1%** achieved by CatBoost. This demonstrates the superiority of gradient-boosting techniques, particularly CatBoost, for fault prediction tasks. CatBoost's ability to handle categorical features and its efficient use of hyperparameter optimization were key factors in its success.

Table 5.1: Comparison of Model Approaches

Model	**Accuracy (%)**
Logistic Regression	61.4
Random Forest*	74.3
CatBoost Classifier	84.1

Once CatBoost was identified as the most effective algorithm, iterative improvements were made through systematic hyperparameter tuning. Table 5.2 summarizes the performance of several selected CatBoost models trained on the final dataset, showing how adjustments to parameters like `iterations` and `depth` contributed to increased accuracy.

Table 5.2: Comparison of Model Performance of selected CatBoost Models

Model Run	**Accuracy (%)**	**Iterations**	**Depth**
fds_pt_run_17	84.10	30,000	11
fds_pt_run_13	83.78	20,000	11
fds_pt_run_6	83.67	10,000	12
class_4	83.42	10,000	10
run_class_3	80.94	10,000	7

As shown in Table 5.2, the final CatBoost configuration (`fds_pt_run_17`) achieved the highest accuracy of **84.1%** with 30,000 iterations and a depth of 11, while still maintaining reasonable runtime, of about one hour. Earlier models, such as `fds_pt_run_6` and `class_4`, also performed well but achieved slightly lower accuracy scores. This progression illustrates how the iterative tuning of hyperparameters such as `depth` and `iterations` contributed to improved performance.

In addition to the classification-based approach, a Risk Score model was developed to provide a numerical representation of bug likelihood. This was achieved by adapting the CatBoost model into a regression-based approach, allowing the assignment of a probability value between 0 and 1 for each file at a specific commit. The Risk Score model enables finer granularity in assessing risk, offering insights beyond binary classification. Despite these changes, the Risk Score model maintained strong predictive performance, achieving an impressive accuracy of **82.2%**, comparable to the classifier approach.

The training of the best-performing classifier model took approximately one hour, which in the ML context is not all too much. But this fact should be taken with a grain of salt since over a hundred hours of training had been invested before finding promising hyperparameters, with which this model then was initialized.

Although highly technical, a brief discussion of the hyperparameters of the CatBoost algorithm is warranted, as a significant portion of the training effort was dedicated to their optimization. The model parameters were the following: `depth` 11, `iterations` 30.000, `l2_leaf_reg` 7, `border_count` 256, and `bagging_temperature` 0. What exactly they mean will be discussed in chapter 6, 'Discussion'.

These results indicate that, on average, across all commits from the 34 open-source projects in the dataset, the model achieves an 84% accuracy in correctly predicting whether a bug is present, evaluated on a per-file, per-commit basis.

Furthermore, its precision and recall scores for bugs were 84% and 83% respectively, while on Non-bugs 85% and 86%, implying that neither of the two categories skewed the results. This could happen if it were to mislabel too many data points as bugs for example, which would lead to a high recall Score for bugs, but low precision, and vice versa for Non-bugs. Further details of these results, how they were achieved, the Risk Score, and the implications will be discussed in the next chapter.

A notable limitation of these results is the reliance on bugfixes as a proxy for actual bugs in the target values of the model.

5.2 RQ2 – Which metrics are best suited for Fault prediction use in Machine Learning models and how reliable are they?

To address this question, the feature importance values generated by the model were examined. These values indicate the extent to which each input feature contributes to the algorithm's decisions. Specifically, a higher feature importance value signifies that the classifier assigns greater weight to that feature – representing, in this case, a specific metric – compared to features with lower importance values. This analysis reveals which features exert the most influence on the classifier's decision-making process.

As shown in Figure 5.1, the top three metrics in the dataset for bug prediction are ranked as follows: *hcpf1*, *projectBus1*, and *hcpf2*. The *hcpf* metrics represent code change entropy, as proposed by Hassan [24], while *projectBus1* is derived from the SAGE project and corresponds to its implementation of the Bus factor metric.

Importantly, no single metric dominates the feature set. For example, *hcpf1* achieves a score of 15.2, *projectBus1* scores 6.8, and *hcpf2* scores 5.0, with many other metrics ranging between scores of 1 and 5. This indicates that the model does not rely exclusively on a single metric, which helps mitigate potential risks to its accuracy when applied to future projects.

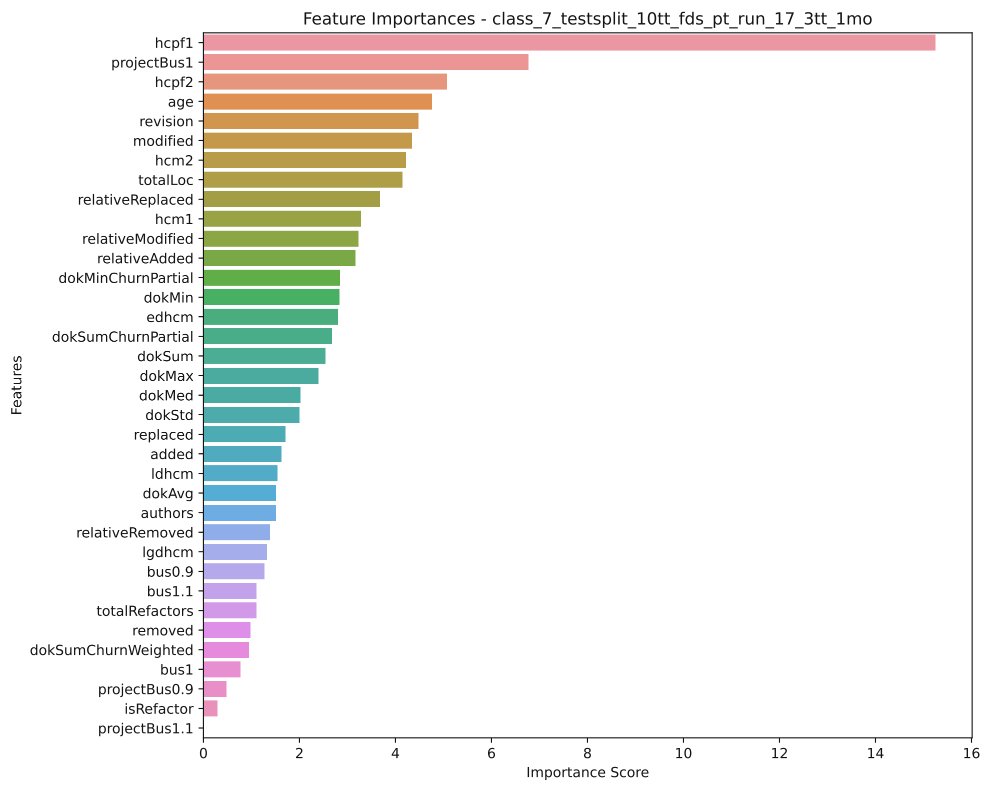

Figure 5.1: The Feature Importance Chart of the best Model developed.

Further details about the top metrics and their contributions will be discussed in the next chapter, including a notable exception: the *totalBugfix* metric, which is absent from Figure 5.1 and the final dataset.

5.3 RQ3 – How does a newly developed Supervised ML model perform in comparison to an Unsupervised one?

As mentioned earlier in section 3.2 when discussing the difference between supervised and unsupervised machine learning, the research revealed that an unsupervised approach would be entirely unsuitable for this task. Consequently, the topic of this research question was abandoned at an early stage.

5.4 RQ4 – How does a Classifier approach fare versus a Linear Regression approach?

While the research indicated that the task was best suited for a classifier-based approach, some time was also dedicated to exploring a solution using linear regression. This exploration was motivated by curiosity and the desire to establish a baseline for comparison with the classifier approach. However, given the anticipated limitations of linear regression for this task, significantly less effort was allocated to developing this model compared to the classifier. This disparity in effort should be taken into account when interpreting the results.

For the linear regression approach, two algorithms were implemented: an ElasticNet model and a Logistic Regression model with an ElasticNet penalty. The accuracy scores for these models were approximately **58% and 61%**, respectively, with the Logistic Regression model achieving the higher score.

Comparing these results to the 84.1% accuracy achieved with the classifier-based approach clearly demonstrates that the initial research was correct in identifying linear regression as ill-suited for the problem at hand.

It is important to note that while one might assume the less accurate solution is only 20% worse than the best model, this interpretation can be misleading. Considering that a simple random guess or coin flip would yield an average accuracy of 50%, the performance of these less accurate solutions is significantly inadequate by comparison.

Another unfortunate fact for the regression approaches is the Recall values for bugs, which are only 31% for ElasticNet and 54% for Logistic Regression. This means that when deciding on commits that contained bugs, the former misjudges them more than half of the time as non-buggy, while for the latter it is almost evenly split. This is very bad and an algorithm like that would be of no practical use and in sharp contrast to the high observed values for precision and recall for the classifier model.

In summary, the findings of this thesis confirm the initial assumptions formed during the Review and Preparation phase: a classifier-based approach significantly outperforms a linear regression-based one for the task at hand. This result underscores the validity of the chosen methodology and highlights the superiority of classification models in this context.

CHAPTER 6
Discussion

This chapter provides a detailed analysis of the findings presented in the Results section. The primary objective is to interpret these outcomes in relation to the research questions, examining their implications, limitations, and relevance. The discussion focuses on the accuracy and reliability of the developed machine learning models for fault prediction, identifies the most effective metrics for these tasks, and evaluates the performance differences between classifier- and regression-based approaches. Through this analysis, the strengths of the proposed solution are highlighted, its limitations are addressed, and potential areas for future improvement are outlined.

6.1 Classifier Results

In the previous chapter, the primary result of this thesis was presented: the development of a classifier model achieving an accuracy of 84.1%. While that section detailed what was accomplished, this chapter revisits those results, delves deeper into the process that led to them, explores their implications, and provides additional insights into the model's characteristics.

As already mentioned, the best model achieved an accuracy score of 84.1%. Table 6.1 presents the classification report generated during its run. The precision and recall rates of 84% and 83% for bugs (1), respectively, highlight the model's effectiveness in identifying and predicting buggy files.

	precision	recall	f1-score
0	0.84	0.85	0.85
1	0.84	0.83	0.83

Table 6.1: Classification Report taken from the best Model.

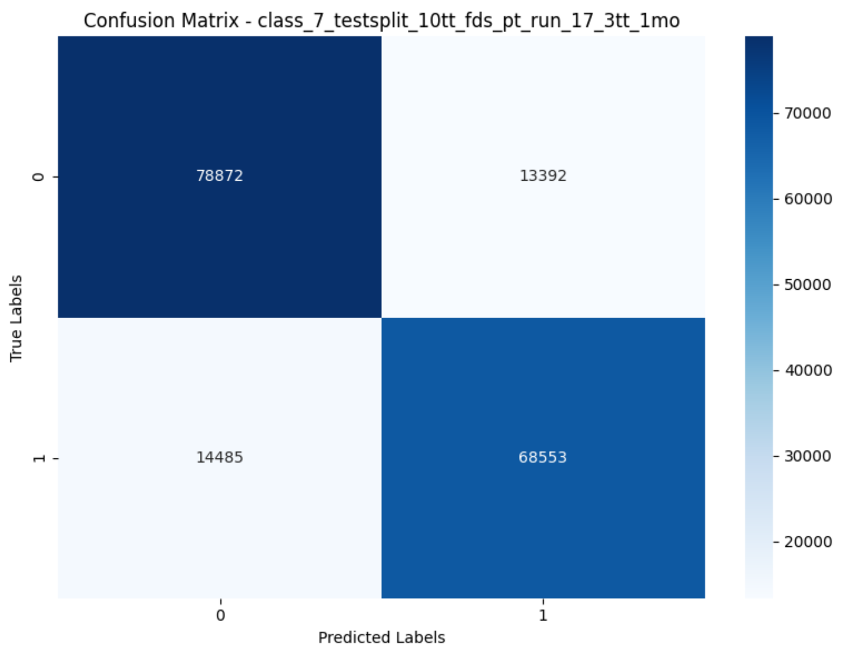

Figure 6.1: Confusion Matrix from the testing of the best Model.

The confusion matrix in Figure 6.1 illustrates the model's prediction performance. The X-axis represents predicted labels, and the Y-axis represents actual labels. Correctly classified non-bugs (true negatives) appear in the top-left quadrant, while correctly classified bugs (true positives) are shown in the bottom-right quadrant. The goal was to maximize these values while minimizing false positives and false negatives. The high numbers (78,872 and 68,553) in the 'true' quadrants, and significantly lower numbers (14,485 and 13,392) in the other two, underline the accuracy of the model, emphasized as well by the shades of blue for the quadrants.

6.1.1 Model Evolution and Hyperparameters

As detailed in chapter 4, the theoretical foundation and code implementation underlying the results have already been explained. This section shifts the focus to the model's evolution, specifically the hyperparameters used. Over several months of training, these parameters were iteratively refined and tuned to identify optimal configurations. For further details, the progression of these efforts is documented in two Excel files available in the accompanying repository: `Param Comparisons.xlsx` and `Runtime Comparisons.xlsx`.

Table 6.2 provides a sample view of the `Param_Comparisons.xlsx` file, highlighting how parameters like `depth`, `iterations`, `border count`, and `bagging temperature` were documented during the search for the best model.

Table 6.2: Sample Model Run Information

No.	Date	Runime	Accuracy	Depth	Iterations	l2	bc	bt
1	12.12.2024	00:43:23	83.86%	11	20,000	7	256	0
2	12.12.2024	00:40:50	82.92%	11	20,000	5	128	1
3	12.12.2024	00:40:40	82.82%	11	20,000	9	128	1
4	12.12.2024	00:42:49	83.85%	11	20,000	9	256	0
5	12.12.2024	00:42:33	84.01%	11	20,000	5	256	0
6	12.12.2024	00:43:44	83.40%	11	20,000	5	256	1
7	13.12.2024	01:05:49	84.10%	11	30,000	7	256	0

Throughout the model training process, several hyperparameters were adjusted and refined. Among the most critical were the number of iterations the algorithm performed and the depth of the solution trees it analyzed, referred to simply as `iterations` and `depth`. Early in the process, it became evident that higher values for these parameters correlated positively with improved accuracy. However, optimizing the model was not as straightforward as maximizing these values, as other hyperparameters also influenced performance significantly.

In the beginning, no one-size-fits-all solution was found and there was no clear 'the more the better' correlation between any of those variables and the accuracy score of the resulting model. This fact made it difficult to optimize the pre-selection of those values (lower and upper bounds of the arrays) and led to many hours spent in analysis trying to fine-tune said process. However, as the hyperparameters and the model evolved, the dataset also grew significantly. Initially, a smaller dataset was used, but by the end of the process, the input file had expanded to a substantial size of 750 MB. This growth introduced practical challenges, such as difficulties opening the file in tools like Microsoft Excel. Additionally, the increasing complexity and size of the dataset rendered earlier model configurations inadequate. As a result, the hyperparameters were revisited and refined, with increases in `iterations` and `depth`, which consistently led to incremental improvements in accuracy. This can be seen in Table 6.3, which documents the correlations of parameters with the accuracy score in the final dataset.

Table 6.3: Correlation of Parameters with Accuracy Results of a Model

Depth	Iterations	l2_leaf_reg	border_count	bagging_temperature
0.874869	0.707815	0.52867911	0.438907818	0.057237

These correlations heavily imply that more iterations or higher depth tend to increase accuracy. But also that a combination of high depth and iterations is the most promising, as can be seen in the best-found Parameters, which are 11 for depth and 30,000 for iterations, while a run with 50,000 iterations and lower depth yielded a worse accuracy. For more details and the in-depth analysis of each parameter's behavior in each of the tested runs, the `Param Comparisons.xlsx` file in the accompanying project is recommended.

However, this iterative tuning eventually confronted hardware limitations. Despite using a high-performance GPU for training, the prediction phase of the algorithm relied on the CPU, which became a bottleneck. This issue, combined with the tenfold increase in dataset size, caused the process to fail during the prediction phase whenever a `depth` of 12 or higher and `iterations` exceeding 20,000 were used. As a result, it is plausible that even better solutions could be achieved with access to more advanced hardware, as will be discussed in section 6.4, 'Future Work'.

It is also important to note that while increasing `iterations` and `depth` may lead to potential accuracy gains, this comes at the cost of performance. Most of the results ranged from 82% to 84% accuracy, even though some models required three to five times more computation time than others. This observation suggests that the accuracy may be approaching a plateau, given the constraints of the current implementation and dataset.

6.1.2 Accuracy on individual Projects

For further analysis, the trained model's performance was evaluated across the individual projects in the dataset. The results revealed significant variance in accuracy among the projects, which are taken from Table 3.3. The lowest accuracy observed was **71.56%** for the *termux* project, while the highest accuracy reached **96.08%** for the *Paper* project. Several projects, such as *j2cl*, *mylyn*, and *sigma*, also achieved high accuracies exceeding **93%**, indicating strong model performance on these datasets. Conversely, projects like *tomcat* and *error-prone* exhibited relatively lower accuracies at **73.28%** and **77.86%**, respectively. This variability highlights differences in the complexity or characteristics of the projects and suggests that certain datasets align better with the model's strengths than others.

Importantly, besides the three projects mentioned, for each of the 31 remaining ones, the model achieved an accuracy score of 80% or higher, often significantly, with seven even being above 90%. A file with all the individual accuracies can be found under `/modelTestResults/individual_projects/summary_accuracies.txt` in the accompanying Repository.

6.2 Risk Score

Given that resources for addressing software bugs are often limited, it is crucial to prioritize files with the highest risk while deferring low-risk files for later handling. To support this prioritization and facilitate a more intuitive visualization of the findings, a Risk Score was developed. This score is derived using the same CatBoost algorithm, with training performed similarly. However, instead of classifying data points as *bug* or *Non-bug*, each file (at a specific commit) is assigned a score ranging from 0 to 1, representing its likelihood of containing a bug.

The use of the same underlying technique as the classifier ensures comparable accuracy levels, with the Risk Score achieving **82.2%** accuracy, slightly lower than the 84.1% achieved by the classifier. This small difference can be attributed to the allocation of fewer resources for optimizing the Risk Score model and minor technical differences between the implementations.

Key Observations from the Visualizations

The analysis of Risk Scores involved various visualizations to better understand their distribution and trends. These visualizations offered distinct insights into temporal patterns, project structure, and the files most susceptible to bugs.

Heatmap of Risk Scores.

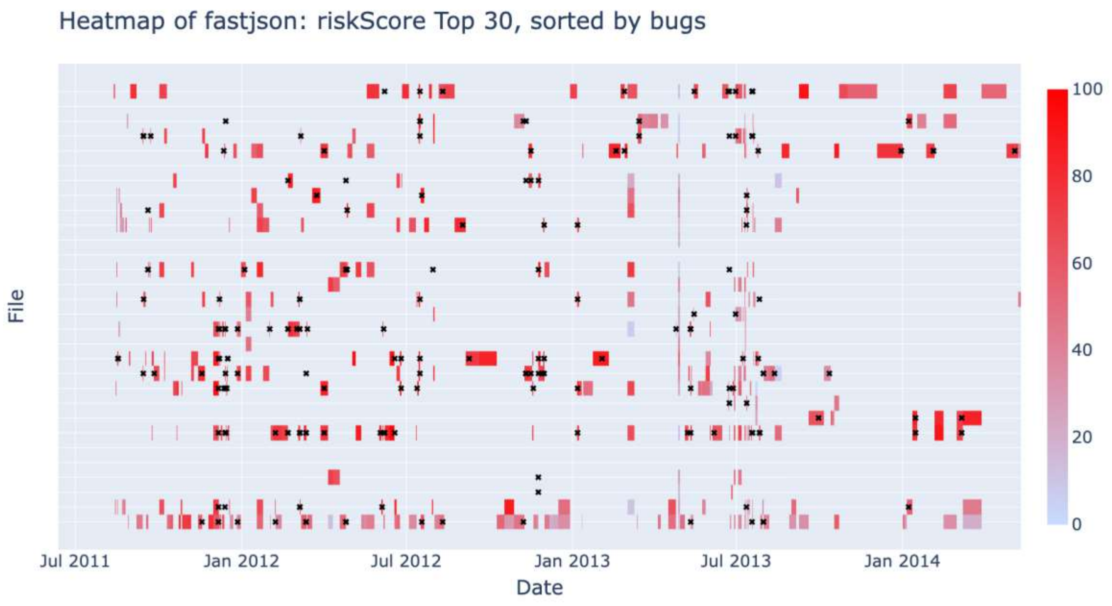

Figure 6.2: Heatmap of the top 30 files based on Risk Scores, sorted by bugs.

The heatmap (Figure 6.2) visualizes the temporal evolution of Risk Scores across the top 30 files with the highest bug counts. Each row represents a file, while the color intensity reflects the Risk Score, with darker red shades indicating higher values. Black markers are used to denote bug-related events, allowing for a direct correlation between spikes in Risk Scores and actual bug occurrences. This visualization provides a comprehensive overview of how risk evolves over time for the most bug-prone files.

From the heatmap, several key insights emerge. Files with consistently high Risk Scores over extended periods highlight systemic issues or persistent high-risk characteristics.

Temporal trends reveal that periods of heightened risk, often indicated by clusters of darker red cells, frequently coincide with significant development activities, such as the introduction of new features or large-scale code refactoring. For instance, certain time windows, such as mid-2012 and mid-2013, exhibit spikes in Risk Scores across multiple files, likely corresponding to such events. Additionally, the visualization shows that not all bug-related events (black markers) align with high Risk Scores, suggesting that while the model is effective, there are instances where other factors may influence bug occurrences.

These observations underline the importance of monitoring Risk Scores over time, as they can help developers anticipate periods of heightened risk and prioritize their efforts accordingly. For future work, a deeper investigation into why certain bug-related events occur without significant Risk Score spikes could provide valuable insights into the limitations of the current model. Additionally, incorporating this visualization into real-time development workflows could enable proactive risk management during the coding process.

Line Chart of Risk Scores for Top 5 Files.

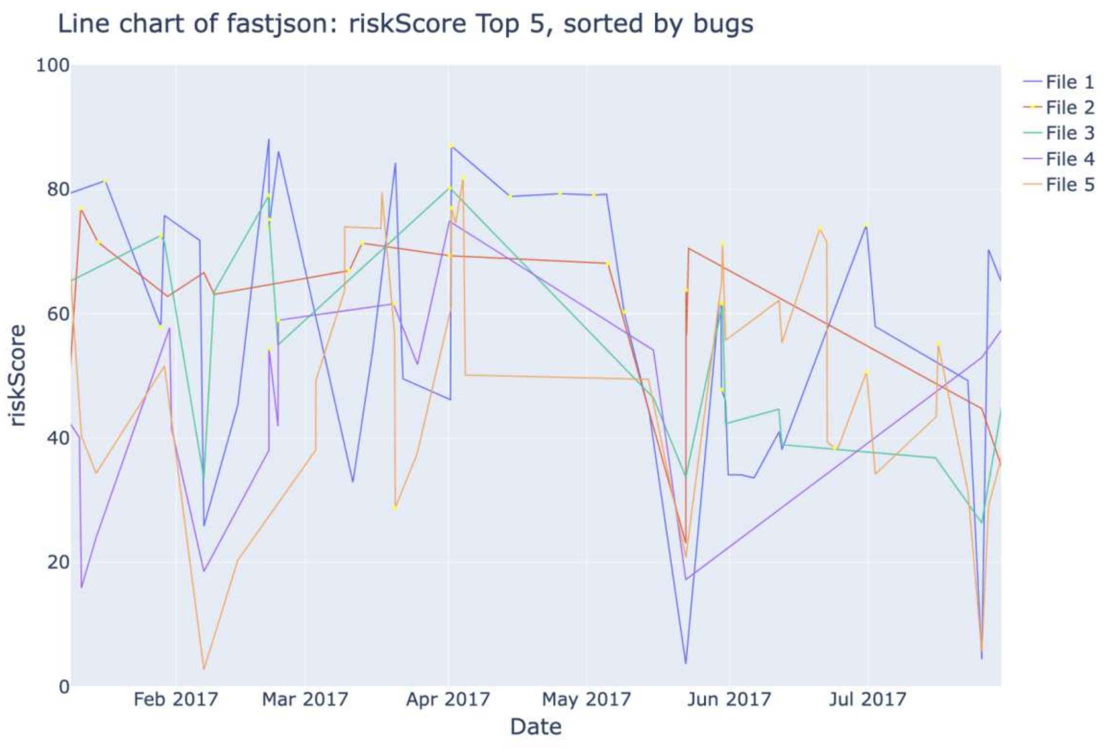

Figure 6.3: Line chart of the top 5 files by Risk Scores, sorted by bugs.

The line chart (Figure 6.3) tracks Risk Scores over time for the top 5 files with the highest bug counts. Each line represents an individual file, using distinct colors for differentiation. The chart reveals that certain files, such as File 1, exhibit consistently high Risk Scores throughout the observed period, indicating systemic issues or persistent high-risk characteristics. In contrast, files like File 2 display significant fluctuations in their Risk Scores, with spikes potentially corresponding to major feature introductions or extensive code refactoring. Additionally, an overall increase in Risk Scores across multiple files during the period from March to June 2017 suggests a phase of heightened development activity, which may have introduced additional risk. These insights help identify files requiring immediate attention versus those with periodic spikes in risk, providing actionable guidance for prioritizing bug fixes and code reviews.

Treemap of Risk Scores by Files and Folders.

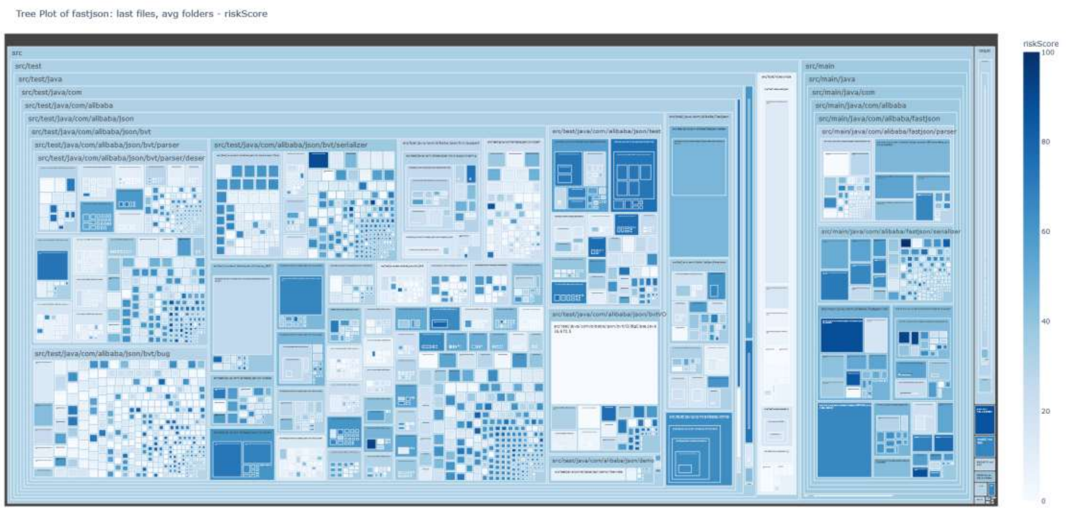

Figure 6.4: Treemap of Risk Scores by files and folders.

The treemap (Figure 6.4) provides a hierarchical visualization of the Risk Scores across the project's structure, including files and folders. Each block in the treemap represents a file or folder, with its size corresponding to the relative size of the file or folder and its color intensity reflecting the average Risk Score. Darker shades indicate higher Risk Scores, signaling areas that are more likely to contain bugs. This visualization offers an intuitive way to understand the distribution of risk within the codebase, enabling developers to quickly identify high-risk components.

From the treemap, several key insights were obtained. Certain folders and files, such as those under `src/test/java/alibaba/json/bvt`, consistently exhibit darker shades, indicating elevated Risk Scores. These areas represent hotspots that warrant immediate attention for potential bug fixes or deeper analysis. Additionally, the visualization reveals that risk is not evenly distributed across the project; some modules, such as those in the `serializer` directory, show higher average Risk Scores compared to others, suggesting that specific functionalities or file groups are more prone to bugs. Furthermore, larger blocks with darker colors, such as key files within `src/main/java/alibaba/fastjson`, highlight components that are both substantial in size and high in risk. This combination makes them critical targets for prioritization in future development efforts.

The lessons learned from this analysis emphasize the importance of focusing resources on high-risk areas, particularly those that are both large in size and high in Risk Score. These insights can inform bug-fixing strategies, helping developers prioritize their efforts effectively. For future work, integrating this visualization into real-time development tools could provide continuous feedback during the coding process, further enhancing its utility. Additionally, conducting a more detailed analysis of the identified hotspots could help uncover underlying causes of elevated risk, such as code complexity, frequent modifications, or insufficient testing coverage.

Implications

The visualizations collectively demonstrate that certain files and folders consistently exhibit high Risk Scores, aligning with bug-related events. This underscores the importance of using Risk Scores to guide resource allocation, prioritize code reviews, and mitigate potential issues in high-risk areas. Moreover, the temporal trends suggest that development phases involving significant changes correlate with increased risk, providing actionable insights for managing future projects.

By leveraging these findings, developers can proactively address high-risk components, improving overall software quality while optimizing resource use. Future work could explore integrating these visualizations into development tools to provide real-time risk assessments and insights during the coding process.

6.3 Feature Importance & Top Metrics

As summarized in section 5.2, the top three metrics most relevant for the model when determining whether a commit contains a bug are *hcpf1*, *hcpf2*, and *projectBus1*. The first two metrics, *hcpf1* and *hcpf2*, are code change entropy metrics proposed by Hassan [24], while *projectBus1* represents the implementation of the Bus factor metric, developed as part of the SAGE project.

In the case of hcpf1, a complexity value H_i of a given period is assigned to every modified file within that period. This means that all files that changed are assumed to be fully affected by the total complexity present during that period. Essentially, this approach considers that any modification to a file is influenced by the complete complexity of the period's workload, without differentiating among individual files' modification frequency or significance [24].

On the other hand, hcpf2 offers a more refined approach by assigning the complexity H_i according to the probability p_j that a specific file j is modified during period i. In this model, files that are modified more frequently are allocated a larger portion of the complexity impact. In other words, the complexity's influence is distributed proportionally, meaning that a higher frequency of change increases the likelihood that a file is more strongly tied to the overall complexity of the development activities during that period [24].

The Bus factor metric, in simple terms, addresses the question: "How many team members would need to become unavailable, for example, due to unforeseen circumstances, before the codebase becomes unmaintainable?" [34]. Essentially, it evaluates how many individuals possess critical knowledge about specific parts of the code. The *projectBus1* metric expands this concept to the scope of an entire project, assessing how many contributors possess crucial knowledge about the project as a whole. A low Bus factor often signals significant risk, as it indicates that only a small number of individuals are familiar with key aspects of the project. This risk highlights why the *projectBus1* metric plays a central role in the decision-making process of the model.

Conversely, a metric that was removed from the dataset due to disproportionately high feature importance was *totalBugfix*. In earlier iterations, this metric achieved a feature importance value of nearly 80. This meant that the model primarily relied on the historical number of bug fixes associated with a file to infer whether a commit contained a bug. Such reliance introduced a major limitation: the model would tend to overlook bugs in newly created files, as they inherently have no prior bug-fix history. This highlights one of the key challenges of using bug fixes as a proxy for bugs. To address this issue, *totalBugfix* was excluded from the dataset, demonstrating the value of feature importance analysis in identifying and mitigating such biases.

Comparison of Feature Importance and Coefficients

To further analyze the differences between the models explored in this work, a comparison is made between the feature importance rankings of the CatBoost classifier, the CatBoost-based Risk Score model, and the individual coefficients assigned to each feature in the Logistic Regression model. For this purpose, two additional visualizations are provided: a Feature Importance Graph for the Risk Score model (Figure 6.5) and a Coefficient Graph for the Logistic Regression model (Figure 6.6).

Comparison of Feature Importances: CatBoost Classifier vs. Risk Score Model

Upon examining the feature importance rankings of the CatBoost classifier and the Risk Score model (Figure 6.5), substantial overlap in the most significant metrics can be observed. This alignment indicates that the features deemed critical for the binary classification task are similarly influential in determining the Risk Score. For instance, metrics like *hcpf1, projectBus1, and age* appear prominently in both models, underscoring their universal relevance to bug prediction across different modeling approaches. This consistency reinforces the robustness of these metrics and highlights their importance in identifying buggy code segments.

Comparison of Feature Importances vs. Logistic Regression Coefficients

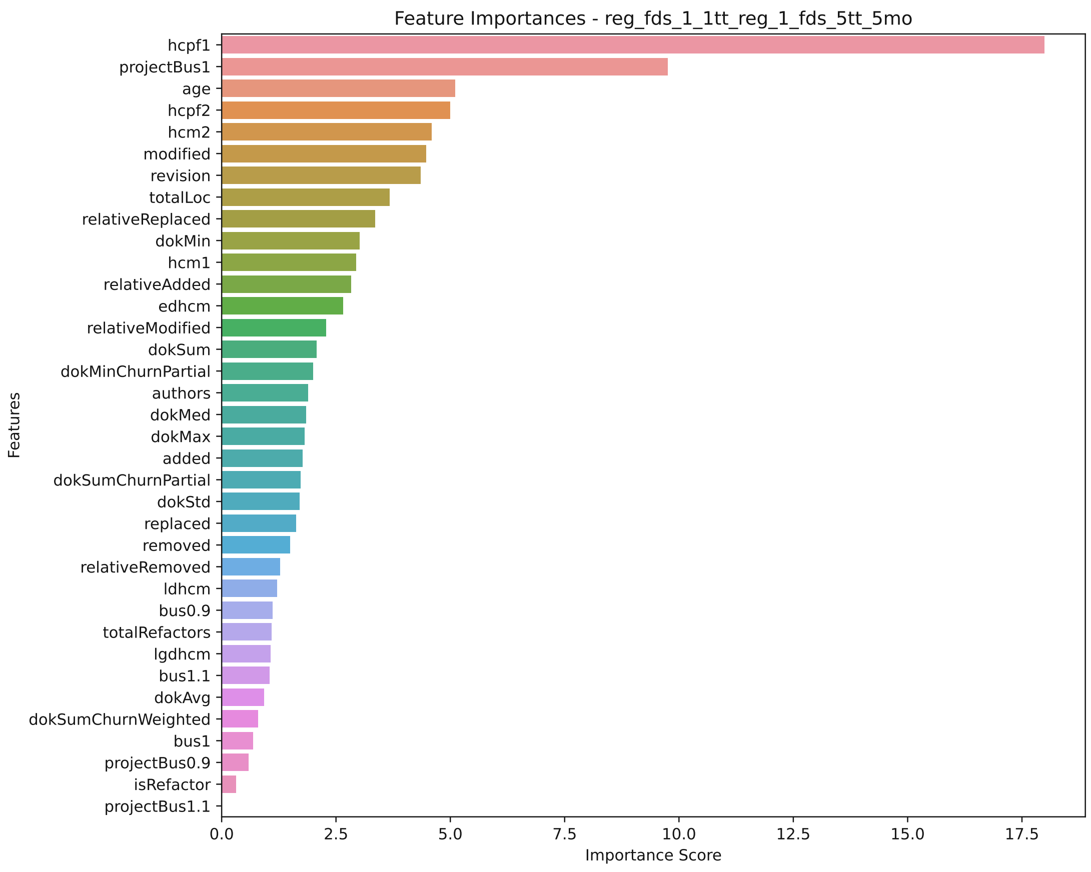

Figure 6.5: Feature Importance Chart for the CatBoost Risk Score model.

Contrasting the feature importances of the CatBoost models with the absolute values of the coefficients from the Logistic Regression model (Figure 6.6) reveals vast differences, most notably *hcpf1 and hcpf2* having no impact, while the *relative* metrics boasted the biggest impact. The Logistic Regression model's significantly lower accuracy and recall can be attributed to its inability to capture the complex, non-linear relationships inherent in the dataset, i.e., between the individual features/complexity metrics. While linear models like Logistic Regression provide a straightforward interpretation of feature contributions – the coefficients associated with the individual features – they fail to account for the nuanced interactions between metrics, leading to a distinctly different prioritization of features.

This divergence underscores the limitations of linear models for this task. The linear correlations captured by Logistic Regression offer an overly simplistic view of the problem, which does not generalize well to real-world scenarios. Consequently, the reliance on linear relationships alone, as seen in traditional correlation analyses of code metrics from the literature, proves insufficient for robust bug prediction. This finding highlights

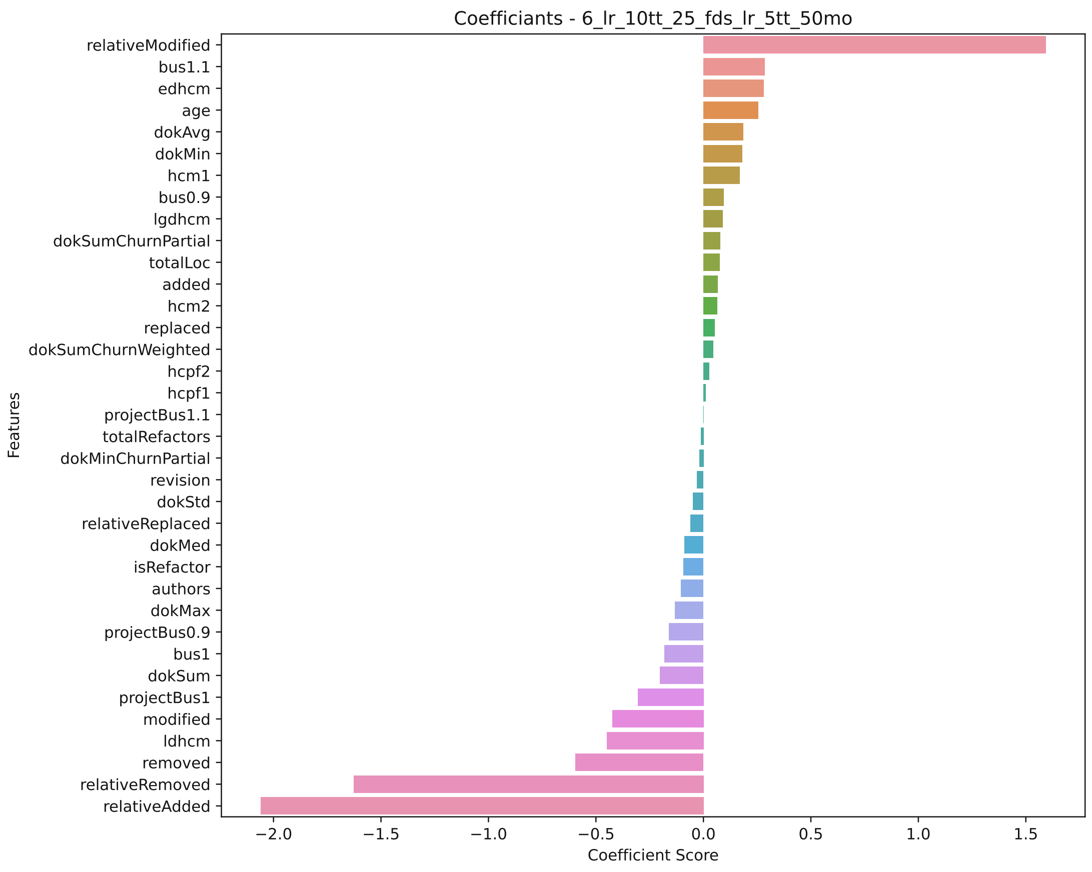

Figure 6.6: Feature-related Coefficients for the Logistic Regression model.

the necessity of employing non-linear approaches, such as the CatBoost algorithm, which dynamically models complex interactions between features, as evidenced by the significantly lower accuracy of the Logistic Regression model, achieving only 61.44%.

Implications for Metric-Based Analysis

The stark contrast between the linear and non-linear models emphasizes the importance of using advanced tools capable of capturing non-linear relationships in datasets. While correlation analyses provide insights into linear relationships, they should be approached with caution when aiming for generalized results. The use of more sophisticated, non-linear models, such as gradient boosting classifiers, is essential for accurately understanding and leveraging the intricate dynamics of code metrics. This insight aligns with the need for a shift away from solely relying on traditional correlation studies in favor of more dynamic and adaptive methodologies.

6.3.1 Lessons Learned for Software Projects

To summarize the findings from the feature importance analysis, developers should take proactive measures to ensure that their code and activities within a given period do not become overly complex. Excessive complexity can increase the likelihood of faults being introduced into the codebase.

Simultaneously, companies and project maintainers should remain vigilant when only a small number of individuals possess critical knowledge about a project or its components. This lack of knowledge distribution poses a significant risk, as it can increase the chances of bugs being introduced and hinder the project's maintainability.

6.4 Future Work

While the results achieved in this thesis demonstrate the potential of machine learning models for fault prediction, several opportunities remain to further improve and expand upon this work. Future research could focus on exploring additional machine learning algorithms, incorporating more advanced techniques such as deep learning, or experimenting with hybrid approaches. Moreover, expanding the dataset to include a broader variety of projects, programming languages, and code metrics could enhance the generalizability of the findings. Addressing the current limitations, such as balancing datasets or mitigating the impact of class imbalance, could also yield more reliable outcomes. Finally, evaluating the proposed models in real-world software development environments and integrating them into continuous integration pipelines would provide valuable insights into their practical applicability and performance.

6.4.1 Algorithm Improvement and GPU-Utilization for Prediction

As mentioned earlier when discussing hyperparameters, the final, relatively large dataset exposed some limitations of the available resources. This suggests that an even better-performing model could potentially be achieved using the same codebase but with access to more advanced computational resources.

One potential improvement involves leveraging the GPU for the prediction phase, in addition to its use during training, provided such an implementation is feasible. However, as these limitations were identified late in the research process and addressing them was unlikely to yield significant benefits while requiring substantial time, this option was not pursued. Nonetheless, this represents a straightforward and promising direction for future work.

Another avenue for enhancement lies in optimizing the algorithm itself. Although the model was refined extensively over multiple iterations, the primary focus of development was on improving accuracy rather than performance. With a deeper understanding of machine learning techniques and Python optimization, the efficiency of the algorithm could likely be significantly enhanced.

6.4.2 State-of-the-art and Algorithm comparison

The research revealed a surprising lack of meta-studies comparing different machine learning techniques for bug prediction. Even among existing studies, meaningful comparisons are often hindered by variations in datasets, methodologies, and other influencing factors. Addressing this gap could offer valuable insights and context for future research, making it an important direction for extending this work.

Future research could explore the development and evaluation of multiple algorithms to compare their performance. This thesis focused exclusively on the CatBoost classifier, which demonstrated its effectiveness. However, other promising approaches, such as Support Vector Machines or Naive Bayes, could also be investigated. Comparing these methods could provide a broader understanding of which techniques are most suitable for bug prediction across different contexts.

6.4.3 Risk Score

Due to the binary nature of the training labels, the Risk Score does not account for the varying severity or complexity of bugs. Future work could focus on refining the Risk Score calculation by incorporating additional features, such as bug severity or historical patterns of bug occurrence. Furthermore, the model's ability to generalize the Risk Score across different datasets and projects could be evaluated more extensively.

6.4.4 Increase Dataset Size and In-Depth Metric and Project Analysis

An additional promising direction for future work is to expand the size of the dataset. A larger dataset would enable the model to generalize more effectively, which could lead to improved accuracy and enhanced performance when applied to new projects.

Similarly, conducting a more comprehensive analysis of the impact of individual metrics, essentially an extension of the feature importance analysis performed in this thesis, could yield valuable insights. Such an analysis would be particularly beneficial for understanding metric relevance in different contexts and for optimizing the model's application to future projects.

Additionally, the observed variability in the complexity and characteristics of the projects in the dataset, as discussed in section 6.1, presents another avenue for future exploration. Investigating the specific differences between projects and identifying factors that align certain datasets more closely with the model's strengths could provide valuable insights. Such an effort could enhance the model's adaptability and effectiveness when applied across a broader range of projects.

6.4.5 Bugfixes as a Proxy

One particularly promising avenue for future work lies in addressing the reliance on bug fixes as a proxy for actual bugs, a common practice both in this thesis and in the field at large. While it is true that bug fixes inherently indicate the presence of bugs, being able to train directly on bugs themselves, rather than their proxies, would provide significant advantages. Such an approach could enable more targeted and effective training, improving both model accuracy and practical applicability.

However, tracing bugs directly poses a considerable challenge, particularly when attempting to automate the process. The difficulty is compounded when working with datasets of substantial size, such as the one used in this thesis. Despite these challenges, successfully addressing this limitation would likely yield substantial benefits, making it one of the most impactful areas for further research.

CHAPTER 7

Conclusion

This thesis explored the application of machine learning models for fault prediction in software development. By integrating code metrics with supervised learning techniques, a classifier was developed that demonstrated notable accuracy in identifying buggy code.

The primary contribution of this thesis, the CatBoost classifier model, achieved a noteworthy accuracy score of **84.1%** on data not used during training. This high performance reflects the model's ability to generalize effectively across the dataset, which comprised 34 open-source projects spanning various domains. Additionally, the model demonstrated a recall value of **83%** for detecting buggy files, indicating balanced training with no significant bias toward either class. This equilibrium between precision and recall underscores the robustness of the classifier and its reliability in distinguishing buggy from non-buggy files.

The classifier operates by leveraging a diverse set of **37 code metrics**, ranging from simple measures, such as *Lines of Code* (LoC), to more sophisticated metrics like code churn and *projectBus1*. Feature importance analysis revealed that metrics such as *projectBus1* and code churn played pivotal roles in predicting buggy files, providing valuable insights into which aspects of code contribute most significantly to fault prediction. The model's gradient-boosting approach enabled it to iteratively refine its predictions, optimizing performance through careful hyperparameter tuning. Key parameters, such as `depth`, `iterations`, and `learning_rate`, were refined over several months to strike a balance between accuracy and computational efficiency.

Trained on historical software data from **34 open-source real-world projects** – including well-known repositories by *Eclipse*, *Apache*, and *Google*, each characterized by varying codebases, complexity levels, and development practices – the machine learning model demonstrated strong predictive capabilities for identifying bug-prone files and assessing risk.

In addition to its classification capabilities, the CatBoost model was adapted to assign a **Risk Score**, quantifying the likelihood of a given code segment containing bugs. This was achieved by transforming the classification model into a regression model, allowing the output to range between 0 and 1. The Risk Score model achieved an accuracy of **82.2%**, providing a more granular perspective and enabling developers to prioritize code reviews and mitigate potential issues proactively.

Compared to alternative approaches, such as regression-based models, which achieved accuracies of only 58% and 61%, the CatBoost classifier demonstrated clear superiority. The model's strong performance underscores the value of combining advanced supervised machine learning methods with carefully selected code metrics for fault prediction.

This research contributes to ongoing efforts in fault prediction by validating the utility of combining code metrics with advanced machine learning methods. The findings can aid developers in identifying risky code components more effectively, improving software quality, and reducing costs. The Risk Score further enhances this utility by enabling prioritization of the most vulnerable components. In the future, the developed algorithm could be integrated into development environment plug-ins or as standalone code analysis software.

Despite the promising results, the model's performance may vary when applied to new datasets or different programming languages. Additionally, the use of bug-fixing commits as a proxy for bugs remains a constraint that could influence outcomes. Investigating the differences in project characteristics, as well as analyzing how specific metrics align with the model's strengths, could yield valuable insights. These areas represent promising directions for future work.

Future research could also explore the integration of deep learning techniques or hybrid models, as well as expanding the dataset size or conducting deeper feature importance analysis to further improve fault prediction.

In conclusion, this thesis highlights the potential of machine learning in fault prediction, offering a robust approach to improving software quality. By addressing the outlined challenges and building upon this foundation, further advancements can be made in automating and refining the bug prediction process.

Overview of Generative AI Tools Used

During the preparation of this thesis, OpenAI's ChatGPT-4o mini model [69] was utilized as a supplementary tool. Its usage was primarily limited to debugging code and refining textual content, such as rephrasing and spell-checking.

When applied to code beyond debugging, the model functioned essentially as an enhanced and efficient search engine, enabling quick retrieval of information that could otherwise be found on platforms like StackOverflow.

Overall, the AI served as a review and support mechanism for pre-existing content, whether in the form of text or code, which was originally authored by the thesis creator.

Bibliography

[1] H. Krasner, “The cost of poor software quality in the us: A 2020 report,” *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, vol. 2, 2021.

[2] F. Hashim. (2018) Lion air 737 max 8 crash confirmed, 189 dead. Visited on 2024-11-21. [Online]. Available: https://www.flightglobal.com/safety/lion-air-737-max-8-crash-confirmed-189-dead/130047.article

[3] BBC. (2019) Ethiopian airlines: ’no survivors’ on crashed boeing 737. Visited on 2024-11-21. [Online]. Available: https://www.bbc.com/news/world-africa-47513508

[4] D. K. Linnan, “Iran air flight 655 and beyond: Free passage, mistaken self-defense, and state responsibility,” *Yale J. Int’l L.*, vol. 16, p. 245, 1991.

[5] H. Haq. (2010) Toyota recall update: dealers face full lots, anxious customers. Visited on 2024-11-21. [Online]. Available: https://www.csmonitor.com/USA/2010/0129/Toyota-recall-update-dealers-face-full-lots-anxious-customers

[6] CBS. (2010) Toyota "unintended acceleration" has killed 89. Visited on 2024-11-21. [Online]. Available: https://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/

[7] L. K. Wee. (2024) Here comes the wave of insurance claims for the crowdstrike outage. Visited on 2024-11-21. [Online]. Available: https://www.businessinsider.com/businesses-claiming-losses-crowdstrike-outage-insurance-billions-losses-cyber-policies-2024-7

[8] D. Weston. (2024) Helping our customers through the crowdstrike outage. Visited on 2024-11-21. [Online]. Available: https://blogs.microsoft.com/blog/2024/07/20/helping-our-customers-through-the-crowdstrike-outage/

[9] H. Dolfing. (2019) The $440 million software error at knight capital. Visited on 2024-11-19. [Online]. Available: https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html

[10] M. Dowson, “The ariane 5 software failure,” *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 2, p. 84, Mar. 1997, Accessed on 2024-12-18. [Online]. Available: https://doi.org/10.1145/251880.251992

[11] B. J. Sauser, R. R. Reilly, and A. J. Shenhar, “Why projects fail? how contingency theory can provide new insights - a comparative analysis of nasa’s mars climate orbiter loss,” *International Journal of Project Management*, vol. 27, no. 7, pp. 665–679, 2009, Accessed on 2024-12-18. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0263786309000052

[12] Synopsys Inc. The heartbleed bug. Visited on 2024-11-19. [Online]. Available: https://heartbleed.com/

[13] C. Isidore. (2020) Boeings 737 max debacle could be the most expensive corporate blunder ever. Visited on 2024-11-21. [Online]. Available: https://edition.cnn.com/2020/11/17/business/boeing-737-max-grounding-cost/index.html

[14] E. Giger, M. D’Ambros, M. Pinzger, and H. C. Gall, “Method-level bug prediction,” in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 171–180, Accessed on 2024-12-18. [Online]. Available: https://doi.org/10.1145/2372251.2372285

[15] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, “A developer centered bug prediction model,” *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 5–24, 2018.

[16] G. R. Choudhary, S. Kumar, K. Kumar, A. Mishra, and C. Catal, “Empirical analysis of change metrics for software fault prediction,” *Computers & Electrical Engineering*, vol. 67, pp. 15–24, 2018, Accessed on 2024-12-18. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790617336121

[17] T. Graves, A. Karr, J. Marron, and H. Siy, “Predicting fault incidence using software change history,” *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[18] M. D’Ambros, M. Lanza, and R. Robbes, “An extensive comparison of bug prediction approaches,” in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 2010, pp. 31–41.

[19] T. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[20] C. E. Walston and C. P. Felix, “A method of programming measurement and estimation,” *IBM Systems Journal*, vol. 16, no. 1, pp. 54–73, 1977.

[21] SonarSource SA. Sonarqube. Visited on 2024-11-20. [Online]. Available: https://www.sonarsource.com/products/sonarqube/

[22] G. A. Campbell, “Cognitive complexity-a new way of measuring understandability,” *SonarSource SA*, p. 10, 2018.

[23] S. Chidamber and C. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[24] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 78–88.

[25] S. Kim, E. J. Whitehead, and Y. Zhang, “Classifying software changes: Clean or buggy?” *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[26] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, “Reducing features to improve bug prediction,” in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 600–604.

[27] M. Pradel and K. Sen, “Deepbugs: a learning approach to name-based bug detection,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018, Accessed on 2024-12-18. [Online]. Available: https://doi.org/10.1145/3276517

[28] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 297–308, Accessed on 2024-12-18. [Online]. Available: https://doi.org/10.1145/2884781.2884804

[29] P. König, F. Obermann, K. Mallinger, and A. Schatten, “Root cause analysis of software aging in critical information infrastructure,” in *Critical Information Infrastructures Security*, B. Hämmerli, U. Helmbrecht, W. Hommel, L. Kunczik, and S. Pickl, Eds. Cham: Springer Nature Switzerland, 2023, pp. 3–8.

[30] SBA Research gGmbH. Core - sba research. Visited on 2024-12-18. [Online]. Available: https://www.sba-research.org/research/research-groups/core/

[31] S. R. Chidamber and C. F. Kemerer, “Towards a metrics suite for object oriented design,” in *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA ’91. New York, NY, USA: Association for Computing Machinery, 1991, pp. 197–211, Accessed on 2024-12-18. [Online]. Available: https://doi.org/10.1145/117954.117970

[32] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, “A degree-of-knowledge model to capture source code familiarity,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE ’10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 385–394, Accessed on 2025-01-17. [Online]. Available: https://doi.org/10.1145/1806799.1806856

[33] J. Munson and S. Elbaum, “Code churn: a measure for estimating the impact of code change,” in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, 1998, pp. 24–31.

[34] E. Jabrayilzade, M. Evtikhiev, E. Tüzün, and V. Kovalenko, "Bus factor in practice," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 97–106, Accessed on 2024-12-18. [Online]. Available: https://doi.org/10.1145/3510457.3513082

[35] J. M. Scanlon, K. D. Kusano, T. Daniel, C. Alderson, A. Ogle, and T. Victor, "Waymo simulated driving behavior in reconstructed fatal crashes within an autonomous vehicle operating domain," *Accident Analysis & Prevention*, vol. 163, p. 106454, 2021, Accessed on 2024-12-18. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0001457521004851

[36] K. Kourou, T. P. Exarchos, K. P. Exarchos, M. V. Karamouzis, and D. I. Fotiadis, "Machine learning applications in cancer prognosis and prediction," *Computational and Structural Biotechnology Journal*, vol. 13, pp. 8–17, 2015, Accessed on 2024-12-18. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2001037014000464

[37] E. A. Gerlein, M. McGinnity, A. Belatreche, and S. Coleman, "Evaluating machine learning classification for financial trading: An empirical approach," *Expert Systems with Applications*, vol. 54, pp. 193–207, 2016, Accessed on 2024-12-18. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417416000282

[38] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023, Accessed on 2024-12-18. [Online]. Available: https://arxiv.org/pdf/2303.08774

[39] Gemini Team et al., "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023, Accessed on 2024-12-18. [Online]. Available: https://arxiv.org/pdf/2312.11805

[40] Apple Inc. Apple intelligence. Visited on 2024-11-07. [Online]. Available: https://www.apple.com/apple-intelligence/

[41] N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[42] Google. Supervised versus unsupervised learning. Visited on 2024-10-29. [Online]. Available: https://cloud.google.com/discover/supervised-vs-unsupervised-learning?hl=en

[43] J. Delua. (2021) Supervised versus unsupervised learning. Visited on 2024-10-29. [Online]. Available: https://www.ibm.com/think/topics/supervised-vs-unsupervised-learning

[44] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, “Handwritten digit recognition with a back-propagation network,” in *Advances in Neural Information Processing Systems*, D. Touretzky, Ed., vol. 2. Morgan-Kaufmann, 1989, Visited on 2024-12-18. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1989/file/53c3bce66e43be4f209556518c2fcb54-Paper.pdf

[45] S. B. Kotsiantis, I. D. Zaharakis, and P. E. Pintelas, “Machine learning: a review of classification and combining techniques,” *Artificial Intelligence Review*, vol. 26, no. 3, pp. 159–190, 2006.

[46] scikit-learn developers. Random forest classifier by scikit-learn. Visited on 2024-10-30. [Online]. Available: https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[47] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[48] A. Natekin and A. Knoll, “Gradient boosting machines, a tutorial,” *Frontiers in Neurorobotics*, vol. 7, 2013, Accessed on 2024-12-18. [Online]. Available: https://www.frontiersin.org/journals/neurorobotics/articles/10.3389/fnbot.2013.00021

[49] J. H. Friedman, “Greedy function approximation: A gradient boosting machine,” *The Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001, Accessed on 2024-12-18. [Online]. Available: http://www.jstor.org/stable/2699986

[50] A. V. Dorogush, V. Ershov, and A. Gulin, “Catboost: gradient boosting with categorical features support,” 2018, Accessed on 2024-12-18. [Online]. Available: https://arxiv.org/abs/1810.11363

[51] Kaggle. Kaggle - state of data science and machine learning 2021. Visited on 2024-10-30. [Online]. Available: https://www.kaggle.com/kaggle-survey-2021

[52] scikit-learn developers. Elasticnet by scikit-learn. Visited on 2024-10-30. [Online]. Available: https://scikit-learn.org/dev/modules/generated/sklearn.linear_model.ElasticNet.html

[53] scikit-learn developers. Logistic regression by scikit-learn. Visited on 2024-10-30. [Online]. Available: https://scikit-learn.org/1.5/modules/generated/sklearn.linear_model.LogisticRegression.html

[54] NumFOCUS Inc. Pandas - python data analysis library. Visited on 2024-10-31. [Online]. Available: https://pandas.pydata.org/

[55] J. R. Quinlan, “Induction of decision trees,” *Machine learning*, vol. 1, pp. 81–106, 1986.

[56] R. A. Disha and S. Waheed, “Performance analysis of machine learning models for intrusion detection system using gini impurity-based weighted random forest (giwrf) feature selection technique,” *Cybersecurity*, vol. 5, no. 1, p. 1, 2022, Accessed on 2025-01-17. [Online]. Available: https://doi.org/10.1186/s42400-021-00103-8

[57] scikit-optimize: sequential model-based optimization in python. Visited on 2024-10-31. [Online]. Available: https://scikit-optimize.github.io/stable/

[58] RapidsAI. cuml - rapids machine learning library. Visited on 2024-10-31. [Online]. Available: https://github.com/rapidsai/cuml

[59] S. Raubitzek, L. Corpaci, R. Hofer, and K. Mallinger, “Scaling exponents of time series data: A machine learning approach,” *Entropy*, vol. 25, no. 12, 2023, Accessed on 2025-01-17. [Online]. Available: https://www.mdpi.com/1099-4300/25/12/1671

[60] S. Raubitzek and K. Mallinger, “On the applicability of quantum machine learning,” *Entropy*, vol. 25, no. 7, 2023, Accessed on 2025-01-17. [Online]. Available: https://www.mdpi.com/1099-4300/25/7/992

[61] Yandex. Catboost - open-source gradient boosting. Visited on 2024-10-31. [Online]. Available: https://catboost.ai/

[62] H. He, Y. Bai, E. A. Garcia, and S. Li, “Adasyn: Adaptive synthetic sampling approach for imbalanced learning,” in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 2008, pp. 1322–1328.

[63] The imbalanced-learn developers. Randomundersampler by imbalanced-learn. Visited on 2024-10-31. [Online]. Available: https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html

[64] H. Zou and T. Hastie, “Regularization and variable selection via the elastic net,” *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 67, no. 2, pp. 301–320, 03 2005, Accessed on 2025-01-17. [Online]. Available: https://doi.org/10.1111/j.1467-9868.2005.00503.x

[65] A. Ng and M. Jordan, “On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes,” in *Advances in Neural Information Processing Systems*, T. Dietterich, S. Becker, and Z. Ghahramani, Eds., vol. 14. MIT Press, 2001, Accessed on 2025-01-17. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2001/file/7b7a53e239400a13bd6be6c91c4f6c4e-Paper.pdf

[66] NVIDIA Corporation. Cuda toolkit. Visited on 2024-11-05. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[67] Anaconda Inc. Anaconda - the operating system for ai. Visited on 2024-11-05. [Online]. Available: https://www.anaconda.com/

[68] RapidsAI. Installation guide - rapids docs. Visited on 2024-11-05. [Online]. Available: https://docs.rapids.ai/install/#wsl2

[69] OpenAI. Gpt-4o mini. Visited on 2024-12-03. [Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/