UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Geanderson Esteves dos Santos

Understanding Software Defects with Machine Learning

Belo Horizonte
2023

Geanderson Esteves dos Santos

Understanding Software Defects with Machine Learning

Final Version

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Eduardo Figueiredo
Co-Advisor: Adriano Veloso

Belo Horizonte
2023

Santos, Geanderson Esteves dos.

S237u Understanding software defects with machine learning [recurso eletrônico] / Geanderson Esteves dos Santos – 2023.
1 recurso online (152 f. il, color.) : pdf.

Orientador: . Eduardo Magno Lages Figueiredo.
Coorientador: Adriano Alonso Veloso.
Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciências da Computação.
Referências: f. 128 -143

1. Computação – Teses. 2. Aprendizado de máquina – Teses. 3. Predição de falhas – Teses. 4. Código fonte (Computação) – Teses. I. Figueiredo,Eduardo Magno Lages II. Veloso, Adriano Alonso. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. IV.Título.

CDU 519.6*82(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6/1510 Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

UNDERSTANDING SOFTWARE DEFECTS WITH MACHINE LEARNING

GEANDERSON ESTEVES DOS SANTOS

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Eduardo Magno Lages Figueiredo - Orientador
Departamento de Ciência da Computação - UFMG

Prof. Adriano Alonso Veloso - Coorientador
Departamento de Ciência da Computação - UFMG

Prof. Ivan do Carmo Machado
Instituto de Computação - UFBA

Prof. Valter Vieira de Camargo
Departamento de Computação - Universidade Federal de São Carlos

Prof. Marco Túlio de Oliveira Valente
Departamento de Ciência da Computação - UFMG

Prof. Wagner Meira Júnior
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 13 de fevereiro de 2023.

Documento assinado eletronicamente por **Eduardo Magno Lages Figueiredo**, **Professor do Magistério Superior**, em 15/03/2023, às 14:06, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Adriano Alonso Veloso**, **Professor do Magistério Superior**, em 15/03/2023, às 16:43, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Marco Tulio de Oliveira Valente**, **Professor do Magistério Superior**, em 15/03/2023, às 16:51, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Valter Vieira de Camargo**, **Usuário Externo**, em 21/03/2023, às 18:36, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Wagner Meira Junior**, **Professor do Magistério Superior**, em 22/03/2023, às 13:51, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Ivan do Carmo Machado**, **Usuário Externo**, em 24/03/2023, às 12:34, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2146913** e o código CRC **EB5DC348**.

To my wife, son, parents, and sister.

Acknowledgments

First of all, I would like to thank my advisor, Eduardo Figueiredo, for always being very helpful and patient during these years of my Ph.D. His teachings were valuable, not only for the conclusion of this thesis but also for my personal and academic growth as a student. I will carry them with me forever. Furthermore, I would like to thank Professors Adriano Veloso and Nivio Ziviani for their valuable feedback. I would also like to express my gratitude to Professors Glívia Barbosa and Rosilane Mota for giving me the opportunity to work with academic research during the initial phase of my undergraduate degree at PUC Minas. This opportunity sparked my interest in the academic field, while many of my colleagues were only focused on getting a job. Lastly, I thank Professor Maria Augusta, who served as a mentor in my academic and professional career, and contributed significantly to important decisions during my undergraduate studies.

In addition, I would like to thank my parents, my mother Silvana, and my father Getúlio, for their dedication, love, and trust in my dreams. My mother Silvana has always supported me, even when I did not deserve it, and my father Getúlio has worked so hard to take care of our family. I also want to thank my grandmother Maura (in memoriam), who was like a second mother to me, and whom I still miss from the bottom of my heart. I would like to acknowledge my sister Gerliane, who has been such a great example in my life. I am grateful for the support and hard work of my closest aunt and uncle, my aunt Éloa, who is an example of incredible hard work in everything she does. I am also deeply saddened by the recent passing of my uncle Zé (in memoriam), and I cannot stop thinking about him. Most importantly, I want to thank my wife Theresa. I am so grateful for the love, affection, and support she has shown me in every moment of my life. I could write a whole book about how much she has changed my life for the better. I thank her for being with me all the time, and even in the most difficult moments, never giving up on our dreams. Moreover, she gave me our son Gabriel, who I already love so much. This victory is not only mine, but mainly hers.

I would also like to express my gratitude to Kunumi and Loggi for supporting me during these past couple of years, especially when I needed to take time off work to attend classes or meetings with Labsoft. I could not have completed my thesis without the incredible support of these companies. I am also grateful to my former manager, Allan, for always being there to help me whenever I needed it. Furthermore, I am thankful to all the colleagues I met while working at these companies for their support over the years. While my Ph.D. was always my main goal, working as a software engineer has provided

me with the financial support to achieve many other goals in my life.

Moreover, I would like to express my gratitude to PUC Minas, especially for the opportunity to teach for five semesters during my Ph.D. I am very grateful for having had the support of Professors Rodrigo Richard and Wladmir Brandão throughout my time at PUC Minas. They have supported me since my days as a student, and they continue to believe in my work as a colleague in the computer science department. During these semesters, I was able to share my knowledge of computers with my students. I am particularly thankful to Wladmir for allowing me to work with him in the IRIS Laboratory at PUC Minas and for publishing several papers together in recent years. Being a supervisor is just as rewarding as being a student, and I am thrilled to have had the opportunity to work with him in his laboratory.

In addition, I would like to thank my colleagues at LabSoft who have contributed to my work by providing relevant feedback in every paper I have published. Special thanks to João, Jonathan, and Euler for always providing feedback on my work. It is great to have a group that is willing to collaborate with other members. Besides our group, I thank other colleagues from UFMG, especially Pedro and Aline, for always discussing my work and helping me with this process at UFMG. I would also like to thank my co-authors Markos, Amanda, Igor, and Gustavo for the incredible opportunity to work together. I have learned a lot from all of you. Furthermore, I am very grateful to all the staff and professors of the DCC/UFMG who have made this university a reference in Brazil. Special thanks to the defense board that kindly accepted the invitation to participate in my thesis defense, professors Ivan Machado, Valter Camargo, Marco Túlio, and Wagner Meira. Finally, I am grateful for the quality of public education in our country.

"*Computer science is no more about computers than astronomy is about telescopes.*"
(Edsger Dijkstra)

Resumo

A predição de defeitos representa uma área de interesse tanto no meio acadêmico quanto na indústria. Os defeitos são comuns no desenvolvimento de software e podem gerar muitas dificuldades para gerentes de projetos, usuários, e desenvolvedores. Estudos recentes revelam que cerca de 42% do orçamento de desenvolvimento é gasto corrigindo defeitos. Embora a literatura atual ofereça múltiplas abordagens para prever a probabilidade de defeitos, ainda existe uma falta de compreensão sobre as características que contribuem para os defeitos. Além disso, a maioria destes estudos concentra-se na predição de defeitos a partir de um amplo conjunto de características. Entretanto, o poder discriminador individual das características ainda é desconhecido, já que algumas têm um bom desempenho apenas em projetos específicos. Por essa razão, nesta tese, nosso objetivo é compreender as características que afetam os defeitos em projetos de software. Para isso, aplicamos técnicas de aprendizado de máquina em conjuntos de dados populares. Portanto, realizamos uma investigação exploratória que produziu milhares de modelos a partir de uma coleção diversa de características. Estes modelos são aleatórios porque selecionam as características de todo o conjunto de características. Embora a imensa maioria dos modelos seja ineficaz, conseguimos produzir vários modelos que fornecem previsões precisas. Logo, os modelos distinguem classes propensas a defeitos de classes que não tenham defeitos. Concentramos nossa investigação em modelos que classificam com mais de 85% de precisão uma classe defeituosa. Assim, utilizamos esses resultados para discutir um conjunto de características que contribuem para a explicabilidade do modelo. Como resultado, notamos que os modelos mais eficientes são fáceis de entender, pois dependem de um conjunto pequeno de características. Além disso, comparamos o limite dessas características. Para validar os resultados, realizamos uma pesquisa com 40 desenvolvedores para medir suas percepções sobre os modelos e concluímos que os modelos são bastante explicáveis. Complementarmente, também avaliamos a percepção dos desenvolvedores sobre os atributos de qualidade com desenvolvedores ativos do GitHub, onde obtivemos 54 participantes. Assim, concluímos que as percepções dos desenvolvedores diferem significativamente dos modelos. Finalmente, comparamos as similaridades entre os modelos de predição de defeito com o mau cheiro do código. Ao final, esta tese promove o raciocínio sobre quais características de software influenciam os defeitos desses projetos.

Palavras-chave: predição de defeitos, aprendizado de máquina explicável, características do código

Abstract

Software defect prediction represents an area of interest in both academia and industry. In fact, defects are prevalent in software development and might generate numerous difficulties for project managers, users, stakeholders, and developers. Recent studies reveal that approximately 42% of the software development budget goes to fixing defects. Although the current literature offers multiple alternative approaches to predict the likelihood of defects, there is a lack of understanding about the features that contribute to the defects of a software project. Furthermore, most of the literature concentrates on predicting defects from a broad set of features. However, the individual discriminating power of software features is still unknown as some perform well only with specific projects. For this reason, in this thesis, we aim at understanding the features that impact the defectiveness of software projects. To do so, we applied machine learning techniques to popular datasets. Hence, we convey an exploratory investigation that produced thousands of models from a diverse collection of software features. These models are random because they promptly select the features from the entire pool of software features. Even though the immense majority of models are ineffective, we could produce several models that yield accurate predictions. Thus, the models distinguish defect-prone classes from clean ones. We focus our investigation on models that rank a randomly chosen defective software class higher than a randomly selected non-defective class with over 85% accuracy. More importantly, we employ these results to discuss a set of features contributing to the understandability of model decisions. As a result, we notice that the best-performing models are simple to understand as they rely on a small set of features. Therefore, we present which features contribute to the defects of twelve projects. Further, we also compare the threshold of these features. To validate the results, we survey 40 developers to measure their perceptions of the models and conclude that the models are fairly understandable. Complementary, we also evaluate developers' perception of the quality attributes with active GitHub developers, where 54 participated in the investigation. Then, we conclude that developers' perceptions differ significantly from the machine learning models in terms of quality attributes. Finally, we compare the redundancies and similarities between defect models with code smell as they share several features. By the end, this thesis promotes reasoning on which software features influence the defects of these projects.

Keywords: defect prediction, explainable machine learning, source code features.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

With the continuous expansion of software development, the reliability of software systems has become a key concern [Nagappan et al., 2010, Jing et al., 2014, Wang et al., 2016a, Tantithamthavorn and Hassan, 2018]. The intrinsic complexity of software systems may cause defects, leading them to collapse in various stages of development. To assist project managers and developers in anticipating defects, software defect prediction[1] is one of the research directions applying machine learning techniques to software engineering [Menzies and Zimmermann, 2013, Ghotra et al., 2015]. In this area, prior studies have statistically analyzed defect data investigating the impact of code features [Nagappan and Ball, 2005, Nagappan et al., 2006, Menzies et al., 2007, D'Ambros et al., 2010, Nagappan et al., 2010, Menzies et al., 2010, Menzies and Zimmermann, 2013, Madeyski and Jureczko, 2015, Tantithamthavorn et al., 2019], development activities [Hindle et al., 2009, Shihab et al., 2010, Wang et al., 2016a], code smells [Khomh et al., 2009, 2012, Cruz et al., 2020], object-oriented programming [Chidamber and Kemerer, 1994, Moser et al., 2008, Binanto et al., 2018], continuous integration [Storey et al., 2008, Vasilescu et al., 2015, Vassallo et al., 2018], and the interpretation of software features in defect prediction [Mori and Uchihira, 2018, Jiarpakdee et al., 2020]. This area is still strengthening as software engineering and machine learning techniques join efforts to predict defects.

Despite the undeniable importance of existing efforts for defect prediction, these studies are concerned with limited aspects of the source code [Hassan, 2009, Nagappan et al., 2010, D'Ambros et al., 2010, Jing et al., 2014, Tantithamthavorn et al., 2015, Wang et al., 2016b, Xu et al., 2018, Tantithamthavorn and Hassan, 2018]. In fact, these works lack an understanding of why the machine learning model has predicted a target software class as defective-prone. This problem happens because machine learning models consider features as individual inputs to the machine learning algorithm. Thus, the current literature needs an overview of the software features as groups serving as predictors for the defective task. Recently, other studies have taken the first steps into explainable defect prediction, using tools such as LIME and BreakDown [Mori and Uchihira, 2018, Jiarpakdee et al., 2020]. While explaining the decisions made by the models is key to

[1] In this thesis, we use the term "prediction" because it is more commonly used in the current literature, even though most of the data we applied relates to "identification".

understanding the arrangement of defects and avoiding them, the software engineering literature often left aside this topic for a considerable time [Ghotra et al., 2015, Agrawal and Menzies, 2018]. Furthermore, little is known about the discriminating ability of software features proposed and used in the current literature on software defect prediction [Mori and Uchihira, 2018].

For this reason, this thesis investigates the application of machine learning techniques to predict software defects. More importantly, we aim to understand these machine learning models and how each software feature impacts the defectiveness of the target classes [Menzies and Zimmermann, 2013, Lundberg et al., 2018a]. To do so, we rely on an algorithm to explain the software features that interfere with the defectiveness of software classes [Lundberg and Lee, 2017, Lundberg et al., 2018b, 2020]. We then explore a large set of software defects located in publicly available datasets for defect prediction [Jureczko and Spinellis, 2010, Jureczko and Madeyski, 2010, D'Ambros et al., 2010, Menzies et al., 2010]. In this manner, we examine the predictive power of these software features and their capacity to explain the defects to stakeholders [Mori and Uchihira, 2018, Jiarpakdee et al., 2020]. Finally, we examine developers' understanding of the software features and their importance in software defect prediction.

1.1 Problem and Motivation

Software defects are a problem for developers, users, and stakeholders alike. Defects are not only able to reduce software quality and increase the software budget but also suspend the development schedule [Fowler, 1999]. As a result, finding and fixing defects cost a lot of money [Knab et al., 2006]. For instance, data from the US Department of Defense demonstrates that the United States spent around 780 billion dollars fixing existing software defects [Knab et al., 2006]. In addition, the US Department of Defense also estimates that around 42% of the software development budget in Information Technology (IT) products is spent on fixing defects [Knab et al., 2006]. This is a significant amount of money that could be used to develop new features and improve the quality of the software. Therefore, it is important to find ways to predict software defects to mitigate the cost of fixing them.

Usually, during software development, the team learns about defects by testing results or Continuous Integration/Continuous Delivery (CI/CD) pipelines [D'Ambros et al., 2010]. However, these methods are not able to predict software defects [Menzies and Zimmermann, 2013, Tóth et al., 2016, Pornprasit et al., 2021]. Therefore, the team has to wait until the software system is deployed to the production environment to discover any

defects in the source code. This is a time-consuming process that can be avoided by predicting software defects. Additionally, the team may fix defects before the software system is deployed to the production environment [Zimmermann and Nagappan, 2008]. Hence, the software team may reduce the cost of fixing defects and improve the overall quality of the software. It is important to find methods to predict software defects to avoid the cost of fixing them [Shepperd et al., 2014]. For this reason, software defect prediction is relevant to both software engineering and the research community.

Although the current literature has an extensive amount of research studies about software defect prediction, there is a lack of understanding of the software features that contribute to the defectiveness of software classes [Tantithamthavorn et al., 2017, 2019, Jiarpakdee et al., 2020, Pornprasit et al., 2021]. The current literature lacks an understanding of why the machine learning model has predicted a target software class as defect-prone [Tantithamthavorn and Hassan, 2018]. While explaining the decisions made by the models is key to understanding the arrangement of defects and avoiding them, the software engineering literature has often neglected this topic for a considerable time [Ghotra et al., 2015, Agrawal and Menzies, 2018]. Furthermore, many investigations focus on the various software features extracted from source code that may lead software projects to a defective state [Nagappan et al., 2006, Moser et al., 2008, Nagappan et al., 2010, Wang et al., 2016a, Amasaki, 2018]. However, as code complexity increases in software development, the project may hinder characteristics that contribute to the defective state. For this reason, one of the greatest challenges faced by this research community is the identification of relevant software features (i.e., the effective features for defect prediction) and the irrelevant ones [Ghotra et al., 2015, Tantithamthavorn et al., 2015, Nam et al., 2018]. One technique to tackle this issue is data preparation aimed at identifying important software features from the code [D'Ambros et al., 2010, Ghotra et al., 2015]. Hence, the extensive literature about defect prediction has proposed features more specific to software attributes, such as class-level features [D'Ambros et al., 2010, Couto et al., 2012, Herbold, 2015], entropy features [Hassan, 2009, D'Ambros et al., 2010, Kaur et al., 2015], change features extracted from source code [Moser et al., 2008, D'Ambros et al., 2010, Kumar and Sureka, 2017, Rhmann et al., 2020], and established features to measure source code complexity [McCabe, 1976, Halstead, 1977, McCabe and Butler, 1989].

For these reasons, this thesis aims to fill the gap between not only assessing the performance of machine learning models for defect prediction but also understanding the reasons behind the software features that contribute to defective code. To do so, we employ historical data with existing defects and other features to predict the defects in the source code. In the end, we also focus on developers' perception of the software features as they are the primary beneficiary of the results reported in this thesis. As a result, we provide insights into the software defect prediction community that may support future explorations in this area.

1.2 General Goals

This thesis aims at providing a detailed investigation into understanding defects in software source code (i.e., classes or modules). In other words, our main goal is to understand machine learning models for defect prediction based on the software features. Hence, we employ a variety of software features to verify the understandability of machine learning models for defect prediction. These software features represent many aspects of the source code, including code size and complexity. To do so, we explore the feature space of machine learning models to investigate the impact of software features on the defectiveness of a particular class or module. Thus, the results may help practitioners to reason about their code quality. In addition, understanding these machine learning models may help project managers to make decisions about the software development process and identify defective-prone classes or modules. To explore these overarching objectives, we propose the following specific goals.

- *SG1* Investigate the datasets commonly applied in the current literature to predict software defects.
- *SG2* Find a machine learning model that can search the space of software features comparing the predictive accuracy of these models with baseline machine learning models.
- *SG3* Understand the software features that may generate defects in several software projects.
- *SG4* Evaluate developers' perceptions about the software features that contribute to their defective code.
- *SG5* Compare the similarities and redundancies between models for defect prediction and models for code smells detection.

SG1, investigating the datasets commonly applied in the current literature to predict software defects, is relevant because it helps to establish a baseline understanding of what data have been used in past studies to predict software defects and allows us to compare the results to previous research in the defect prediction literature. SG2, finding a machine learning model that can search the space of software features, is key to systematically explore the potential predictive power of different software features. In addition, we might identify the software features that are most effective in predicting defects. SG3, understanding the software features that may generate defects in several software projects, is relevant because it may help to identify potential causes of defects

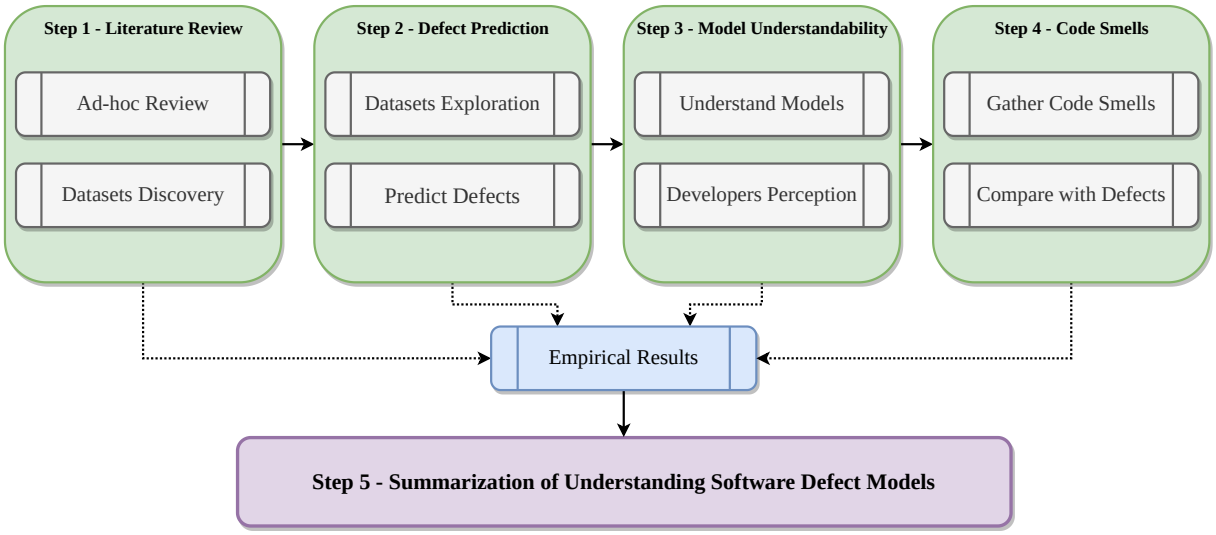
in software projects, which can inform strategies for preventing or addressing them. SG4, evaluating developers' perceptions about the software features that contribute to their defective code, is essential to provide valuable insights into how developers perceive the causes of defects in their work. Likewise, it might help to identify potential areas for improvement in their development processes. Finally, SG5, comparing the similarities and redundancies between defect and code smell models, is relevant because usually defect and code smells share equivalent software features, and a group of code smell models might be similar to defect models. Overall, each of these steps is important for advancing our understanding of software defects and for developing effective strategies for predicting and preventing them.

1.3 Research Method

We have divided this thesis into three main steps. Figure 1.1 provides an overview of these steps, along with the executed activities required to accomplish the objectives of each step. Therefore, this research project begins with an ad-hoc literature review of the existing literature on defect prediction and model understandability (Step 1 - Literature Review). This step is important to identify the machine learning models, evaluation metrics, and software features applied to understand software defect prediction (*Ad-hoc Review*). As a result, we have also classified the relevant research projects that explored defect prediction and understandability of machine learning models for defect prediction (first rectangle of the first box of Figure 1.1). More importantly, this step has resulted in the identification of the datasets [Jureczko and Spinellis, 2010, Jureczko and Madeyski, 2010, D'Ambros et al., 2010, Menzies et al., 2010] used in the literature to investigate defect prediction (*Datasets Discovery*). Hence, we have discovered that these datasets differ in many aspects, such as the software features they use to predict defects, the distribution of defects within the classes or modules, and how they define a software defect (second rectangle of the first box of Figure 1.1).

We then proceed to explore the datasets to predict software defects using machine learning techniques (Step 2 - Defect Prediction). In this step, we use the datasets to understand the distribution of software features that contribute to defects and how they are related to the occurrence of defects. By analyzing the datasets, we can gain insights into the characteristics of software systems that are most likely to cause defects and use this information to develop more effective strategies for predicting and preventing defects. The first activity of this step is to explore the datasets, clean the data, and identify the software features that contribute to the defects (*Datasets Exploration*). We concluded that

Figure 1.1: Overview of the Research Method Main Steps.



Source: Elaborated by the author.

the software features vary in a range of source code aspects [Moser et al., 2008, Hassan, 2009, D'Ambros et al., 2010, Couto et al., 2012, Herbold, 2015, Kaur et al., 2015, Kumar and Sureka, 2017, Rhmann et al., 2020]. For instance, we categorize the software features into class-level, entropy, change, and specific features (first rectangle of the second box of Figure 1.1). We then apply machine learning techniques to predict software defects using the identified software features (*Predict Defects*). As a result, we investigate a machine learning model to predict the software class that is defective-prone (second rectangle of the second box of Figure 1.1). To do so, we focus on feature-space exploration and propose an algorithm from the analysis. In the end, we compare the predictive accuracy with several machine learning models identified in the ad-hoc literature review. In a popular study, Hall et al. [2012] already conducted a Systematic Literature Review (SLR) to investigate the machine learning models applied to predict software defects. However, this study did not consider the understandability of the machine learning models, which is the main focus of this thesis.

In the third step of the investigation, we execute two studies to reason about software defects (Step 3 - Model Understandability). For the first study, we employ a recent machine learning technique known as SHAP (Shapley Additive exPlanations) [Lundberg et al., 2018a] to understand the predictions made by machine learning models (*Understand Models*). We then reason about the understandability of these machine learning models and how they impact software defects (first rectangle of the third box of Figure 1.1). Finally, we conduct a survey with developers from different backgrounds to understand how they perceive the influence of several software features (*Developers Perception*) on defects in the source code (second rectangle of the third box of Figure 1.1). In the concluding step of the research project, we compare defect and code smell models to understand the similarities and redundancies between them (Step 4 - Code

Smells). To do so, we gather a list of class-level code smells (*Gather Code Smells*) and then compare the generated models (*Compare with Defects*). The final step (Step 5 - Summarization of Understanding Software Defect Models) is the summarization of the results of the understandability of defect models focused on software features.

1.4 Main Contributions

We expect the main contribution of this thesis project is the understandability of machine learning models for defect prediction based on the software features. Moreover, we evaluate how each software feature contributes to the defectiveness of the software class or module. In this case, we focused on the limit of selected software features for defect prediction. We believe our empirical results may assist both developers and project managers in understanding the software features that contribute to software defects. Consequently, we may help developers to enhance their software quality and ability to reason about different aspects of their code. Besides, the defect prediction community may rely on our empirical investigations to improve current efforts in the defect prediction field. For instance, the applied machine learning model can serve as a benchmark for future explorations to detect defects. By this point, the research reported in this thesis produced the following publications [Mariano et al., 2019, dos Santos et al., 2020a, dos Santos and Figueiredo, 2020b,a, dos Santos et al., 2020b, Mariano et al., 2020, dos Santos et al., 2022a,b, 2023]. Only one of the publications listed below (the last one) is under review at an international journal.

1. dos Santos, G. E., Figueiredo, E., Veloso, A., Viggiato, M., and Ziviani, N. (2020). Understanding machine learning software defect predictions. Automated Software Engineering Journal (ASEJ).

2. dos Santos, G. E., Santana, A., Vale, G., Figueiredo, E. (2022). Yet Another Model! A Study on Model's Similarities for Defect and Code Smells. 26th International Conference on Fundamental Approaches to Software Engineering (FASE), Paris, France.

3. dos Santos, G. E. and Figueiredo, E. (2020). Failure of one, fall of many: An exploratory study of software features for defect prediction. In Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), Adelaide, Australia.

4. dos Santos, G. E., Figueiredo, E., Veloso, A., Viggiato, M., and Ziviani, N. (2020). Predicting software defects with explainable machine learning. In Proceedings of the 19th Brazilian Symposium on Software Quality (SBQS), São Luís, Brazil.

5. dos Santos, G. E. and Figueiredo, E. (2020). Commit classification using natural language processing: Experiments over labeled datasets. In Proceedings of the XXIII Iberoamerican Conference on Software Engineering (CIbSE), Curitiba, Paraná, Brazil.

6. dos Santos, G. E., Veloso, A., Figueiredo, E. (2022) Understanding thresholds of software features for defect prediction. 36th Brazilian Symposium on Software Engineering (SBES), Uberlândia, Brazil.

7. dos Santos, G. E., Veloso, A., Figueiredo, E. (2022). The Subtle Art of Digging for Defects: Analyzing Features for Defect Prediction in Java Projects. International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Online Streaming.

8. Mariano, R., dos Santos, G. E., Viggiato, M., and Brandão, W. (2019). Feature changes in source code for commit classification into maintenance activities. In Proceedings of the 18th International Conference on Machine Learning and Applications (ICMLA), Boca Raton, USA.

9. Mariano, R., dos Santos, G. E., and Brandão, W. (2021). Improve classification of commits into maintenance activities with quantitative changes in source code. In Proceedings of the 2021 International Conference on Enterprise Information Systems (ICEIS), Online Streaming.

10. dos Santos, G. E., Muzetti, I., Figueiredo, E. (2022). Two Sides of the Same Coin: A Study on Developers' Perception of Defects. In review at the Journal of Software: Evolution and Process (JSEP).

1.5 Empirical Results

This section summarizes the main empirical results of this thesis, as we discuss next.

SG1 We identified three datasets from an ad-hoc literature review conducted in the early stages of this project. These datasets vary in size, software features, and

how they implement the concept of a software defect. The datasets comprise three relevant sources for researchers, and they illustrate the importance of data quality and reproducibility for defect prediction research. Despite most software features being unique to each dataset and software project, the datasets share the imbalanced nature commonly encountered in the current literature. In this case, datasets for defect prediction usually have more non-defective instances (i.e., classes or modules) than defective ones. As a result, we observed that software features relate to various aspects of the source code. For instance, we identified five categories of software features: (i) class-level features, (ii) entropy features, (iii) change features, (iv) McCabe and Halstead features, and (v) additional features not correlated to the remaining features. Most software features relate to object-oriented design (i.e., either CK features or other features that measure code complexity and size).

SG2 In the first empirical study, we employed baseline machine learning models to predict software defects and evaluated the effectiveness of these models on the target datasets. Furthermore, we compared these baseline models with the implementation of gradient boosting. The proposed implementation conveys an exploratory examination that produced hundreds of thousands of random machine learning models from a diverse collection of software features. These machine learning models are random because they promptly select features from the entire pool of software features available for defect prediction. Finally, we analyzed the predictive power of the machine learning models using the target datasets. This investigation reveals how hard it is to detect defects, as only a small fraction of the models (1.8%) achieved a performance higher than 83% based on the AUC numbers. We hope that our efforts can become a baseline for other solutions to defect prediction using Java projects. We also conclude that a limited set of features produced high accuracy numbers.

SG3 In the second empirical investigation, we used a technique known as SHAP (Shapley Additive exPlanations) to explain the machine learning models [Lundberg and Lee, 2017, Lundberg et al., 2018a, Jesus et al., 2021]. Therefore, we can reason about the model decision and how the target software features influence these decisions (i.e., predicting whether a software class is defective or not). We conclude that the best-performing machine learning models are easy to understand, as they employ fewer features from the power-set. Furthermore, the results indicate how difficult it is to generate a unique solution to understand defect models. Independent projects are subject to distinct software features that may cause software defects. The SHAP explanations also suggest that a variety of software features (e.g., Lines of code, Age of a class in weeks, Number of attributes, Average method complexity, among others) tend to lead to a higher probability of defects if the software feature value is high. To confirm that high values tend to indicate defective code [Lundberg

et al., 2018b], we investigated to determine the threshold of software features for defect prediction. We conclude that higher values usually help the model predict software defects. For example, Weighted Method per Class (WMC) higher than 35 consistently increases the probability of predicting a defect in the class.

SG4 In this empirical study, we conducted two survey studies with developers. First, we examined the developers' understanding of SHAP values [Lundberg et al., 2020]. We concluded that SHAP explanations are valuable for developers in two ways. First, developers could understand and reason about the most impactful software features with their knowledge of software development. Second, we questioned developers about how they perceived a list of software features and their relationship to defective code. We concluded that developers' perceptions differ from the machine learning models. We noted that developers classify software feature complexity as the main quality attribute contributing to defects, while the machine learning models classified documentation as the primary quality attribute contributing to defects. This is an interesting result as it contradicts common sense about the quality attributes and their impact on defects. More importantly, the applied technique identifies software features that developers could act upon in the source code, although it is difficult to evaluate to what extent it affects software development.

SG5 In our final empirical investigation, we examine the connection between defects and code smell models. To do this, we employ a tool called Organic to identify and validate code smells in our data. We then clean and select important features for our prediction models. Later, we train and evaluate the models using an ensemble of the models. In the end, as the models present good performance measures, we employ an explainability technique known as SHAP to understand the models' decisions. Our results show that out of the seven code smells we initially identified, only three (Refused Bequest, God Class, and Spaghetti Code) present similar models to the defect model. We also report that the features Nesting Level Else-If and Comment Density are important for all four models. We observe that most features need high values to predict defects and code smells, except for Refused Bequest. Finally, we conclude that documentation, complexity, and size quality attributes are the most important for these models.

1.6 Thesis Structure

This section presents this thesis' structure. We distribute the remainder of this thesis into seven chapters.

Chapter 2 presents the relevant studies that guide this thesis, focusing on relevant machine learning and software engineering definitions. We briefly discuss the datasets used to predict defects, the features applied to evaluate the defect models, and how the current literature addresses defect prediction.

Chapter 3 discusses in-depth the datasets used to predict defects in the current literature, focusing on the software features and defect distribution. As a result, we concentrate on three datasets that vary in their characteristics of the source code.

Chapter 4 examines the results of predicting software defects with the target datasets and compares them with baseline algorithms commonly applied in the literature. Finally, we discuss the predictive power of the machine learning algorithms employed to predict defects.

Chapter 5 investigates the understandability of machine learning models to predict software defects using SHAP. In doing so, we reason about the software features that may help developers to understand the defects in their code. The investigation focuses on several software projects.

Chapter 6 discusses the results of two survey investigations with software developers about the machine learning models' results. To do so, we compare both the SHAP explanations and developers' perception about the software features that may cause defects in their code. The main goal is to understand the developers' perception of the machine learning models.

Chapter 7 investigates the results of the empirical evaluation that compares defect and code smells models. For this purpose, we use class-level software features to compare them. The main goal is to understand the similarities and redundancies between these models.

Chapter 8 presents a summarization of this thesis. We then discuss the current status of the thesis project. Hence, we present the next steps we aim to perform to complete the thesis project. Finally, we provide insights about the expected schedule.

Chapter 2

Background and Related Work

Most software systems must evolve to cope with stakeholders' requirements and fix existing defects. Therefore, the reliability of these systems has become a key concern for the development team [Zimmermann and Nagappan, 2008, Nagappan et al., 2010, Kamei et al., 2013, Jing et al., 2014, Wang et al., 2016a, Zhang et al., 2017, Tantithamthavorn et al., 2019, Jiarpakdee et al., 2020]. To assist developers in finding defects, software defect prediction is one of the research directions that applies machine learning techniques to software engineering. Moreover, predicting software defects is an area of interest in software engineering as it helps development teams maintain significant levels of software quality [Turhan et al., 2009, Ghotra et al., 2015, Pornprasit et al., 2021]. Hence, machine learning models have become increasingly popular for software defect prediction and have demonstrated their effectiveness in many scenarios throughout the last decade [Nagappan and Ball, 2005, Nagappan et al., 2006, Menzies et al., 2007, D'Ambros et al., 2010, Nagappan et al., 2010, Menzies et al., 2010, Menzies and Zimmermann, 2013, Madeyski and Jureczko, 2015, Tantithamthavorn et al., 2019, Jiarpakdee et al., 2020]. As a result, the current literature employs a vast number of machine learning techniques for software defect prediction. This chapter presents the background and related work associated with defect prediction in software projects. We discuss the challenges of predicting software defects and how the current literature applies recent techniques to address these issues. It is important to note that the literature presented in this chapter was not gathered using a Systematic Literature Review (SLR) approach. Instead, we used an ad-hoc approach to identify the most relevant papers in the field of software defect prediction.

In this chapter, we aim to provide a comprehensive overview of software defects and the techniques used to predict them in the software development process. Thus, we divide the remainder of this chapter into six sections. Section 2.1 provides an overview of software defects. We contextualize software defects and how the defect prediction field can help software developers. In addition, we briefly discuss the types of software features one may use to predict software defects in their source code. Section 2.2 briefly discusses class-level code smells and anti-patterns. Section 2.3 introduces the quality attributes that organize the software features in the literature. Section 2.4 examines the machine learning algorithms used to predict defects in the literature. Therefore, we present commonly used

evaluation metrics to test model performance. In the next chapter, we discuss the features that each dataset has to predict defects in multiple projects. Then, Section 2.5 explains the background of predicting defects with source code features such as Lines of Code (LOC) or Weighted Methods per Class (WMC). Additionally, we present studies related to predicting software defects from metadata information. Moreover, we investigate how the current literature explains software defects from these features. Finally, Section 2.6 concludes the chapter with insights into the background and related work.

2.1 Software Defects

The current literature employs different definitions for software defects. In this thesis, we utilize the IEEE definition [IEEE, 1990]. The term "defect" is usually referred to as a fault, error, failure, or bug [IEEE, 1990, Valente, 2020]. A software defect may harm the appearance, operation, functionality, or performance of the target software project [Haskins et al., 2004, Herzig and Zeller, 2013]. Software defects may appear in various stages of software development [Kaur and Sharma, 2019], interrupt the development progress, and increase the planned budget of software projects [Menzies et al., 2010]. Furthermore, a software team may discover software defects after code release, which generates significantly more effort to tackle these defects in production [Levin and Y., 2017]. In addition, we can divide the Software Development Life Cycle (SDLC) into five phases: (i) Analysis, (ii) Design, (iii) Implementation, (iv) Testing, and (v) Maintenance [Bassil, 2012]. As a result, if a company notices more software defects than expected in the Testing or Maintenance phases, it may impair its ability to meet customer deadlines or milestones [Turhan et al., 2009]. Therefore, it is challenging to upgrade a software system that is already defective, as the software team continues to spend development time fixing the existing code instead of developing new features for their customers [Kaur and Sharma, 2019].

To mitigate these issues with software development, defect prediction is a method for predicting defects in a software project [Menzies et al., 2007, 2010]. Thus, the use of defect predictors is valuable for anticipating defects in the source code. For instance, if a software team has limited resources for software inspection, a defect predictor may indicate which modules are most likely to be defective. To generate a machine learning model able to predict software defects, it is necessary to collect the data and build statistical models based on the dataset [Burkov, 2019]. First, we need a dataset with instances of known labels (i.e., defective or clean) [Turhan and Bener, 2009]. Usually, the labels are extracted from bug reports, commits that fix defects, and other tools. We discuss in detail how

several data sources employ the concept of defects in Chapter 3. Since we are interested in identifying defective instances, it is generally ideal to have a dataset with only binary labels. Later, we employ feature extraction to the entire data source extracting the most relevant features from the existing ones. Next, the complete pool of software features with the corresponding label (i.e., defective or clean) serves as the input of the machine learning classifier[1]. Finally, the model can classify unseen software instances (i.e., classes, files, or modules) into defective or clean. More specifically, a group of datasets [Ghotra et al., 2015] defines a defect based on the following expression, where one or more errors change the status of a module to defective.

$$defective? = errorcount >= 1$$

The input of a defect prediction model is usually a set of software features. A software feature is a measure of a specific software property. For example, a software feature may represent the total number of lines of code (LOC) in a class. Therefore, we utilize a set of software features to describe different aspects of software artifacts such as a file, a class, or a module. As a result, several software features are easy to compute, and others are more complex to represent in the source code. For instance, it is easy to calculate a software feature such as LOC. On the other hand, the WMC (Weighted Method per Class) is more complex to compute and express in a software artifact. Furthermore, these software features can express numerical values (integer or real) or represent a linearly ordered range of values known as continuous features [Fayyad and Irani, 1993]. However, the use of continuous software features may interfere with the predictive capacity of the model. As an example, the Lack of Cohesion in Methods (LCOM) is a continuous feature that may assume any value in the feature space [Jureczko and Madeyski, 2010, Ferenc et al., 2020b]. For this reason, we need to define the feature space of the model by discretizing the continuous software features [Ma et al., 2014]. Finally, as most studies apply a classification learning method [Elish and Elish, 2008], it focuses on learning a model that classifies new instances into one of the two labels (i.e., defective or clean) [Jing et al., 2014]. We cover in detail the entire list of software features we employ to explore the defect prediction field in the next chapter of this thesis (Chapter 3).

[1]In this thesis, the terms “classifier” and “model” are used interchangeably to refer to a statistical or machine learning algorithm that is used to predict a certain outcome or target based on input data. Both terms refer to the same concept.

2.2 Code Smells

Brown et al. [1998] proposed a catalog of anti-patterns, which are solutions to recurring problems based on design patterns. However, instead of providing reusable code, anti-patterns have a negative impact on the source code. A code smell (short for "bad smells" or "smells") is a symptom of poor design and implementation choices [Tufano et al., 2015]. Therefore, a code smell is a surface indication that usually corresponds to a deeper problem in the system [Fowler, 1999]. By definition, a code smell is something quick to spot (i.e., "sniffable"). For example, a long method may represent a code smell because a developer may quickly realize that the method has too many lines simply by looking at the code [Fowler, 1999]. However, code smells do not always indicate a problem within the source code. Some long methods are acceptable [Fowler, 1999]. As a result, code smells are often an indicator of a problem rather than the problem itself [Fowler, 1999]. Additionally, code smells are not defects, but they are indicators of potential problems in the source code [Khomh et al., 2009, 2012, Fontana et al., 2013, Sjøberg et al., 2013, Yamashita and Counsell, 2013, Yamashita and Moonen, 2013, Palomba et al., 2014, Cunha et al., 2020]. Since this thesis focuses on class-level datasets, only problems related to classes are considered. Table 2.1 presents a comprehensive list of class-level code smells and anti-patterns. The first column shows the code smell name, while the second column provides a brief definition of the smell or anti-pattern. Finally, the last column presents the reference that discusses it in more detail.

Table 2.1: Code Smell and Anti-Pattern Definitions.

Problem	Definition	Proponent
God Class (GC)	A large class that have too many responsibilities and centralizes the module functionality.	Riel [1996]
Refused Bequest (RB)	A class that does not want to use its parent behavior.	Fowler [1999]
Spaghetti Code (SC)	A class that has methods with large and unique multistage process flow.	Brown et al. [1998]
Class Data Should be Private (CP)	A class with too many public fields.	Shvets [2021]
Data Class (DC)	Classes that have only fields, getters and setters.	Fowler [1999]
Lazy Class (LC)	Classes that have little behavior, with few methods and fields.	Fowler [1999]
Speculative Generality (SG)	Classes that support future behavior, usually interacting with test classes only.	Fowler [1999]

Source: Elaborated by the author.

2.3 Quality Attributes

A software feature may be related to a quality attribute [Ferenc et al., 2018, 2020a]. Although the current literature proposes different quality attributes to group similar software features [Aghajani et al., 2020, Basili et al., 1996, Stroulia and Kapoor, 2001], we focus on the quality attributes previously discussed in one of the main datasets used for defect prediction [Ferenc et al., 2018, 2020a]. These quality attributes cluster a wide range of software features. Therefore, we consider seven quality attributes to group the entire collection of software features: (i) *Complexity* [Basili et al., 1996, Stroulia and Kapoor, 2001, Ferenc et al., 2020a], (ii) *Coupling* [Stroulia and Kapoor, 2001, Abdullah AlOmar et al., 2019, Ferenc et al., 2020a], (iii) *Size* [Fowler, 1999, Wang et al., 2016a, Ferenc et al., 2020a], (iv) *Documentation* [Fowler, 1999, Aghajani et al., 2020, Ferenc et al., 2020a], (v) *Clone* [Tóth et al., 2016, Ferenc et al., 2018], (vi) *Inheritance* [Fowler, 1999, Aghajani et al., 2020, Ferenc et al., 2020a], and (vii) *Cohesion* [Tóth et al., 2016, Ferenc et al., 2020a]. Table 2.2 presents the quality attributes with their definition and reference. The first column shows the name of the quality attribute. The second column is the definition of the quality attribute, and the third column represents the literature reference. For instance, inheritance measures the different aspects of the inheritance hierarchy of the project [Fowler, 1999, Aghajani et al., 2020, Ferenc et al., 2020a].

Table 2.2: Quality Attributes Definition.

Name	Definition	Proponent
Clone	Measure code cloning, we identify instances of copy-pasting existing source code or making minor modifications to the original code.	Tóth et al. [2016], Ferenc et al. [2018, 2020a]
Cohesion	Measure to what extent the source code elements are coherent in the system.	Tóth et al. [2016], Ferenc et al. [2020a]
Complexity	Measure the complexity of source code elements (typically algorithms).	Basili et al. [1996], Stroulia and Kapoor [2001]
Coupling	Measure the amount of dependencies of source code elements.	Stroulia and Kapoor [2001], Abdullah AlOmar et al. [2019]
Documentation	Measure the amount of comments and documentation of source code elements in the system.	Fowler [1999], Aghajani et al. [2020]
Inheritance	Measure the different aspects of the inheritance hierarchy of the system.	Fowler [1999], Aghajani et al. [2020]
Size	Measure the system's properties in terms of different aspects (e.g., number of code lines, number of classes, or methods).	Fowler [1999], Wang et al. [2016a]

Source: Elaborated by the author.

2.4 Machine Learning and Software Engineering

Machine learning refers to the process of solving practical problems by gathering a dataset and algorithmically building a statistical model based on that data [Burkov, 2019]. To do so, data collection relies on examples of some phenomenon, which can come from nature, be handcrafted by humans, or be generated by another algorithm [Burkov, 2019]. In the software engineering field, researchers use machine learning techniques to create cognitive services for vision and speech, conversational agents for human interaction, real-time translation of text and voice, and many more applications [Amershi et al., 2019]. One of the uses of machine learning in software applications is to detect defects in software projects [Bowes et al., 2018]. In fact, predicting software defects is a task that researchers have employed distinct machine learning models for. To compile these models, we conducted an ad-hoc literature review, considering popular digital libraries for the research community such as ACM, IEEE, Springer, Google Scholar, among other resources. Hence, we noticed that researchers have implemented eight algorithms to predict defects in source code. Some of the machine learning models identified in previous works include Logistic Regression, Naive Bayes, K-Nearest Neighbor, Neural Networks, Decision Trees, Support Vector Machine, Random Forest, and Gradient Boosting Machine. Next, we present a detailed explanation of each algorithm utilized in the defect prediction literature.

Logistic Regression: This algorithm represents a statistical method used for classification in a dataset. It is commonly employed when there are one or more independent variables that determine the outcome. The classifier defines the value of one of two potential outcomes.

Naive Bayes: This method performs classification based on Bayes' rule. Here, the algorithm finds the conditional probability of an instance holding a specific label from the set of possible labels. Thus, the algorithm picks the label with the highest probability as the chosen classification.

K-Nearest Neighbors: This algorithm represents a non-parametric decision that classifies an unknown instance based on the nearest neighbor. The algorithm employs a distance function to determine the distance between the unknown and the known instances. Therefore, the method selects the k-nearest neighbors of the unknown instance and utilizes the majority vote to decide the approximate classification.

Neural Networks: This method consists of layers of units known as neurons. The layers are typically named the input layer, hidden layer, and output layer. Multiple

hidden layers may exist between the input and output layers. The algorithm uses the weights of the connections between the layers to predict the output.

Support Vector Machine: This algorithm uses a set of labeled data considering two possible labels to classify. Then, the method builds a model mapping the data as points in a space so that the generated mapping divides the two separate labels by a clear gap as wide as possible. As a result, the model can map unknown data into a mapped space. Hence, the method determines the label prediction of the unknown instance based on the side of the gap it is located on.

Decision Trees: This method is a type of flowchart that shows a clear pathway to a decision. Therefore, it is a type of algorithm that includes conditional statements to classify the data. Consequently, a decision tree starts at a single point and then branches in two or more directions. Each branch proposes different outcomes, incorporating several decisions and chance events until the algorithm achieves a definite outcome.

Random Forest: This algorithm consists of a collection of tree predictors, each of which is employed to classify an unknown instance. Therefore, the learning method considers the number of trees, node size, and the number of features sampled. Finally, the classification applies the majority result of the trees' predictions to decide the correct label.

Gradient Boosting Machine: This method can improve the prediction accuracy of machine learning classifiers by combining a set of weak classifiers to create a robust classifier. The algorithm is a combination of ensemble learning and boosting techniques. Popular implementations of this algorithm include XGBoost and LightGBM.

Table 2.3 presents a list of relevant investigations that have implemented each of the mentioned classifiers for defect prediction, along with their respective literature references. The list is sorted alphabetically. The first column of the table represents the literature reference to facilitate access to the source. The remaining columns (two to nine) list the eight algorithms that were applied for the defect prediction task. By analyzing the references, we conclude that the Random Forest algorithm is a popular option in current literature, followed by the Decision Trees and Naive Bayes algorithms. Another important option is the Logistic Regression algorithm. The least applied algorithm is the Gradient Boosting Machine. We believe that the reason for the limited application of the Gradient Boosting Machine is the fact that it is the most recent algorithm among those discussed in the literature [Chen and Guestrin, 2016, Ke et al., 2017].

Table 2.3: Baseline Machine Learning Classifiers Employed in Defect Prediction.

Proponent	LR	NB	KNN	NN	SVM	DT	RF	GB
Elish and Elish [2008]	•	•	•	•		•	•	
Ferenc et al. [2020a]						•		
Fukushima et al. [2014]							•	
Gray et al. [2011]					•			
Jiang et al. [2013]	•	•				•		
Jing et al. [2014]		•			•	•		
Kamei et al. [2013]	•							
Kaur and Sharma [2019]				•				
Knab et al. [2006]						•		
Nagappan et al. [2006]	•							
Pascarella et al. [2020]					•	•	•	
Pornprasit et al. [2021]	•						•	•
Shuai et al. [2013]					•			
Sun et al. [2012]		•				•	•	
Tantithamthavorn et al. [2015]							•	
Tantithamthavorn et al. [2017]	•	•					•	
Tantithamthavorn and Hassan [2018]	•						•	
Turhan et al. [2009]		•				•	•	
Wang and Li [2010]		•						
Xuan et al. [2015]	•	•	•		•		•	
Yang et al. [2016]	•		•			•	•	
Yatish et al. [2019]							•	•
Zhongbin et al. [2018]		•				•	•	

Source: Elaborated by the author.

***LR**: Logistic Regression; **NB**: Naive Bayes; **KNN**: K-Nearest Neighbors; **NN**: Neural Networks; **SVM**: Support Vector Machine; **DT**: Decision Trees; **RF**: Random Forest; **GB**: Gradient Boosting Machines.*

Additionally, we compiled a list of the data and programming languages used in the literature. Table 2.4 presents the literature reference in the first column (which corresponds to Table 2.3). The second column shows the data availability. We note that the majority of the data is not available to the community because it is proprietary. For example, the study by Nagappan et al. [2006] focuses on Microsoft data that is not available to the community. However, a significant number of papers have open data, although a substantial amount applies to the NASA datasets [Wang and Li, 2010, Gray et al., 2011, Jing et al., 2014]. Other literature references only had a portion of the data available for the community. In terms of programming languages used in the literature, we found that the majority of the papers focused on Java, with several papers exclusively using that language. Other popular languages included C and C++. Some studies did not clearly state the programming languages their data was based on. For data availability, we only considered data to be open if it was directly cited in the paper. We did not search the internet for data. Furthermore, we only considered programming languages explicitly

stated in the paper and did not search for source code to identify the language. Finally, the last column displays the type of classification (i.e., binary or multiclass). We note that most papers focus on binary classification. However, there are several papers that focus on multiclass classification [Nagappan et al., 2006, Sun et al., 2012, Kamei et al., 2013, Kaur and Sharma, 2019].

Table 2.4: Details about the Defect Prediction Literature.

Proponent	Data	Language	Binary/Multiclass
Elish and Elish [2008]	Open	C++, Java	Binary
Ferenc et al. [2020a]	Open	Java	Binary
Fukushima et al. [2014]	Closed	C, C++, Perl, Ruby	Binary
Gray et al. [2011]	Open	C, C++, Java, Perl	Binary
Jiang et al. [2013]	Closed	C, Java	Binary
Jing et al. [2014]	Open	C, C++, Java	Binary
Kamei et al. [2013]	Closed	C, C++, Java	Multiclass
Kaur and Sharma [2019]	Open	Java	Multiclass
Knab et al. [2006]	Closed	Not Mentioned	Binary
Nagappan et al. [2006]	Closed	C++, C#	Multiclass
Pascarella et al. [2020]	Closed	Java	Binary
Pornprasit et al. [2021]	Closed	Not Mentioned	Binary
Shuai et al. [2013]	Open	C, C++	Binary
Sun et al. [2012]	Open	C, C++, Java, Perl	Multiclass
Tantithamthavorn et al. [2015]	Partially	Not Mentioned	Binary
Tantithamthavorn et al. [2017]	Partially	Java	Binary
Tantithamthavorn and Hassan [2018]	Closed	Not Mentioned	Binary
Turhan et al. [2009]	Partially	C, C++, Java	Binary
Wang and Li [2010]	Open	C, C++, Java, Perl	Binary
Xuan et al. [2015]	Open	Java	Binary
Yang et al. [2016]	Closed	Java	Binary
Yatish et al. [2019]	Closed	Java	Binary
Zhongbin et al. [2018]	Open	Java	Binary

Source: Elaborated by the author.

Evaluation Metrics. To evaluate the performance of a defect prediction model, we typically analyze four possible outcomes [Zimmermann et al., 2007, Peters et al., 2013, Fukushima et al., 2014, Pascarella et al., 2020], which represent whether the instance (e.g., file, class, module, or method) is defective or clean (i.e., also referred to as non-defective). Table 2.5 describes these possible outcomes of the defect prediction model and their respective descriptions. The first two columns present the predicted outcomes (i.e., positive or negative), and the last two columns show the actual outcomes (i.e., positive or negative). Thus, the confusion matrix is a table that illustrates the number of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). Below Table 2.5, we explain the meaning of each outcome for the defect prediction task.

Table 2.5: Confusion Matrix for Defect Prediction Outcomes.

		Actual	
		Positive	Negative
Predicted	Positive	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

True Positive (TP) → A defective entity is classified as defective
False Negative (FN) → A defective entity is classified as clean
True Negative (TN) → A clean entity is classified as clean
False Positive (FP) → A clean entity is classified as defective

Source: Elaborated by the author.

Thus, the confusion matrix represents a valuable asset for evaluating the performance of a defect prediction model. However, the most comprehensive performance measurement of a classification model is a Receiver Operating Characteristic Curve (ROC) [Kuhn and Johnson, 2013]. To calculate the ROC, it is necessary to examine some aspects of the performance. For this reason, we will briefly introduce how to calculate ROC. To do so, we will present two important evaluation metrics: recall (i.e., sensitivity) and specificity.

– Recall (Sensitivity) → This metric represents how many existing defective instances are detected by the model. Recall is calculated as the percentage of true positives (tp) over the total number of actual positives (tp + fn). This measurement is also known as the true positive rate of the model's predictions [Kuhn and Johnson, 2013].

$$recall : \frac{tp}{tp + fn}$$

– Specificity → This metric represents the proportion of actual clean instances that are predicted as defective. Specificity is calculated as the number of true negatives (tn) over the total number of actual negatives (tn + fp). This metric is also called the true negative rate of the model's predictions [Kuhn and Johnson, 2013].

$$specificity : \frac{tn}{tn + fp}$$

If we plot sensitivity (recall) versus specificity of a prediction model, it creates a ROC curve. ROC improves the model's performance by maximizing sensitivity and specificity, thus maximizing the Area Under the Curve (AUC) calculation. The maximum value for AUC is 1, which indicates the model's highest performance. In the defect prediction literature, an acceptable threshold for AUC in both binary or multi-class classification is around 69%, as previously reported [Tong et al., 2018]. AUC is a strong indicator of model performance, but two additional evaluation metrics can help investigate model performance in defect prediction.

– Precision → This metric represents the proportion of correct predictions of defective instances. Precision is calculated as the percentage of true positives (tp) over the total number of predicted positives (tp + fp) [Kuhn and Johnson, 2013].

$$precision : \frac{tp}{tp + fp}$$

– F-Score → This metric represents the harmonic mean of precision and recall. The F1-Score is also referred to as F1 or F-Measure. The F1-Score is calculated as the weighted average of precision and recall multiplied by two [Kuhn and Johnson, 2013].

$$f1 : 2 * \frac{precision * recall}{precision + recall}$$

2.5 Related Work

This section discusses relevant research that applies machine learning to predict software defects. First, we present related work on datasets used for defect prediction. Second, we discuss studies that effectively use source code features to predict defects in source code. Then, we briefly discuss the use of metadata information to predict defects. Finally, we present recent studies on applying machine learning model interpretability to defect prediction.

Datasets for Defect Prediction. This section discusses the main studies involved in four of the most commonly used datasets for defect prediction. In the early stages of the defect prediction literature, the lack of data sources was a significant challenge for the community [Kamei and Shihab, 2016]. To address this, NASA decided to publish data on the NASA Metric Data Program for the first time. The dataset[2] gained considerable popularity among the defect prediction community. The dataset [Menzies et al., 2007, 2010] is based on features from Halstead's operator-operand counts [Halstead, 1977] and McCabe's dependencies and complexity [McCabe, 1976, McCabe and Butler, 1989]. The main reason for its popularity is the fact that the dataset was available for free, and the data does not require much additional processing to predict defects with moderate performance [Hall et al., 2012]. Despite the dataset's popularity, several studies have pointed out that the data quality is not optimal, and the dataset requires a data cleaning process to improve the classifiers' overall performance [Ghotra et al., 2015]. Furthermore,

[2] https://www.nasa.gov/sites/default/files/files/nasa_metrics_data_program_data_set_v1.0.xlsx

the current literature recognizes that the choice of the classifier is essential since the data is not standardized and is noisy. However, the dataset is relevant to the defect prediction community because it confirms how important data quality is for predicting defects.

Due to the expansion of data sources and algorithms for defect prediction over the years, researchers have had the opportunity to explore new approaches to defect prediction. This transformation originated in the study of the use of software features for Object-Oriented Programming (OOP) [Chidamber and Kemerer, 1994]. Chidamber and Kemerer [1994] proposed a novel set of software features based on Object-Oriented design patterns. Later, the research community named these features CK features after the authors' names. They relate these features to measurements that reflect the viewpoint of experienced Object-Oriented programmers. After extensive experimentation with the novel features, the authors found that the features are beneficial for reducing the cost of software development, testing, and maintenance, and improving the overall software quality. They concluded that higher values of individual software features are associated with higher design effort and lower productivity of the software team to fix potential defects [Chidamber and Kemerer, 1994]. Other works investigated the benefits of the software features proposed by Chidamber and Kemerer [1994]. In Basili et al. [1996], the authors drew connections between the CK features and the defectiveness of a module. They experimented with the set of features and concluded that some software features could generate satisfactory predictors using the C++ programming language.

A couple of years later, Jureczko and Spinellis [2010] gathered an impressive collection of data based on the CK features. The data is publicly available under the PROMISE repositories[3] [Jureczko and Madeyski, 2010, Jureczko and Spinellis, 2010]. The authors discussed the inefficiency of current methods of extracting CK features from existing software projects [Jureczko and Spinellis, 2010]. To collect the data, the authors used a tool known as *ckjm* to calculate each feature for each project. Additionally, they employed a tool called *BugInfo* to identify the defects. *BugInfo* analyzed the logs of each target repository (using a Version Control System). Therefore, the tool could extract the content and determine if a commit was a fix to a defective state. To do so, the authors identified the comments that indicate defect fixes and proposed a regular expression to identify the commits that were intended to fix defects. Then, the tool compared the regular expression with the commit comments. In case the comment fit the regular expression, *BugInfo* incremented the defect count for all classes the commit modified. The authors concentrated on the Java programming language. After experimenting with the tool and data source, the authors documented that two CK features are class size factors: Weighted Methods per Class (WMC) and Lines of Code (LOC). Finally, the authors demonstrated the effectiveness of the data for defect prediction using simple regression models [Jureczko and Spinellis, 2010].

[3] http://promise.site.uottawa.ca/SERepository/

In another study over the same dataset, the works of Jureczko and Madeyski [2010] described an analysis of 92 releases of 38 proprietary, open-source and academic projects. The authors used software features to generate clusters that could join projects that have similar defect causes. They experimented with the clusters with a state-of-art technique for defect prediction based on hierarchical and k-means clustering. They found at least six groups in the dataset, but statistically, only two demonstrated to be true. The authors prove the existence of those two clusters with statistical tests. The comparison confirmed that the clusters did not have significant inconsistencies with other studies about defect prediction, and the overlap is consistent with the current literature. They conclude that the results were not astonishing enough to justify using the clusters to detect defects [Jureczko and Madeyski, 2010] in future explorations. Even so, the data sources gathered are relevant as the authors expanded the previous work [Jureczko and Spinellis, 2010]. For this reason, the authors strongly encourage the use of the data to improve defect prediction using CK features [Jureczko and Madeyski, 2010, Jureczko and Spinellis, 2010].

Similarly, another approach to defect prediction data sources is the Bug Prediction dataset[4] [D'Ambros et al., 2010]. In this data source, the authors present a publicly available dataset for the defect prediction community. The data source consists of several software systems related to the Java Programming Language, including the popular Eclipse IDE. Unlike previous works [Jureczko and Madeyski, 2010, Jureczko and Spinellis, 2010], the dataset does not focus entirely on CK features. Instead, the authors propose novel features about other aspects of the source code [D'Ambros et al., 2010]. The authors built the data based on several approaches: (i) changes cause defects in the source code [Moser et al., 2008]; (ii) previous defects may predict future defects [Kim et al., 2007]; (iii) complex components are harder to change, and hence defect-prone [Basili et al., 1996]; and finally, (iv) complex changes are more defect-prone than simpler ones [Hassan, 2009]. Finally, they divide the software features into three main categories: class-level [Gyimothy et al., 2005, Herbold, 2015], entropy [Hassan, 2009, Kaur et al., 2015], and change features [Moser et al., 2008, Rhmann et al., 2020]. The software features included in the data are part of Object-Oriented Programming (OOP) as they represent aspects included in that paradigm. In the end, the authors propose a new set of features for defect prediction based on code churn. D'Ambros et al. [2010] state that the dataset may serve as a benchmark for further explorations and comparisons between machine learning approaches for defect prediction.

More recently, Ferenc et al. [2020a] presented a collection of relevant open-source projects with a wide range of software features and quality attributes[5]. The authors merged several bug reports available in the literature with static features gathered using the OpenStaticAnalyzer (OSA) [Department of Software Engineering, 2022]. To do

[4] https://bug.inf.usi.ch/index.php
[5] https://zenodo.org/record/3693686

so, they considered five bug reports that are widely available in the defect prediction literature [Tóth et al., 2016]. Furthermore, they assessed the capabilities of the unified dataset in defect prediction with a decision tree model. As a result, they produced a large unified dataset comprising 47,618 classes with 71 software features. Using the OpenStaticAnalyzer, the software features can be divided into seven quality attributes: (i) Clone Duplication, (ii) Cohesion, (iii) Complexity, (iv) Coupling, (v) Documentation, (vi) Inheritance, and (vii) Size. Finally, the authors concluded that the unified dataset is a valuable resource for defect prediction [Ferenc et al., 2018, 2020a], and they encourage the use of the dataset for future explorations [Ferenc et al., 2020b]. We provide more details about the datasets discussed in this section in Chapter 3.

Learning from Source Code Features. Software defect prediction using machine learning techniques has received extensive recognition in the software engineering community for a long time. Several research studies rely on source code metadata [Wang et al., 2016a] and software features [Menzies et al., 2007, Jing et al., 2014] as features for machine learning-based algorithms. For instance, Wang et al. [2016a] studied the impact of using the program's semantics as features for the prediction model. The authors used deep learning networks to automatically learn semantic features from token vectors obtained from abstract syntax trees [Wang et al., 2016a]. In a similar approach, Xu et al. [2018] employed a non-linear mapping method to extract representative features by embedding the original data into a high-dimensional space. Their results achieved an average F-measure, g-mean, and balance of 0.480, 0.592, and 0.580 [Xu et al., 2018].

The current literature applies several software features for defect prediction. As an example, Menzies et al. [2007] presented defect classifiers utilizing code attributes defined by McCabe and Halstead features. They concluded that the choice of the learning method is more important than which subset of the available data we use for learning the software defects. The study also discusses the usefulness of defect prediction models for software development [Menzies et al., 2007]. From a different perspective, Jing et al. [2014] used a dictionary learning technique to predict software defects by using characteristics of software features mined from open-source software projects. They employed datasets from NASA projects as test data to evaluate the proposed method, which achieved a recall value of 79%, improving the recall by 15% compared to other methods previously employed to predict defects in the same dataset [Jing et al., 2014].

Some studies investigate cross-project defect prediction with cross-company defect prediction [Fukushima et al., 2014, Turhan et al., 2009]. For instance, Fukushima et al. [2014] explored cross-project prediction models within the context of just-in-time prediction. Their results demonstrate no relationship between project prediction performance and cross-project prediction performance. Furthermore, they conclude that just-in-time prediction models built using projects with similar characteristics or applying ensemble

methods usually perform well in a cross-project context [Fukushima et al., 2014]. In a similar approach, Turhan et al. [2009] used cross-company data for building localized defect predictors. They employed principles of analogy-based learning to cross-company data to fine-tune these models for localization. The authors analyzed static code features extracted from the source code, such as complex software features and Halstead code complexity features. The paper concludes that cross-company data are useful in extreme cases and when within-company data is not available [Turhan et al., 2009].

Similarly, He et al. [2012] investigate defect prediction based on data selection. The authors propose a brute force approach to select the most relevant data for learning the software defects. To do so, they experiment with three large-scale experiments on 34 datasets obtained from ten open-source projects. They conclude that training data from the same project does not always help to improve the prediction performance [He et al., 2012]. In the same direction, the work of Turhan et al. [2011] evaluates the effect of mixing data from different project stages. In this case, the author uses within and cross-project data to improve the prediction performance. They show that mixing project data based on the same project stage does not significantly improve the model performance. For this reason, they conclude that optimal data for defect prediction is still an open challenge for researchers [Turhan et al., 2011].

Defect prediction is a challenging task, and previous work has addressed this subject [Tantithamthavorn and Hassan, 2018, Tantithamthavorn et al., 2015]. For instance, Tantithamthavorn and Hassan [2018] documented the pitfalls and difficulties of applying novel defect prediction modeling. The authors divided their model into seven steps: hypothesis formulation, designing features, data preparation, model specification, model construction, model validation, and model interpretation. They discussed the pitfalls for each step of the proposed defect modeling process [Tantithamthavorn and Hassan, 2018]. In a different paper, Tantithamthavorn et al. [2015] showed the impact of noisy data on creating defect prediction models. They argue that mislabeled data could not only impact the effectiveness but also the reliability of the model. The authors apply a case study with thousands of manually curated issue reports. Next, we will discuss the learning methods where researchers use metadata information to predict software defects [Tantithamthavorn et al., 2015].

Furthermore, studies also differ in terms of the scope of the software features they employ to predict defects [Tosun et al., 2010, Pascarella et al., 2020]. For instance, Pascarella et al. [2020] reports negative results about method-level defect prediction using a set of popular Java projects. They conclude that (i) the performance of previously proposed models, tested using the same strategy but on different systems/timespans, is, in fact, comparable. However, (ii) when evaluated with a more practical strategy, all the models show a dramatic drop in performance, with results close to that of a random classifier. Finally, they find that (iii) the contribution of some features is limited. Fur-

thermore, Tosun et al. [2010] conducted a study in a large telecommunications company in Turkey to employ a software measurement program and predict pre-release defects. They conclude that implementing statistical techniques and machine learning in a real-life scenario is a difficult yet possible task. Therefore, they recommend optimizing the hyperparameter space of defect models.

Learning from Metadata Information. The prediction of software defects is a complex task by definition. In some cases, the source code features, such as the ones mentioned in the previous section, are not sufficient and efficient for the defect prediction task. For these reasons, many papers have adopted models that employ code metadata information. For example, Wang et al. [2016a] examined the impact of using a system's semantics as the prediction model's features. The authors used deep belief networks to automatically learn these features from token vectors collected from abstract syntax trees. Then, they evaluated the model on ten open-source projects and improved the F1 score for both within-project defect prediction by 14.2% and cross-project defect prediction by 8.9% [Wang et al., 2016a]. Similarly, the works of Xu et al. [2018] employed a non-linear mapping method to extract representative features by embedding the initial data into a high-dimensional space. Their results achieved an average F-measure, g-mean, and balance of 0.480, 0.592, and 0.580, respectively, and outperformed nearly all baseline methods [Xu et al., 2018].

In a similar perspective, Zhongbin et al. [2018] compared the effectiveness of predicting software defects from the Jureczko dataset. The authors demonstrate that the model performance is more dependable on the machine learning process. For instance, they discuss that applying an optimal data cleaning process in the target data is more important than the machine learning model. Furthermore, the authors conclude that data cleaning is necessary for the Jureczko dataset as they achieved better model performance. To do so, the authors compare the performance of a cleaned version of the Jureczko dataset with the raw data available on the internet [Zhongbin et al., 2018]. In the same direction, Ferenc et al. [2018] gathered a wide variety of defect prediction datasets, including Jureczko, and studied the accuracy of decision trees. The authors concluded that these datasets for defect prediction could join data to create a resource for the community. Next, we discuss the different machine learning methods commonly applied to predict defects in source code [Ferenc et al., 2018].

Explaining Software Defects. Software defect explainability is a relatively recent topic in the machine learning community [Mori and Uchihira, 2018, Jiarpakdee et al., 2020, Pornprasit et al., 2021]. Mori and Uchihira [2018] analyzed the trade-off between the accuracy and interpretability of various defect models. The experiment displays a comparison between the balanced output that satisfies both accuracy and interpretability

criteria. To do so, the authors propose a unique classification machine learning model called superposed naive Bayes (SNB), which transforms a naive Bayes ensemble method into a simple naive Bayes model by linear approximation. To evaluate the SNB interpretability, they apply a qualitative approach that assesses the interpretability of different types of classification techniques based on another study [Lipton, 2016]. The approach concentrates on model transparency (simulatability), component transparency (decomposability), and algorithmic transparency. They conclude that the proposed method may deliver a balanced output that satisfies both accuracy and interpretability criteria [Mori and Uchihira, 2018]. This thesis does not focus on qualitative approaches to explain software defects. Instead, we rely on SHAP values to understand the machine learning models and the software features. Finally, we employ a qualitative evaluation of developers' perception of the software features that may indicate defects and the machine learning model itself.

Likewise, Jiarpakdee et al. [2020] empirically evaluated two model-agnostic procedures, Local Interpretability Model-agnostic Explanations (LIME) [Ribeiro et al., 2016] and BreakDown [Staniak and Biecek, 2019] techniques. To do so, the authors investigated a large case study with several defect datasets and nine open-source projects. The authors find that (i) model-agnostic techniques are relevant to explain individual predictions of defect prediction models; (ii) instance explanations generated by model-agnostic techniques mostly overlap with the global explanation of defect models (except for the LIME technique) and are reliable when they are re-generated; (iii) model-agnostic techniques usually are fast to generate instance explanations (i.e., they do not take more than a minute); and (iv) the explanations are necessary and useful to understand the predictions of defect models. In the end, they improve the results obtained with LIME using hyperparameter optimization, which they called LIME-HPO. This work concludes that model-agnostic methods are necessary to explain individual predictions of defect models [Jiarpakdee et al., 2020]. This thesis also focuses on model-agnostic methods to explain individual predictions of defect models. However, we employ a larger set of software features, machine learning models, and a different technique to understand the defects (i.e., SHAP [Lundberg and Lee, 2017, Lundberg et al., 2018b]). Furthermore, we measure developers' understandability using SHAP explanations and the software features they perceived as defect predictors.

Finally, Pornprasit et al. [2021] proposes a tool called PyExplainer that predicts defects using the Python programming language. The input data consists of 40,978 software commits, and the authors compare its performance with the LIME technique [Ribeiro et al., 2016]. Through a basic case study of two large open-source projects (OpenStack and Qt [McIntosh and Kamei, 2018]), they conclude that PyExplainer produces (i) synthetic neighbors that are 41%-45% more similar to an instance to be explained; (ii) 18%-38% more accurate local models; and (iii) explanations that are more unique and consistent

with the actual characteristics of defect-introducing commits compared to LIME. The authors claim that the tool might help practitioners focus on the most important aspects of the commits to mitigate the risk of being defect-introducing. They conclude that the results are comparable to state-of-the-art technology to explain models [Pornprasit et al., 2021]. Unlike these papers, this thesis does not focus on commits that may cause software defects. Instead, we focus on a large set of software features that may indicate software defects. Furthermore, we are interested in the usefulness of the machine learning models for the developers as they are the most beneficiary from the understandability concepts applied to software defects. Next, we discuss the methods to detect code smells on the source code.

Code Smells Detection. Several automated detection strategies for code smells and anti-patterns have been proposed in the literature [Fokaefs et al., 2011, Khomh et al., 2011, Maiga et al., 2012b,a, Fontana et al., 2013, Amorim et al., 2015, Fernandes et al., 2016, Fontana et al., 2016, Di Nucci et al., 2018, Oizumi et al., 2018, Cruz et al., 2020, PMD, 2021]. They use diverse strategies in their identification. For instance, some methods are based on combinations of features [Oizumi et al., 2018, PMD, 2021], refactoring opportunities [Fokaefs et al., 2011], textual information [Palomba et al., 2016], historical data [Palomba et al., 2013], and machine learning techniques [Khomh et al., 2011, Maiga et al., 2012b,a, Fontana et al., 2013, Amorim et al., 2015, Fontana et al., 2016, Di Nucci et al., 2018, Cruz et al., 2020]. Khomh et al. [2011] used Bayesian Belief Networks to detect three anti-patterns. They trained the models using two Java open-source systems. Maiga et al. [2012b] investigated the performance of Support Vector Machines trained in three systems to predict four anti-patterns. Later, the authors introduced a feedback system to their machine learning model [Maiga et al., 2012a]. Amorim et al. [2015] investigated the performance of Decision Trees to detect four code smells in one version of the Gantt software project. Differently from these works, our dataset is composed of fourteen systems, and we evaluate nine class-level code smells (Chapter 7).

Cruz et al. [2020] evaluated seven machine learning models to detect four code smells in twelve systems. The authors discussed that algorithms based on trees encountered a better F1 score than other machine learning models. Fontana et al. [2016] evaluated six machine learning models to predict four smells. However, they used the severity of the smells as the target. They reported high-performance numbers for the evaluated models. Later, Di Nucci et al. [2018] replicated [Fontana et al., 2016] to address several limitations that potentially generated bias in the models' performance. The authors found out that the models' performance, when compared to the reference study, was 90% lower, indicating the need to further explore how to improve code smell prediction. In contrast to previous work on code smell prediction, we are interested in exploring the similarities and differences between models for predicting code smells and defect prediction models

(Chapter 7). Next, we discuss how the current literature investigate defects and code smells.

Defects and Code Smells. Several studies have tried to understand how code smells can affect the software life-cycle, evaluating different aspects of quality, such as maintainability [Fontana et al., 2013, Sjøberg et al., 2013, Yamashita and Counsell, 2013], modularity [Santana et al., 2021], program comprehension [Abbes et al., 2011], change-proneness [Khomh et al., 2009, 2012], and how developers perceive code smells [Yamashita and Moonen, 2013, Palomba et al., 2014]. Other studies aim to evaluate how code smells impact defect proneness [Hall et al., 2014, Jebnoun et al., 2022, Khomh et al., 2012, Olbrich et al., 2010, Openja et al., 2022, Palomba et al., 2018]. For instance, Olbrich et al. [2010] evaluated the fault-proneness evolution of God Class and Brain Class in three open-source systems. They discovered that classes with these two code smells can be more faulty, although this finding is not valid for all analyzed systems. Similarly, Khomh et al. [2012] evaluated the impact of 13 different smells on fault-proneness in several versions of three large open-source systems. They reported the existence of a relationship between some code smells and defects, but it is not consistent for all system versions. Additionally, Openja et al. [2022] evaluated how code smells can make a class more fault-prone in quantum projects. Unlike these studies, we aim to understand whether models built for defects and code smells are similar or not (Chapter 7).

Regarding the relationship between code smells and defects, Hall et al. [2014] investigated whether files with smells have more defects than files without them. They found that, for most of the analyzed smells, there was no statistically significant difference between smelly and non-smelly classes. In contrast, Palomba et al. [2018] evaluated the impact of 13 code smells on the presence of defects using a dataset of 30 open-source Java systems. They reported that classes with smells tend to have more bug fixes than those that do not have any smells. Moreover, Jebnoun et al. [2022] studied the relationship between Code Clones and defects in three different programming languages and found that smelly classes are more defect-prone, although this varies depending on the language. In contrast to these three studies, our focus is on understanding the differences between models used for defect prediction and those used for code smell detection, rather than establishing a correlation between defects and code smells (Chapter 7).

2.6 Final Remarks

This chapter presented a contextualization of software defects and how the current literature predicted them in the source code. For this reason, we started with a definition of software defects. Then, we introduced code smells as we compared them with the defect models because they may share similar software features (Chapter 7). In addition, we presented quality attributes that grouped several software features. Hence, we discussed the machine learning models commonly applied to predict defects in the literature. We noted that most studies focused on the Random Forest algorithm within the Java programming language with binary classification. Unfortunately, most of the datasets used to predict software defects were not available for replication. Furthermore, we also introduced in this chapter a set of suitable evaluation metrics for the classification. Finally, we discussed the related works of this thesis project. As a result, we identified four main datasets to predict software defects in the current literature [Menzies et al., 2007, D'Ambros et al., 2010, Jureczko and Madeyski, 2010, Ferenc et al., 2018]. These datasets mostly employed a set of software features based on Object-Oriented Programming (OOP) and their complexity and size. We also discussed relevant studies about learning from source code features and metadata information. Then, we presented recent investigations about the understandability of software defect models using different techniques. To conclude, we discussed studies that compared defect prediction models with code smell detection. In the next chapter, we will present an overview of the datasets employed to predict software defects identified in the literature. Thus, we will focus on the software features that compose these data sources. We identified these data sources from the ad-hoc literature review executed in the early stages of this thesis and discussed in this chapter.

Chapter 3

Datasets for Defect Prediction

Machine learning and statistical modeling techniques are commonly used to predict defects in software systems. As a result, the effectiveness of defect prediction heavily relies on the quality of the data used, as it provides insights about the software features for the development team [Menzies et al., 2010, Ghotra et al., 2015, Tantithamthavorn and Hassan, 2018]. After discussing the studies that introduced the datasets in Section 2.5, this chapter focuses on the dataset's features and how they can be grouped into categories. Additionally, the class-level features can be clustered into seven quality attributes. We identified these datasets from an ad-hoc literature review conducted in the early stages of this thesis. These datasets vary in size, software features, and how they implement the concept of defects. The selected datasets come from four relevant sources for research, and they illustrate the importance of data quality and reproducibility for defect prediction. Furthermore, these datasets are popular in the community [Menzies et al., 2004, 2007, 2010, Jureczko and Madeyski, 2010, Ghotra et al., 2015, Ferenc et al., 2018, 2020a] and are publicly available on the internet. Although most software features are unique to each dataset and software project, the datasets share the commonly found imbalanced nature in the literature. In this case, the data sources for defect prediction usually have more non-defective instances (i.e., classes or modules) than defective ones.

We have organized the remainder of this chapter into seven sections. Section 3.1 introduces the NASA dataset and explains how we use it for defect prediction [Menzies et al., 2010, Ghotra et al., 2015]. In Section 3.2, we discuss the Jureczko dataset and provide information about the data distribution of each project [Jureczko and Spinellis, 2010, Jureczko and Madeyski, 2010]. Section 3.3 presents the Bug Prediction dataset, including its data distribution and applications [D'Ambros et al., 2010]. We discuss the Unified dataset for defect prediction in Section 3.4, highlighting its wide range of features [Ferenc et al., 2018, 2020b]. In Section 3.5, we describe the software features used by each dataset to predict defects and divide them into five categories. It is worth noting that the datasets differ in the software features they provide for defect prediction. Therefore, in Section 3.6, we present the quality attributes that group the software features [Ferenc et al., 2020a]. Finally, in Section 3.7, we conclude the chapter by providing insights into the different opportunities to explore defect prediction using these datasets.

3.1 NASA Dataset

The NASA dataset is an important source of data for the defect prediction community. One of the main reasons for its popularity is that it was one of the first large publicly available datasets. The dataset represents software modules with their respective defect distribution and contains features from Halstead's operator-operand counts [Halstead, 1977] and McCabe's dependencies and complexity [McCabe, 1976, McCabe and Butler, 1989]. Furthermore, the data are easy to handle, and the software features require little processing to predict defects with fair accuracy [Menzies et al., 2010]. As the dataset has multiple available versions, the current literature usually investigates the cleaned version of the data [Ghotra et al., 2015]. This allows us to avoid the need to deeply process the data and maintain a version with fewer inconsistencies in terms of the overall data quality of each module [Ghotra et al., 2015]. The dataset is popular in the defect prediction community and is the subject of study in many contexts of the field [Menzies and Stefano, 2004, Menzies et al., 2007, 2010, Agrawal and Menzies, 2018]. The various NASA projects comprise a broad range of NASA systems, with CM1 referring to spacecraft instruments, *KC1*, *KC3*, *MC2* referring to storage management for grounded data, *MW1* managing data transactions, and *PC1*, *PC2*, *PC3*, *PC4* referring to software systems for an earth-orbiting satellite.

Table 3.1 exemplifies the dataset with the five projects that have the most instances, along with their features (out of the nine projects available in the dataset). The NASA projects contain 21 software features, mostly related to McCabe and Halstead software features. For each module within a project, a value is assigned to each software feature. The average value for a project represents the sum of values assigned to each module divided by the number of modules within the project (Table 3.1). These average values differ between projects, as observed in Table 3.1. For example, the average BRANCH_COUNT (number of branches) for project *CM1* is 12.98, whereas for *KC1*, it is 7.24. Table 3.1 also shows the percentage of defective modules in each project, and it is clear that the data is imbalanced, i.e., defective software modules are heavily underrepresented in comparison with non-defective modules. The data is not balanced, as the number of defective modules is not equal to the number of non-defective modules. The modules are associated with three programming languages: C, C++, and Java. Java is not included in Table 3.1 because we only display the top-5 projects. As the NASA features differ from the features used in other datasets, we include a description of all features in Appendix A.

Table 3.1: Overview of NASA Data Program Features (Top-5 Projects out of 9).

	Projects	KC1	PC1	PC2	PC3	PC4
	Programming Language	C++	C	C	C	C
	Number of Modules	1,571	1,059	4,505	1,511	1,347
	Defective Modules	20.31%	7.18%	0.51%	10.59%	13.21%
1	BRANCH_COUNT	7.24	13.20	7.62	12.60	8.28
2	CYCLOMATIC_COMP.	4.13	7.41	4.39	6.99	4.75
3	DESIGN_COMP.	3.63	4.32	3.18	3.62	2.88
4	ESSENTIAL_COMP.	2.17	3.46	2.19	2.97	2.28
5	HALSTEAD_CONTENT	31.37	37.52	22.95	43.52	28.46
6	HALSTEAD_DIFFICULTY	10.36	20.34	14.38	18.30	18.09
7	HALSTEAD_LEVEL	0.19	0.08	0.11	0.08	0.11
8	HALSTEAD_EFFORT	9248	42547	12995	47008	21432
9	HALSTEAD_ERR._EST	0.15	0.32	0.14	0.35	0.20
10	HALST._LENGTH	82.00	157.45	79.46	162.21	110.89
11	HALST._PROG_TIME	513.8	2363.7	721.94	2611.6	1190.6
12	HALSTEAD_VOLUME	438.11	964.11	426.10	1036.2	596.65
13	LOC_BLANK	2.98	8.76	11.12	8.22	7.82
14	LOC_CODE_AND_COMM.	0.21	1.43	14.55	1.75	2.38
15	LOC_COMMENTS	1.65	5.80	5.74	5.73	5.49
16	LOC_EXECUTABLE	24.17	29.85	2.62	28.26	20.47
17	LOC_TOTAL	32.28	31.27	17.17	30.01	22.85
18	NUM_OPERANDS	31.16	68.43	32.45	72.68	42.72
19	NUM_OPERATORS	50.84	89.02	47.01	89.54	68.17
20	NUM_UNIQ._OPERAN.	15.07	27.20	12.70	27.77	14.40
21	NUM_UNIQ._OPERAT.	10.58	16.46	11.88	15.21	12.73

Source: Elaborated by the author.

3.2 Jureczko Dataset

The Jureczko dataset is a collection of projects related to various aspects of the source code. The data contain features representing the CK features [Jureczko and Madeyski, 2010]. The dataset introduces the concept of defining a defect based on the commit message and comments that indicate a defect fix [Jureczko and Spinellis, 2010]. The dataset includes several versions of different repositories, and the cleaned version maintains data from seven software projects, making it the most common representative of the dataset [Zhongbin et al., 2018]. In this version, the Jureczko dataset consists of 23 releases of the seven software projects. Thus, each instance in the data represents a Java class corresponding to the software features and a label indicating the number of defects in the selected class. As is commonly applied in the literature, we preprocess the Jureczko datasets by considering a software class as defective if the value of the labeled feature is equal to or greater than one [Ghotra et al., 2015, Zhongbin et al., 2018]. The current

literature employs this adaptation to transform the dataset into a classification problem [Jureczko and Madeyski, 2010, Zhongbin et al., 2018]. As a result, we process the Jureczko dataset with 20 features and a binary label (i.e., defective or clean class instance). Another predominant characteristic of the Jureczko dataset is its imbalanced nature. As a result, the data have more clean classes than defective classes. Specifically, the Jureczko dataset has a ratio of approximately 26% of the total number of classes being defective, while the rest are non-defective. The proportion of defective/clean classes is similar to that of the NASA dataset [Menzies et al., 2007]. Furthermore, we employ a technique known as Synthetic Minority Oversampling Technique (SMOTE) [Tantithamthavorn et al., 2019] to deal with the imbalanced nature of the dataset.

Table 3.2 shows the data for seven software projects and the 20 software features of the Jureczko dataset [Jureczko and Madeyski, 2010, Jureczko and Spinellis, 2010]. These projects are the *Tomcat*, *Ant*, *Log4J*, *Prop*, *Xalan*, *Camel*, and *JEdit* [Zhongbin et al., 2018]. The Java programming language is predominant in all seven projects. Again, for each software class, there is a value attached to each software feature. The average value is the sum of all values referred to each module divided by the number of classes in the project. These average values differ between projects, for instance, the software feature LCOM (lack of cohesion of methods) for project *JEdit* is 233.00, and for the software system *Log4J* is only 37.17. Table 3.2 also illustrates the percent of defective modules in each project and the imbalanced nature of the data [Zhongbin et al., 2018]. Furthermore, we conclude that the proportion of defective classes varies in the dataset. The lowest number of defects is only 8.87% from the *Tomcat* project, while we find the highest number of defects in the *Log4J* project, where 57.90% of the modules had at least one defect.

3.3 The Bug Prediction Dataset

Another source of data for defect prediction is known as the Bug Prediction dataset [D'Ambros et al., 2010]. In contrast to the other datasets discussed previously in this chapter, these data are from several sources. First, D'Ambros et al. [2010] report the application of changelogs, including reconstructed transactions and links to model the software classes. Second, they identify software defects using a defect repository. Third, the authors report the use of Biweekly versions of the systems parsed into object-oriented models. Fourth, they employ values of all features used as machine learning predictors considering each version of each software class. Finally, they use data from post-release defect counts for each software class. As a result, the dataset contains software classes

Table 3.2: Jureczko Dataset Features.

	Projects	TOMCAT	ANT	LOG4J	PROP	XALAN	CAMEL	JEDIT
	Classes	835	745	449	69653	3320	2784	1749
	Defects (%)	8.97%	22.28%	57.90%	12.28%	54.39%	20.18%	17.32
1	WMC	12.95	11.07	7.72	5.22	11.15	8.43	12.78
2	DIT	1.68	2.52	1.67	3.02	2.54	1.95	2.61
3	NOC	0.36	0.73	0.26	0.59	0.55	0.52	0.45
4	CBO	7.65	11.04	7.20	15.03	12.76	10.68	13.33
5	RFC	33.47	34.36	23.58	24.65	29.52	20.87	39.48
6	LCOM	176.27	89.14	37.17	37.59	127.51	70.40	233.00
7	Ca	3.86	5.65	3.93	2.82	6.07	5.13	8.10
8	Ce	0.0	5.74	3.61	12.27	7.39	6.13	6.77
9	NPM	10.77	8.36	5.22	3.46	9.08	6.84	7.75
10	LCOM3	1.08	1.01	1.00	1.35	1.14	1.08	1.03
11	LOC	350.43	280.07	177.45	170.23	412.72	111.76	457.30
12	DAM	0.57	0.64	0.22	0.19	0.43	0.61	0.52
13	MOA	0.94	0.72	0.81	0.091	0.80	0.65	1.05
14	MFA	0.29	0.50	0.29	0.61	0.54	0.39	0.49
15	CAM	0.48	0.47	0.43	0.55	0.47	0.49	0.45
16	IC	0.27	0.72	0.34	1.08	0.80	0.37	0.64
17	CBM	0.59	1.31	0.66	1.71	2.87	0.71	1.53
18	AMC	25.57	23.64	20.25	30.27	57.36	10.94	30.64
19	MAX_CC	4.27	4.66	3.43	3.30	4.35	2.17	6.72
20	AVG_CC	1.25	1.36	1.34	1.28	1.32	0.94	1.83

Source: Elaborated by the author.

and references to defects from five different projects. These projects focus on the Java programming language. Besides, the five projects are open-source and available for contributions from developers. D'Ambros et al. [2010] argued that the use of the same programming language avoids inconsistencies in the target software features. Furthermore, the same programming language allowed the same parser to collect the data from the different repositories. Thus, they avoided issues with reverse engineering tools.

Table 3.3 presents the entire set of features reported in the Bug Prediction dataset. As we can observe, the data have 37 software features related to specific characteristics of the source code [Moser et al., 2008]. Furthermore, the data present features associated with the Object-Oriented Programming paradigm. The dataset also holds information on 5,371 software classes. We associate these classes with five Java projects (*Eclipse JDT*, *Eclipse PDE*, *Equinox*, *Lucene*, and *Mylyn*). Table 3.3 also presents the average numbers of each software feature in the five projects (i.e., 37 software features). We remark that the number of defective classes varies among each project. For example, for the *Equinox* project around 66% of its classes are defective, while only 10% of *Lucene* presented defects (Table 3.3). As in the case of the other datasets, the data is imbalanced as only approximately 20% of the software classes (i.e., instances) represent a defect.

Table 3.3: The Bug Prediction Dataset Features.

	Projects	JDT	PDE	Equinox	Lucene	Mylyn
	Classes	997	1,497	324	691	1,862
	Defects (%)	26.4%	16.2%	66.15%	10.2%	15.1%
1	NR	45.61	3.51	11.39	6.26	10.98
2	NFIX	7.64	0.48	1.35	8.54	0.25
3	NREF	0.01	0.19	0.16	0.1	0.27
4	NAUTH	5.79	3.97	2.54	2.9	1.93
5	LINES	1209.4	211.70	278.78	104.62	106.38
6	mLINES	222.25	68.55	100.63	42.34	38.41
7	aLINES	14.86	12.49	16.33	9.36	6.65
8	LINESr	1209.4	14.21	162.90	75.9	88.58
9	mLINESr	222.25	57.12	66.36	38.77	37.09
10	aLINESr	11.15	8.17	7.22	7.69	5.48
11	CHURN	228.18	68.48	115.87	28.72	17.78
12	mCHURN	78.26	48.84	60.52	19.32	19.72
13	aCHURN	3.71	4.32	9.1	1.67	1.17
14	AGE	279.07	167.95	140.88	149.15	71.01
15	wAGE	63.63	60.69	27.37	52.17	23.44
16	HCM	10.68	8.54	5.47	4.01	5.22
17	WHCM	0.05	0.04	0.15	0.07	0.02
18	LinHCM	0.07	0.02	0.01	0.03	0.11
19	LogHCM	3.66	0.26	0.14	0.23	5.78
20	ExpHCM	0.12	0.05	0.03	0.06	0.19
21	CBO	12.21	10.20	9.67	8.73	8.16
22	DIT	2.72	2.28	1.23	1.76	1.45
23	FanIn	5.36	3.69	2.95	4.66	3.67
24	FanOut	7.39	6.67	7.14	4.16	4.63
25	LCOM	364.72	82.17	124.22	63.28	72.68
26	NOC	0.71	0.59	0.17	0.72	0.42
27	NOA	7.38	5.54	6.7	4.85	5.01
28	NOIA	102.21	3.9	1.45	1.52	0.69
29	LOC	224.72	98.16	122.01	105.91	83.83
30	NOM	13.58	9.62	9.87	7.41	7.81
31	NOMI	49.23	28.62	14.72	20.44	14.52
32	NOPRA	1.67	2.67	3.49	2.61	3.16
33	NOPRM	1.29	2.24	2.08	1.32	1.01
34	NOPUA	2.74	1.88	1.4	1.13	0.93
35	NOPM	8.95	5.57	5.74	4.78	6.03
36	RFC	76.87	47.5	58.33	33.39	34.82
37	WMC	58.38	23.74	32.64	23.68	16.78

Source: Elaborated by the author.

3.4 Unified Dataset

The unified dataset represents a merged version of several resources available for the defect prediction community [Ferenc et al., 2018, 2020a,b]. In total, five data sources provided bug reports about software classes: PROMISE [Sayyad S. and Menzies, 2005], Eclipse Bug Prediction [Zimmermann et al., 2007], Bug Prediction Dataset [D'Ambros et al., 2010], Bugcatchers Bug Dataset [Hall et al., 2014], and GitHub Bug Dataset [Tóth et al., 2016]. PROMISE uses Buginfo to collect whether an SVN or CVS commit is a bugfix or not [Sayyad S. and Menzies, 2005]. Eclipse Bug Prediction extracts defect information from the CVS repository and Bugzilla [Zimmermann et al., 2007]. The Bug Prediction dataset applies to commit logs of SVN and the modification time of each file (Section 3.3). Bugcatchers Bug Dataset employs an Ant script that associates the defects of a file with the SVN or CVS information [Hall et al., 2014]. GitHub Bug Dataset uses the GitHub feature that handles references between commits and issues and then utilizes this information to match commits with bugs [Tóth et al., 2016]. The Unified dataset uses a tool called OpenStaticAnalyzer (OSA) [Department of Software Engineering, 2022] to extract software features from the source code. OSA is a source code analyzer tool that can perform deep static analysis of the source code of complex Java systems and other languages [Ferenc et al., 2020a]. The calculated features are class-level, as they are extracted from the source code of the classes [Ferenc et al., 2018]. To calculate the feature values, the Unified dataset uses the same release version of a given project [Ferenc et al., 2020a]. Therefore, as the project was open-source in the five resources, the authors downloaded and analyzed it individually.

The final version of the dataset contains 47,618 classes from 34 open-source Java projects [Ferenc et al., 2020b]. Some of the relevant projects include *Ant*, *Broadleaf*, *Camel*, *Elasticsearch*, *Hazelcast*, *JDT*, *JEdit*, *Lucene*, *Neo4J*, *OrientDB*, *PDE*, *POI*, *Titan*, *Xalan*, among others. Furthermore, the data comprises 71 software features related to different quality attributes, such as (i) Clone Duplication, (ii) Cohesion, (iii) Complexity, (iv) Coupling, (v) Documentation, (vi) Inheritance, and (vii) Size. The dataset is imbalanced as only around 20% of the classes represent a software defect [Ferenc et al., 2018], which is a known characteristic of defect prediction datasets [Menzies et al., 2007, Jureczko and Madeyski, 2010, D'Ambros et al., 2010]. As the number of features and projects is too high to be presented in a table that fits on a page, we provide a complete overview of the dataset in the replication package of this thesis [dos Santos, 2023b].

3.5 Features for Defect Prediction

Estimating software defects involves predicting defects based on software features. Therefore, we require information about the behavior of the source code, and these features provide such information. This section presents an overview of the software features used to predict defects in the source code. These features are relevant for three reasons: (i) they are used to predict defects in the source code, (ii) they are used to explain the machine learning models, and (iii) they are used to validate the results of our research project with developers. Thus, we divide this analysis into five categories derived from the four datasets presented in the previous sections. First, we discuss the class-level features [D'Ambros et al., 2010, Couto et al., 2012, Herbold, 2015]. Then, we examine the entropy features [Hassan, 2009, D'Ambros et al., 2010, Kaur et al., 2015]. Next, we present the change features [Moser et al., 2008, D'Ambros et al., 2010, Kumar and Sureka, 2017, McIntosh and Kamei, 2018, Rhmann et al., 2020]. We also analyze the McCabe and Halstead features [McCabe, 1976, Halstead, 1977]. Finally, we provide a list of additional features that do not fit into the aforementioned categories [Menzies and Stefano, 2004, Menzies et al., 2007, 2010, D'Ambros et al., 2010].

Class-Level Features - These software features consist of two complementary groups: CK and Object-Oriented features [D'Ambros et al., 2010, Jureczko and Spinellis, 2010, Jureczko and Madeyski, 2010, Couto et al., 2012, Herbold, 2015]. Six software features relate to CK, and eleven software features relate to the Object-Oriented (OO) paradigm [Jureczko and Spinellis, 2010, D'Ambros et al., 2010]. The list of software features related to the class-level is too long to fit on one page. Therefore, Appendix C shows all the software features divided by quality attributes, as we discuss in the next section. Table 3.4 provides an example of each of the seventeen software features related to this category. The first column shows the software feature's type (either CK or OO). The second column displays the acronym, and the last column presents the name of the feature. Notably, current literature considers Lines of Code (LOC) as one of the most critical features for defect prediction in various datasets [Gyimothy et al., 2005, Jiang et al., 2013].

Entropy Features - Table 3.5 describes the five features related to entropy. The first column shows the software feature's acronym, and the second column is the name of the feature. The fundamental idea behind code entropy consists of estimating how distributed changes happen in a system over a time interval [Hassan, 2009, D'Ambros et al., 2010, Kaur et al., 2015]. It suggests that the wider distributed the code is, the higher the code complexity [Kaur et al., 2015]. The intuition is that one change concerning one file is simpler than one affecting many modified

Table 3.4: Class-Level Source Features.

Type	Acronym	Name
CK	WMC	Weighted Method Count
CK	DIT	Depth of Inheritance Tree
CK	RFC	Response for Class
CK	NOC	Number to children
CK	CBO	Coupling Between Objects
CK	LCOM	Lack of Cohesion in Methods
OO	FanIn	Number of classes that reference the class
OO	FanOut	Number of classes referenced by the class
OO	NOA	Number of attributes
OO	NOPRA	Number of private attributes
OO	NOPUA	Number of public attributes
OO	NOAI	Number of attributes inherited
OO	LOC	Number of lines of code
OO	NOM	Number of methods
OO	NOPM	Number of public methods
OO	NOPRM	Number of private methods
OO	NOMI	Number of methods inherited

Source: Elaborated by the author.

files. This occurs because the developer who has to implement the change usually has to maintain all other files [D'Ambros et al., 2010]. For instance, the History Complexity Measure (HCM) is an entropy feature with four variants that take into account the *weight*, *linear*, *logarithmic*, and *exponential* measures of the feature [D'Ambros et al., 2010]. Entropy has demonstrated its efficiency for the defect prediction task [Hassan, 2009, D'Ambros et al., 2010].

Table 3.5: Entropy Level Features.

Acronym	Name
HCM	History Complexity Measure
WHCM	Weighted History Complexity Measure
LinHCM	Linearly Decayed History Complexity Measure
LogHCM	LoGarithmically decayed History Complexity Measure
ExpHCM	Exponentially Decayed History Complexity Measure

Source: Elaborated by the author.

Change Features - Table 3.6 presents a list of change features considered for defect prediction. The first column presents the feature acronym, and the second column shows the feature's name. These features are the principal topic related to the modification executed in source code files [Moser et al., 2008, D'Ambros et al., 2010, Kumar and Sureka, 2017, Rhmann et al., 2020]. Two features deserve attention: the AGE of a file and code churn (CHURN). The AGE describes the number of weeks since the file release date. This feature has one variation that considers the *weight*

of the file age (Table 3.6 last row). Likewise, CHURN is the sum of added lines of code plus deleted lines of code in modifications of a file. According to the current literature [Moser et al., 2008, D'Ambros et al., 2010, Kumar and Sureka, 2017], modifications executed in source code files are one of the major subjects for defects. Hence, the number of fixes in production (NFIX) stores this relevant information [D'Ambros et al., 2010].

Table 3.6: Change Features.

Acronym	Name
NR	Number of revisions
NREF	Number of times file has been refactored
NFIX	Number of times file was involved in bug-fixing
NAUTH	Number of authors who committed the file
LINES	Lines added and removed (sum, max, average)
CHURN	Codechurn (sum, maximum and average)
CHGSET	Change set size (maximum and average)
AGE	Age and weighted age

Source: Elaborated by the author.

Halstead and McCabe Features - Table 3.7 describes the software features related to McCabe and Halstead features [McCabe, 1976, Halstead, 1977, McCabe and Butler, 1989]. The first column represents the software feature's acronym, and the second column is the feature's name. Halstead [1977] proposed these software features to measure the source code complexity. Hence, the software features estimate the reading complexity by counting the number of operators and operands in a specific module. A complement to the Halstead features comes from the complexity features proposed by McCabe [1976]. Unlike Halstead [Halstead, 1977], McCabe and Butler [1989] argued that the complexity of pathways between module symbols is more insightful than just a count of the symbols [McCabe, 1976]. For this reason, the literature applied these software features collectively to measure code complexity. In addition, these classic measures are module-based features, meaning that they were originally proposed to anticipate the complexity of a module and also the reason for the quality of a software system [Weyuker, 1988, Menzies et al., 2007, Ghotra et al., 2015, Menzies and Zimmermann, 2013]. We give more details about the Halstead and McCabe features in Appendix A.

Additional Features - Table 3.8 lists the software features related to the source code that are not part of the changes, entropy, class-level, or Halstead and McCabe features. The first column shows the software feature acronym, and the second column displays the feature's name. These features are associated with crucial aspects of the code, such as coupling (CBM, Ca, Ce, IC), complexity (CC, AMC), and cohesion (CAM) [Jureczko and Madeyski, 2010, Jureczko and Spinellis, 2010].

Table 3.7: Halstead and McCabe Features.

Acronym	Name
Branch count	Number of branches
McCabe's complexity	Number of independent paths
McCabe's design	Complexity of a module
McCabe's essential	Degree of structuredness
Halstead content	Independent complexity of a module
Halstead difficult	Difficult to handle the module
Halstead level	Inverse of the error proneness
Halstead effort	Estimated mental effort
Halstead error	Number of errors in module
Halstead length	Operators and operands numbers
Halstead time	Estimate time to develop module
Halstead volume	Bits required to execute the module
Operands	Total number of operands
Operators	Total number of operators
Unique operands	Number of unique operands
Unique operators	Number of unique operators

Source: Elaborated by the author.

Additionally, a separate group of features pertains to the size of the module (DAM, MOA, MFA) [D'Ambros et al., 2010, Kumar and Sureka, 2017]. The McCabe complexity features are classic examples of defect predictors, and the NASA datasets use these features for the predictors [Menzies and Stefano, 2004, Menzies et al., 2007, 2010].

Table 3.8: Additional Features.

Acronym	Name
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Functional Abstraction
CAM	Cohesion Among Methods of Class
IC	Inheritance Coupling
CBM	Coupling Between Methods
AMC	Average Method Complexity
Ca	Afferent couplings
Ce	Efferent couplings
CC	McCabe's cyclomatic complexity (max and mean)

Source: Elaborated by the author.

3.6 Quality Attributes

Table 3.9 shows the software features that belong to each of the seven quality attributes discussed in the previous section (Section 2.3). These quality attributes are *clone*, *cohesion*, *complexity*, *coupling*, *documentation*, *inheritance*, and *size*. Clone measures the amount of code cloning present in the source code, typically detected by identifying copy/pasted code. Cohesion represents the degree of interdependence between source code attributes. Complexity measures the overall complexity of the source code attributes. Coupling measures the number of dependencies among the source code attributes. Documentation represents the amount of comments and documentation available for the source code attributes. Inheritance measures various aspects of the inheritance hierarchy within the system. Size represents the basic properties of the analyzed projects in terms of their size-related characteristics (e.g., number of lines of code, number of classes, number of methods). Table 3.9 lists the seven quality attributes along with their corresponding software features. The first column shows the acronym of each software feature, while the second column presents its name. To conserve space, we have limited the number of software features displayed to five, but Appendix C contains the full list of software features for each quality attribute. The third column indicates the quality attribute to which the software feature belongs, and the fourth column displays the distribution value of the software feature, which measures its central tendency among all 47,618 classes. Note that, on average, the open-source projects analyzed in this study consist of around 131 lines of code and 22 methods, with 44 statements. The replication package for this thesis contains the distribution value for each software feature [dos Santos, 2023b].

3.7 Final Remarks

This chapter discussed four datasets used to predict software defects in current literature. While these datasets share some similarities, they also have differences. For instance, they all are publicly available on the internet [Menzies et al., 2007, Jureczko and Spinellis, 2010, Jureczko and Madeyski, 2010, D'Ambros et al., 2010, Ferenc et al., 2018, 2020a], but they use different software features to predict defects. We classified the features into five categories: (i) class-level features, (ii) entropy features, (iii) change features, (iv) McCabe and Halstead features, and (v) additional features not correlated to the remaining ones. We also identified seven quality attributes that group the class-level

Table 3.9: List of Quality Attributes with Relevant Software Features.

Acronym	Feature	Quality Attribute	Mean
CC	Clone Coverage	Clone	1.139
CCL	Clone Classes	Clone	0.961
CCO	Clone Complexity	Clone	7.297
CI	Clone Instances	Clone	1.664
CLC	Clone Line Coverage	Clone	0.089
LCOM5	Lack of Cohesion in Methods 5	Cohesion	1.575
NL	Nesting Level	Complexity	1.522
NLE	Nesting Level Else-If	Complexity	1.319
WMC	Weighted Methods per Class	Complexity	17.876
CBO	Coupling Between Object Classes	Coupling	5.846
CBOI	Coupling Between Object Classes Inverted	Coupling	5.614
NII	Number of Incoming Invocations	Coupling	7.531
NOI	Number of Outgoing Invocations	Coupling	7.243
RFC	Response for a Class	Coupling	14.727
AD	API Documentation	Documentation	0.338
CD	Comment Density	Documentation	0.195
CLOC	Comment Lines of Code	Documentation	30.114
DLOC	Documentation Lines of Code	Documentation	23.555
PDA	Public Documented API	Documentation	2.773
DIT	Depth of Inheritance Tree	Inheritance	1.139
NOA	Number of Ancestors	Inheritance	1.417
NOC	Number of Children	Inheritance	0.604
NOD	Number of Descendants	Inheritance	1.219
NOP	Number of Parents	Inheritance	0.668
TLOC	Total Lines of Code	Size	131.904
TNG	Total Number of Getters	Size	4.844
TNM	Total Number of Methods	Size	22.036
TNOS	Total Number of Statements	Size	44.362
TNA	Total Number of Attributes	Size	12.629

Source: Elaborated by the author.

software features: (i) Clone Duplication, (ii) Cohesion, (iii) Complexity, (iv) Coupling, (v) Documentation, (vi) Inheritance, and (vii) Size. Java is the predominant programming language used in the Jureczko [Jureczko and Madeyski, 2010], Bug Prediction [D'Ambros et al., 2010], and Unified datasets [Ferenc et al., 2020a], whereas the NASA dataset [Menzies et al., 2007] includes some projects in C and C++. In the next chapter, we investigate the results of predicting and understanding software defects using baseline models and an explainable technique to reason about the impact of each software feature on the prediction.

Chapter 4

Predicting Defects with Machine Learning

Defect prediction can assist organizations in improving the quality of their software products by identifying potential defects early in the development process [Turhan and Bener, 2009]. This can save time and resources that would otherwise be spent on fixing defects after the product has been released [Knab et al., 2006]. This chapter investigates the use of machine learning techniques to predict software defects in two datasets commonly used in the literature: the Bug Prediction dataset (Section 3.3) and the Jureczko dataset (Section 3.2) [Jureczko and Spinellis, 2010, D'Ambros et al., 2010]. We exclude the NASA dataset results to maintain the clarity of the models, although Chapter 6 presents a study with developers using the features from that dataset. Similarly, the Unified dataset is not used in this chapter, although Chapter 7 investigates the defects and code smells using that dataset. First, we apply baseline machine learning models to predict software defects and evaluate their effectiveness in the target datasets. Then, we compare these baseline models with the implementation of gradient boosting machines, which we chose due to their superior performance compared to the baseline models (Tables 4.1 and 4.2). Our proposed implementation conducts an exploratory study that produces thousands of random machine learning models from a diverse collection of software features. These models are random because they promptly select the features from the entire pool of software features available for defect prediction. Although most of the models are ineffective, we were able to produce several models that yield accurate predictions, thus accurately predicting defects.

In the end, we analyze the predictive power of the machine learning models using the target datasets. Therefore, the primary objective of this chapter is to examine software features used for defect prediction, focusing on their predictive accuracy. We organize the remainder of this chapter into six sections. First, we discuss the study design with the main steps executed to explore the defect prediction datasets (Section 4.1). Here, we focus on data exploration and machine learning model implementation. Second, we present the main results of the exploratory investigation using the baseline models (Section 4.2). Third, we focus on the predictive power of several machine learning models for each

dataset (Section 4.3). Next, we address the implications of these chapter results for practitioners and the research community (Section 4.4). Then, we present the threats to the validity of our investigation (Section 4.5). Finally, we provide concluding remarks for this chapter (Section 4.6).

4.1 Study Design

This section discusses the methodology applied in this research alongside its main stages.

Goals and Research Questions. The primary objective of this chapter is to examine a pool of software features employed for defect prediction in terms of their power in two popular datasets [D'Ambros et al., 2010, Jureczko and Spinellis, 2010]. To investigate the predictive capacity, we introduce an algorithm that randomly selects the software features to predict defects. To do so, the algorithm arbitrarily selects software features for model generation. Guided by this goal, this chapter explores the following overarching research questions.

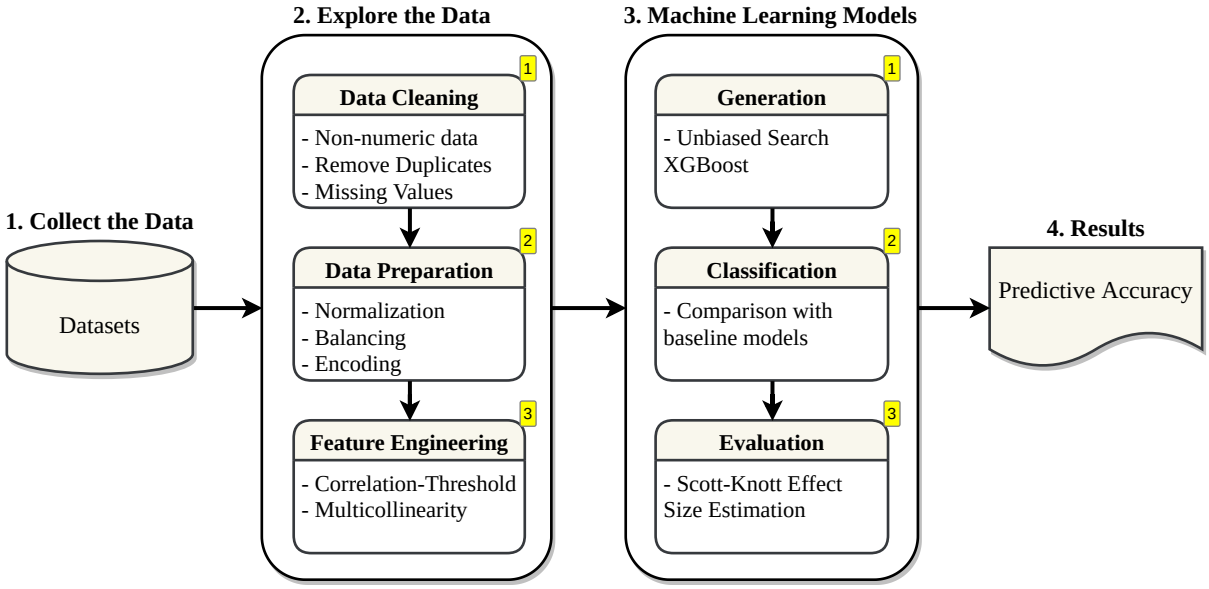
RQ1. How effective are random models in the defect prediction task?

RQ2. Which software features are more effective in predicting defects?

Research Method. To ease the understanding of each methodology phase, Figure 4.1 presents an overview of the method employed in the study. Note that the study has 4 main phases: (1) Collect the Data, (2) Explore the Data, (3) Machine Learning Models, and (4) Results. The rectangle in phase Exploration and Model indicates that the phase has several steps. For instance, we need to execute data cleaning, preparation and feature engineering for the Exploration phase (phase 2 of Figure 4.1). In addition, we employ model generation, classification and evaluation for phase 3 (i.e., Model of Figure 4.1). The first phase is the data exploration, where we analyze the datasets and the features available for defect prediction. The second phase is the baseline models, where we implement several machine learning models to predict defects. The third phase is the gradient boosting, where we implement the gradient boosting algorithm to predict software defects. The fourth phase is the model analysis, where we analyze the predictive power of the machine learning models.

The first phase is the selection of the dataset that contains defects (Collect the Data of Figure 4.1). The only criterion we use to determine the datasets for the experiment

Figure 4.1: Overview of the Methodology to Explore the Defect Prediction Data.



Source: Elaborated by the author.

is the quality of the software features available in the dataset. For this reason, the current literature considers the NASA dataset noisy and problematic [Gray et al., 2011, Ghotra et al., 2015]. Therefore, it is hard to reason about the results of the defect prediction models from that dataset, as there is little contextual information available for researchers [Turhan and Bener, 2009, Menzies et al., 2010]. Consequently, we excluded the NASA dataset from this chapter due to the software features being specific to classic McCabe and Halstead [Menzies et al., 2007, Gray et al., 2011]. Additionally, we did not use the Unified dataset [Ferenc et al., 2020a] because it was proposed after the empirical investigation discussed in this chapter. However, we used the software features from the Unified dataset in Chapter 7 to compare with the code smells. Therefore, we selected the Bug Prediction dataset [D'Ambros et al., 2010] and the Jureczko dataset [Jureczko and Madeyski, 2010, Jureczko and Spinellis, 2010]. These datasets offer a relevant set of features that may help with the defect prediction and understandability of the models.

In phase two (Explore the Data of Figure 4.1), we employ an extensive data exploration process to prepare the datasets and work on the features. First, we perform a data cleaning step as the datasets have some improper data in CSV format. We look for non-numeric fields, remove duplicated entries, and track missing values. Second, we perform a data preparation step where we normalize, balance, and encode the data. Third, we perform a feature engineering step aimed at the highly correlated features and checked for multicollinearity. As a result, we end up with 37 working software features for the Bug Prediction dataset and 20 features for the Jureczko dataset. These software features offer relevant information for the defect prediction task as they represent the size and complexity of software classes. We exclude redundant features that have close to zero

occurrences in the target datasets. The replication package of this thesis contains the executed steps alongside the datasets [dos Santos, 2023b].

In the third phase (Machine Learning Models, as shown in Figure 4.1), we focus on implementing machine learning models to predict defects in software projects. Our approach involves applying several models that have been used in various settings of software defect prediction [Nagappan et al., 2006, Menzies et al., 2007, Jing et al., 2014, Wang et al., 2016a, Tantithamthavorn et al., 2017, Tantithamthavorn and Hassan, 2018, Xu et al., 2018, Ferenc et al., 2018], as discussed in Section 2.4. To validate our experiments for each software project, we use a cross-validation process with a set of parameters: the k value, which indicates how many folds we divide the data, and the evaluation metric, which is the F1 score because that is the metric we want to optimize. We apply the standard 10-fold cross-validation (i.e., setting the k to 10). To identify the best-performing software features, we implement a variant of XGBoost [Tantithamthavorn et al., 2017, Levin and Y., 2017]. XGBoost is a scalable machine learning system for tree boosting [Chen and Guestrin, 2016], which is widely used in the machine learning community [Chen and Guestrin, 2016]. In fact, Kaggle, a popular machine learning competition platform, shows that 17 out of their 29 competitions in 2015 had XGBoost as the primary model [Chen and Guestrin, 2016]. We compare the custom XGBoost with baseline models.

To estimate the predictive accuracy of each feature, we quantify all models that include the target software feature. As a result, the predictive accuracy represents the average AUC number of all models that incorporated the specific software feature. The variability represents the average Mean Absolute Deviation (MAD) value of all models that include the target software feature. We apply the Scott-Knott Effect Size Difference (ESD) to test the statistical significance of the results. Finally, in the last phase (Results, as depicted in Figure 4.1), we present the evaluation results of the model. In the following sections, we detail each phase of the machine learning model (i.e., Generation, Classification, and Evaluation).

Model Generation. The precise method to assess the impact of features for defect prediction would require the exhaustive enumeration of all combinations of software features. However, inspecting all subsets of features is computationally restrictive for many reasons. Alternatively, we examine the model space by arbitrarily selecting the software features that compose a specific model. We begin this process by enumerating all possible 1-feature and 2-feature models. Then, we select each of the 2-feature models and include one experimental feature determined uniformly at random to produce 3-feature models. This means that we start with the most performing feature after testing the entire set of available features. Then, we tentatively add the second feature and capture the predictive accuracy. The process goes on until we cover all software features. However, it is a greedy approach as we do not test all permutations, though we carry on with the best perform-

ing. As an example, we continue this step until we generate a model composed of all 37 software features (i.e., for the Bug Prediction dataset). Hence, we create thousands of models. Specifically, we generate 6,377 models for the Bug Prediction dataset and 5,204 for the Jureczko dataset, resulting in 11,581 models. For research purposes, we have made these models available online [dos Santos, 2023a]. In each process, we ensure that each software feature appears an equal number of times. We also have a constraint that no feature can appear twice within the same model. As a result, these models are unbiased as they arbitrarily select the software features from the entire pool of available features. To exemplify the model generation, Algorithm 1 presents a pseudocode of the executed steps. Lines 2 to 7 represent the main loop implemented by the approach (Figure 4.2).

Figure 4.2: Pseudocode Algorithm based on Sampling the Model Space.

Algorithm 1 Sampling the Model Space

Require: training set D
Require: pool of features F
Require: number of candidate models n
Ensure: the most performant model $m(f^*)$
1: $i \leftarrow 0$
2: **while** $i < n$ **do**
3: $\quad i \leftarrow i + 1$
4: $\quad k \leftarrow$ random integer between 1 and $|F|$
5: $\quad f \leftarrow k$ features randomly selected from F
6: $\quad$ compute the predictive accuracy of $m(f)$ in D
7: **end while**
8: $m(f^*) \leftarrow$ model with the highest performance

Source: Elaborated by the author.

Learning to Predict Software Defects: We assume the following definition of software defects. We have as input the training set (referred to as $\mathcal{D}$), which comprises a set of records in the form $< x, y >$, where x is a module represented as a vector of features $x = \{x_1, x_2, \ldots, x_n\}$, in which each x_i encodes a particular characteristic of the module, and y is the corresponding outcome, i.e., whether the corresponding module is defective. The training set is used to construct a model that relates features of the modules to the corresponding outcome. The test set (referred to as $\mathcal{T}$) comprises records $< x, ? >$ for which only the module x is available, while the corresponding outcome y is unknown. The model learned from $\mathcal{D}$ is used to predict the outcomes for modules in $\mathcal{T}$.

Model Classification. The defect prediction features we use may have complex interactions, requiring a versatile classification algorithm. We selected Gradient Boosting Machines (GBM) as our learning algorithm, known for its ability to build an ensemble of

shallow and weak successive trees, with each tree learning and improving on the previous one [Chen and Guestrin, 2016]. Unlike Random Forest, which builds a deep independent tree, the idea behind GBM is to combine weak models into a more robust model (Section 2.4). We produce each model on the errors of the previous models, giving more importance to problematic cases. At each iteration, we compute the errors by adapting a model to these errors. Finally, we find the contribution of each base model to the prior one by minimizing the overall error of the ultimate model. Fitting the base models is computationally challenging, which is why we implemented a high-performance implementation of GBM [Chen and Guestrin, 2016, Tantithamthavorn et al., 2017], referred to as Unbiased Search XGBoost (US-XGB).

Model Evaluation. We tested the effectiveness of the considered machine learning models by applying the standard Area Under the Curve (AUC) and the F1 score, as adopted by the project Caret: Classification and Regression Training [Kuhn, 2015]. These evaluation metrics consider the sensitivity-specificity trade-off [Kuhn and Johnson, 2013]. Essentially, AUC scores an estimate of the probability that a model ranks a randomly chosen defect example higher than a randomly chosen clean example [Sokolova et al., 2006]. F1 score represents the harmonic mean between the precision and recall [Goutte and Gaussier, 2005]. These metrics are sound for imbalanced classes [Ling et al., 2003, Goutte and Gaussier, 2005, Tan et al., 2015] (especially the F1 score), as in the case of our study (approximately 20% of instances are defective for the Bug Prediction dataset and 26% for the Jureczko dataset).

To begin the experimentation with the models, we employed stratified 10-fold cross-validation for each machine learning model. As a result, we partitioned the datasets into ten partitions, out of which we employed nine as training data and the remaining one as the testing set for the classifier. We then replicated the gradual process ten times using each set exactly once as the testing set, thus producing ten results. Hence, the reported AUC and F1 values represent the average over the ten runs. Therefore, the experiments are even and comparable to each other. Finally, to ensure the relevance of the experiments, we assessed the statistical significance of our measurements using the Scott-Knott Effect Difference test [Tantithamthavorn et al., 2017, 2019]. The Scott-Knott Effect Difference test is a non-parametric test employed to assess the statistical significance of the results. The test represents the mean comparison that leverages a hierarchical clustering to partition the set of treatment means into statistically distinct groups with a non-negligible difference [Tantithamthavorn et al., 2019].

4.2 Competitiveness of Baseline Models

In the first set of experiments, we applied machine learning techniques to explore the best-performing models in the target datasets (i.e., Bug Prediction and Jureczko datasets). This step was necessary to measure the predictive capability of the datasets (data quality). To compare the results of the baseline algorithms, we applied the experiments in the five projects from the Bug Prediction dataset and the seven projects in the Jureczko dataset. Besides, we also focused on the XGBoost (XGB) algorithm and the proposed implementation known as Unbiased Search XGBoost (US-XGB). Tables 4.1 and 4.2 show the results of the baseline models for each dataset. The first column of each table presents the name of the model, and the remaining columns are the projects. As we can see, the AUC numbers are not significantly different between US-XGB and the other baseline models. However, the F1 score shows that US-XGB is slightly better at predicting defects for each project (i.e., it varies from 7% to 52% for the Bug Prediction Dataset and 5% to 46% for the Jureczko dataset). To facilitate understanding of the tables, we use bold font to highlight the best evaluation metric (AUC numbers and F1 score) in each project.

In all considered systems, the Unbiased Search (US-XGB) reached similar or slightly superior accuracy when compared with the baseline models (the remaining seven algorithms and classic XGBoost). On average, 4.5% of the machine learning models using the Unbiased Search are superior to the best-performing baseline models. Hence, we notice that using all software features from the power set was never the best choice to predict defects as the accuracy is lower than using the features tested by US-XGB. The high-performing models used a limited set of software features as opposed to the baseline models that applied the entire set (37 and 20 software features for the respective datasets).

Table 4.1: AUC Numbers / F1 Measure Score for the Bug Prediction Dataset.

Baseline Models Performance (AUC/F1)					
Models	JDT	PDE	Equinox	Lucene	Mylyn
LR	0.835/0.498	0.721/0.246	0.806/0.641	0.797/0.466	0.803/0.252
NB	0.797/0.474	0.732/0.361	0.839/0.599	0.795/0.427	0.752/0.365
KNN	0.764/0.535	0.659/0.221	0.815/0.709	0.744/0.235	0.735/0.209
NN	0.727/0.461	0.634/0.253	0.693/0.728	0.437/0.388	0.751/0.322
SVM	0.621/0.420	0.724/0.167	0.688/0.626	0.730/0.218	0.649/0.144
CART	0.697/0.531	0.676/0.284	0.697/0.687	0.686/0.431	0.651/0.295
RF	0.822/0.630	0.757/0.487	0.762/0.755	0.810/0.388	0.760/0.395
XGB	**0.864**/0.671	0.788/0.558	0.843/0.752	0.769/0.401	**0.837**/0.423
US-XGB	**0.863/0.774**	**0.808/0.638**	**0.879/0.815**	**0.828/0.697**	**0.839/0.642**

Source: Elaborated by the author.

To statistically test the soundness of the baseline results shown in Tables 4.1 and 4.2, we applied the Scott-Knott Effect Size Difference (ESD) test [Tantithamthavorn et al., 2017, 2019] in each dataset. In this case, we focus on the AUC numbers as the principal evaluation metric [Kuhn and Johnson, 2013]. Figure 4.3 shows that US-XGB has the lowest treatment means compared to the seven baseline algorithms. Out of the eight predictors used in this experimentation, we located seven clusters in the Scott-Knott Effect Size Difference test for the Bug Prediction dataset (top portion of Figure 4.3). Similarly, for the Jureczko dataset (bottom portion of Figure 4.3), we also discovered seven clusters using the same setting. The best-performing model is an isolated cluster for both evaluated datasets, separated from the other baseline models (Figure 4.3). Therefore, we can conclude that the proposed approach (Section 4.1) is slightly more effective in predicting software defects when compared to the baseline models applied in the literature [D'Ambros et al., 2010, Jureczko and Spinellis, 2010].

Table 4.2: AUC Numbers / F1 Measure Score for the Jureczko Dataset.

Baseline Models Performance (AUC/F1)							
Models	TOMCAT	ANT	LOG4J	PROP	XALAN	CAMEL	JEDIT
LR	0.785	0.722	**0.722**	0.706	**0.668**	0.665	0.807
	0.237	0.642	0.642	0.511	0.622	0.316	0.288
NB	0.781	0.704	0.704	0.677	0.648	0.607	0.771
	0.296	0.556	0.556	0.254	0.411	0.473	0.381
KNN	0.689	0.584	0.584	0.698	0.620	0.633	0.711
	0.170	0.568	0.568	0.216	0.601	0.436	0.296
NN	0.788	0.657	0.601	0.744	0.655	0.613	0.561
	0.541	0.625	0.547	0.516	0.591	0.519	0.541
SVM	0.775	0.487	0.487	0.523	0.561	0.662	0.761
	0.022	0.625	0.625	0.151	0.625	0.167	0.102
CART	0.602	0.566	0.558	0.635	0.570	0.639	0.630
	0.224	0.585	0.591	0.298	0.583	0.498	0.379
RF	0.766	0.592	0.595	0.739	0.611	0.723	0.753
	0.253	0.625	0.626	0.288	0.614	0.459	0.452
XGB	0.776	0.636	0.626	0.777	0.662	0.762	0.821
	0.328	0.631	0.631	0.433	**0.674**	0.414	0.612
US-XGB	**0.859**	**0.731**	**0.715**	**0.889**	**0.665**	**0.802**	**0.836**
	0.687	**0.667**	**0.692**	**0.655**	**0.669**	**0.601**	**0.693**

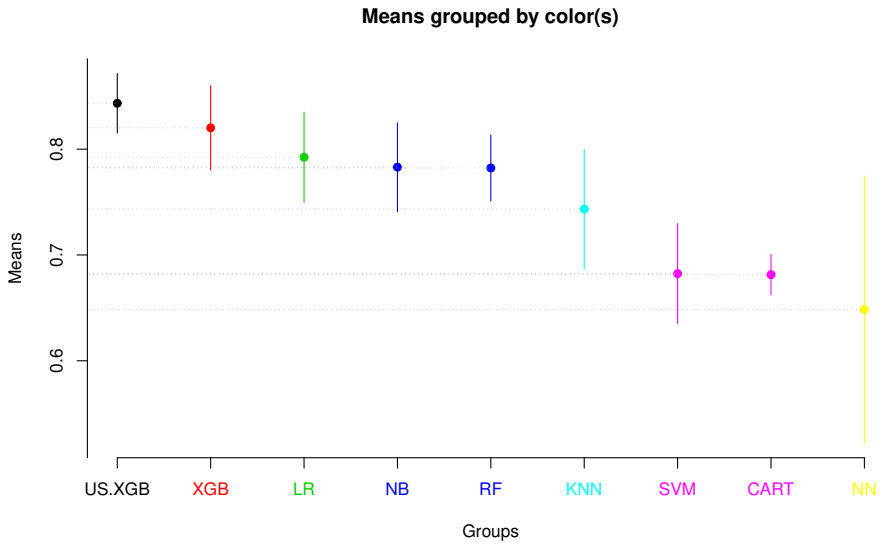
Source: Elaborated by the author.

As an outcome of this experiment, we believe that the superiority of unbiased models is due to two fundamental reasons. First, the use of a flexible tree boosting algorithm [Chen and Guestrin, 2016, Ke et al., 2017]. Second, we employ a particular subset of software features rather than forcing the algorithm to consider all available software features. In summary, our results can serve as a benchmark for future explorations about defect prediction using similar features [Moser et al., 2008]. Therefore, the restricted number of

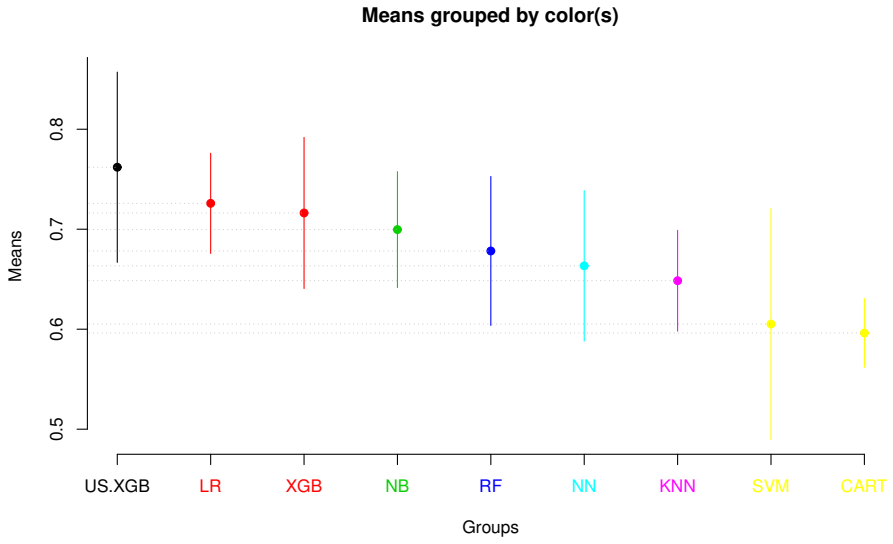
software features adopted in this experiment may suggest that the power of defect prediction features varies among them. We were able to generate accurate models with fewer features than those applicable in the datasets [D'Ambros et al., 2010, Elish and Elish, 2008]. Hereafter, we will compare the predictive accuracy of the machine learning model implementing the ROC curve (AUC). Furthermore, we will focus on the Bug Prediction dataset [D'Ambros et al., 2010] since its overall performance is slightly better (84% and 79% of AUC), and we also have more software features to reason about the predictive accuracy.

Figure 4.3: Scott-Knott Effect Size Estimation Test for the Baseline Models.

(a) Scott-Knott AUC Numbers for The Bug Prediction Dataset



(b) Scott-Knott AUC Numbers for Jureczko Dataset



Source: Elaborated by the author.

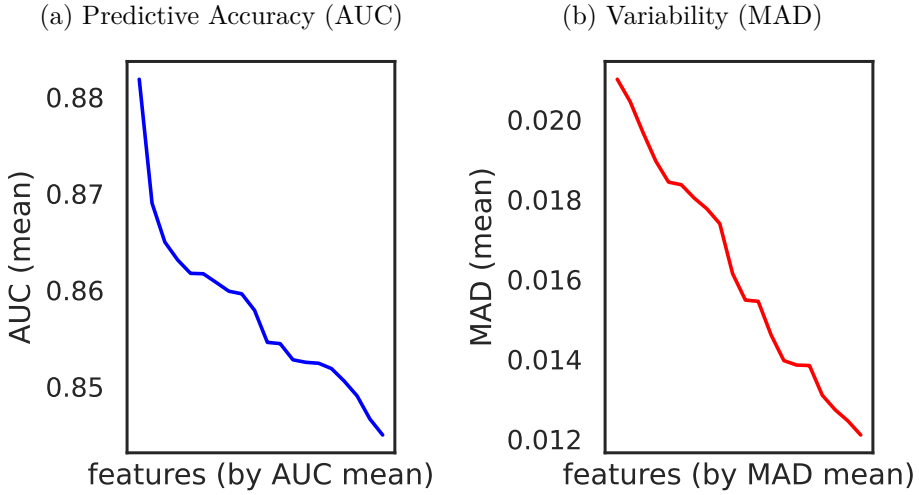
In conclusion, based on the results reported in this section, we have answered the first overarching question. *RQ1. How effective are random models in the defect prediction task?*

> RQ1. We conclude that the random model discussed in this chapter named Unbiased Search XGBoost slightly outperforms baseline machine learning models for the defect prediction task.

4.3 Predictive Accuracy of Software Features

This section evaluates the accuracy and variability of software features generated from the top 10% machine learning models in terms of performance. We focus on this subset of models as they yield better results in terms of both accuracy and variability. Once again, the US-XGB generated thousands of models. Figure 4.4 shows the predictive accuracy and variability of software features using US-XGB. Only a few features delivered superior predictive accuracy. Specifically, around 4% of the features are part of models where the average AUC numbers are higher than 0.83. The unbiased models associated most software features with significantly lower average AUC numbers (around 70%). We observe a similar trend when investigating the distribution of software features in terms of variability. Nearly 2.5% of the considered features relate to models with relatively low variability.

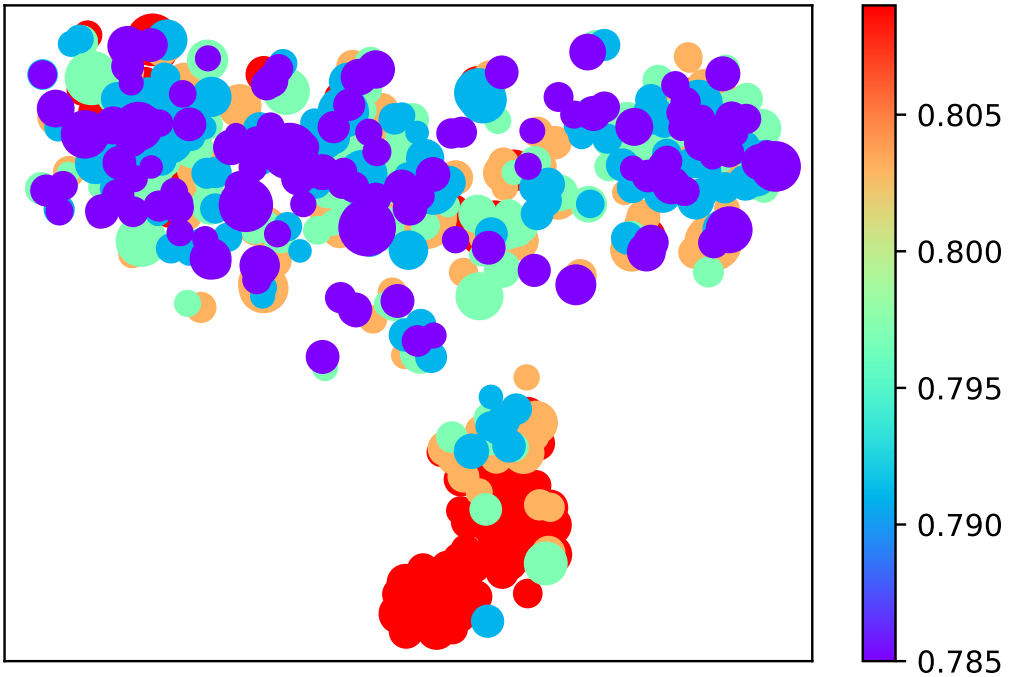
Figure 4.4: Distribution of Features in the Generated Models.



Source: Elaborated by the author.

In Figure 4.5, each dot's diameter (size) corresponds to the ratio between the model's AUC value and corresponding variation (MAD). Thus, each point represents a machine learning model. The color indicates AUC numbers (on a scale of 5), and size symbolizes variability (MAD), with a higher size implying lower variability. We gather the coordinates of each dot from the vector of probabilities assigned by the model to each defect expectation case. Thus, Figure 4.5 presents a scatter plot of the arrangement of the probabilities. To compute this plot, we take these probabilities as a vector and calculate their 2-dimensional representation using t-SNE [Maaten and Hinton, 2008]. This process is known as dimension reduction [Maaten and Hinton, 2008]. We conclude that the AUC numbers are between 0.785 and 0.835. We observe the proximity of a few models with high performance on average (red dots, AUC $>$ 0.82) and low variability. However, some models represented lower variability than the best-performing models, although with lower AUC numbers. To properly interpret the correlation between features and machine learning model performance, we employ the best-performing models and estimate the predominance of features. To clarify the relationship between software features and accuracy (AUC) and variability (MAD), we analyze the top 10% with the highest accuracy and the 10% with the lowest variability and compute the predominance of features in these clusters.

Figure 4.5: Model Predictive Accuracy and Variability.



Source: Elaborated by the author.

US-XGB Highest Accuracy: As we take into consideration models with the highest AUC numbers, we note that features extracted from the change features are more frequent

(47%), compared to entropy (27%) and class-level (26%). Among those features, the most significant occurrence was the AGE feature (specifically the segment related to the age itself and not the weighted age - wAGE). This software feature appeared in around 21% of the top 10% models representing an occurrence on 1 out of 5 models generated by the unbiased search. Closing the top 3 features from the generated models, two software features appeared in 8% of the models each: the History Complexity Measure (HCM) and Weighted History Complexity Measure (WHCM).

US-XGB Lowest Variability: Features from the class-level are the most common among those with the lowest variability, representing around 50% of the data. They include features such as Number of Attributes (NOA), Number of Methods Inherited (NOMI), and Number of Methods (NOM). Change metric features account for approximately 40% of the low variability cluster, while entropy-related features make up only 10%, such as the Weighted History Complexity Measure (WHCM). It is intriguing to note that Weighted Age (wAGE) appeared in nearly 14% of the models, Depth of Inheritance Tree (DIT) in 13%, and Number of Attributes Inherited (NOAI) in 10%. In summary, we conclude that many combinations of features can yield models with high performance and low variability.

To summarize, the results presented in this section support the second overarching question of the investigation. *RQ2. Which software features are more effective in predicting defects?*

RQ2. We conclude that the software features extracted from the change features are more effective in predicting defects. Furthermore, the AGE (i.e., the age of a class in weeks) feature is the most significant feature in the top 10% models with the highest accuracy.

4.4 Implications for Practitioners

This section introduces the main implications of our study for project managers and developers. This investigation can serve as a benchmark for future research on defect prediction using similar models. We discuss these implications next.

(i) The proposed machine learning model could aid in the development of a tool to classify the likelihood of a Java class being defective. This tool could load pre-

existing models that yield high accuracy and attempt to achieve consistent results with the data provided by the Java project. The only limitation to using pre-existing machine learning models relates to the software features. Therefore, it is necessary to collect the features before making a prediction.

(ii) The results suggest that change features are more likely to result in higher predictive accuracy than other features (such as entropy or class-level features). Therefore, the development team could focus on change features.

(iii) Our results indicate that explaining software defects is a project-specific task. Thus, practitioners should not expect the same models to explain defect predictions for different projects. We recommend training machine learning models within the same project. One technique to split the data could involve using pre- and post-release data. By doing so, the model could learn from pre-release data and predict the post-release behavior based on its features.

4.5 Threats to Validity

This section investigates some limitations that could potentially threaten our results. First, we discuss the external threats to validity. Then, we review the internal threats to validity. Next, we examine the construct threats to validity. Finally, we present the conclusion and its threats to validity.

External Validity: Threats to external validity are conditions that limit our ability to generalize the results of our research [Wohlin et al., 2012]. In our investigation, a threat to external validity relates to the limited number of projects we considered (only twelve projects across both datasets) [Jureczko and Spinellis, 2010, D'Ambros et al., 2010]. Moreover, all projects are related to the Java programming language. Therefore, the conclusions may not generalize to projects developed in different programming languages. Another external threat relates to the fact that our results depend on defects within the project context. Hence, we could not draw any conclusion about cross-project defects. Additionally, another threat relates to the limited number of baseline algorithms we used to classify a software module as defective. Thus, we could not guarantee that our results generalize to all existing classification algorithms. Finally, although we used a large set of software features (57 across both datasets), all the software features are related to Object-Oriented Programming design [Jureczko and Spinellis, 2010, D'Ambros et al., 2010]. Therefore, we cannot guarantee that the model will perform well for different soft-

ware features.

Internal Validity: Threats to internal validity refer to factors that could affect the relationship between the independent variable and causality [Wohlin et al., 2012]. In our study, this threat is related to the datasets we used, as we relied on the data provided by the Bug Prediction dataset [D'Ambros et al., 2010] and the Jureczko dataset [Jureczko and Madeyski, 2010, Jureczko and Spinellis, 2010] without being able to verify their accuracy and completeness. As a result, the data may contain errors or omissions that could affect our results. To address the issue of imbalanced data, we applied machine learning techniques such as XGBoost, which is known to handle imbalanced data well. We used two evaluation metrics, ROC Curve (AUC) and F1, to assess the performance of our models. However, we cannot guarantee that our results accurately reflect the nature of the twelve Java projects included in our study. Additionally, our data represent a snapshot of the projects at a specific point in time.

Construct Validity: Construct validity concerns the assumptions linking the results of the experiments to the underlying concepts or theories [Wohlin et al., 2012]. In our context, we evaluate the models using AUC and F1 score. Although we tested other metrics such as accuracy and F1 includes precision and recall, other metrics in the literature could be used to assess the effectiveness of the generated models. For instance, we could employ the Matthews correlation coefficient [Yao and Shepperd, 2020] and the Kappa coefficient [Cohen, 1960]. Therefore, we cannot guarantee the models' performance with these additional metrics.

Conclusion Validity: Threats to the conclusion validity are related to issues that affect the ability to draw the correct conclusion about the relationship between the treatment and the outcome [Wohlin et al., 2012]. The model's predictive ability depends on the defect labels of the Bug Prediction dataset [D'Ambros et al., 2010] and the Jureczko dataset [Jureczko and Madeyski, 2010]. Other studies [Yatish et al., 2019] have found that many datasets rely on a six-month post-release window to predict defects effectively, as opposed to using affected releases of issue reports, such as those used in the target data [D'Ambros et al., 2010]. Therefore, since we are not dealing with the post-release window, the model may not generalize to these cases [Yatish et al., 2019].

4.6 Final Remarks

In conclusion, the machine learning models presented in this chapter are effective for defect prediction in software development based on two popular datasets. First, the proposed implementation of gradient boosting machines reveals how difficult it is to detect software defects, as only a small fraction of the machine learning models (4.5%) achieved a detection performance higher than 83% based on the AUC numbers. We hope that our efforts can serve as a baseline for other solutions to defect prediction using Java projects. Second, we showed how a limited set of features produced high AUC numbers. Third, the collection of dominant software features for the defect prediction task is variable. However, features related to change features are more prominent in the top-performing machine learning models. To reach these results, we employed an algorithm (Figure 4.2) to explore the space of all software features. The proposed approach (US-XGB) resulted in thousands of random models. We then evaluated these models considering their accuracy and variability. By implementing machine learning-based defect prediction techniques, organizations may improve the quality of their software products, resulting in improved customer satisfaction and a stronger reputation for producing high-quality software [Menzies et al., 2007, Turhan et al., 2009]. Based on the results of this chapter, we have created a replication package that may assist future explorations of software features [dos Santos, 2023b], as discussed in Section 4.4. In the next chapter, we discuss the results of applying understandability concepts to reason about the software features found in this chapter.

Chapter 5

Understanding Software Defect Models

Understanding machine learning models for defect prediction is crucial because it allows us to identify potential software defects before release [Jiarpakdee et al., 2020]. By analyzing patterns in past defects, we can create models that predict which parts of a software system are most likely to contain defects, helping us prioritize testing and development efforts [Pornprasit et al., 2021]. This chapter presents the results of understanding software defect models, building on the machine learning model exploration and predictive accuracy presented in Chapter 4. However, rather than solely focusing on model accuracy, this chapter aims to understand the defects reported in the target datasets [D'Ambros et al., 2010, Jureczko and Madeyski, 2010, Jureczko and Spinellis, 2010, Ferenc et al., 2018]. We employ the SHAP (Shapley Additive exPlanations) technique to explain the machine learning model decisions for all three datasets [Lundberg and Lee, 2017, Lundberg et al., 2018a]. SHAP is a game-theoretic approach that connects optimal credit allocation with local explanations, allowing us to reason about the model decision and how target software features influence these decisions (i.e., predicting whether a software class is defective or not).

The remainder of this chapter is divided into seven sections. Section 5.1 provides context on SHAP values and the feature importance of machine learning models. In Section 5.2, we analyze the results of the SHAP analysis over the machine learning models presented in Chapter 4, examining each dataset separately as they do not share the same software features (as discussed in Chapter 3). Section 5.3 investigates the threshold of software features. Section 5.4 discusses the main results of the previous sections. Section 5.5 presents the implications of this study for practitioners and the defect prediction research community. Section 5.6 identifies the main threats to the validity of the reported results, and finally, Section 5.7 concludes the chapter with a summary of the findings.

5.1 Feature Importance and Explanation

Effective machine learning models often produce complex predictions that are difficult to explain. As a result, understanding why a model has made a specific prediction is a central challenge in software defect prediction [Jiang et al., 2013, Lewis et al., 2013, Yatish et al., 2019]. For instance, by knowing that the complexity of a software class is a factor that influences the defectiveness of the target class, the software team may modify that class to address its complexity [Tantithamthavorn et al., 2015, Jiarpakdee et al., 2020].

The typical approach for understanding defect prediction is based on the calculation of feature impact, or feature importance. In current literature, feature importance is defined as an increased error after permuting feature values. Permutation breaks the relationship between the feature and the outcome [Lundberg and Lee, 2017, Lundberg et al., 2018a]. Therefore, a software feature is relevant if permuting its value increases the model error because the model relied on that feature for the prediction [Jiarpakdee et al., 2020, Lundberg et al., 2020]. Conversely, a feature has little importance if permuting its values keeps the model error unchanged because the model ignored that feature for the prediction. Often, software features interact in many different ways to create models that provide accurate predictions. Thus, feature importance represents a function of the interaction between other software features. In this case, we use Shapley values [Shapley, 1953] to find a fair division design that defines how we can distribute the total importance among features.

More specifically, we transform instances into a space of simplified binary features, and the explanation model g is a linear function of binary variables:

$$g(z) = \phi_0 + \sum_{i=1}^{m} \phi_i \times z_i, \tag{5.1}$$

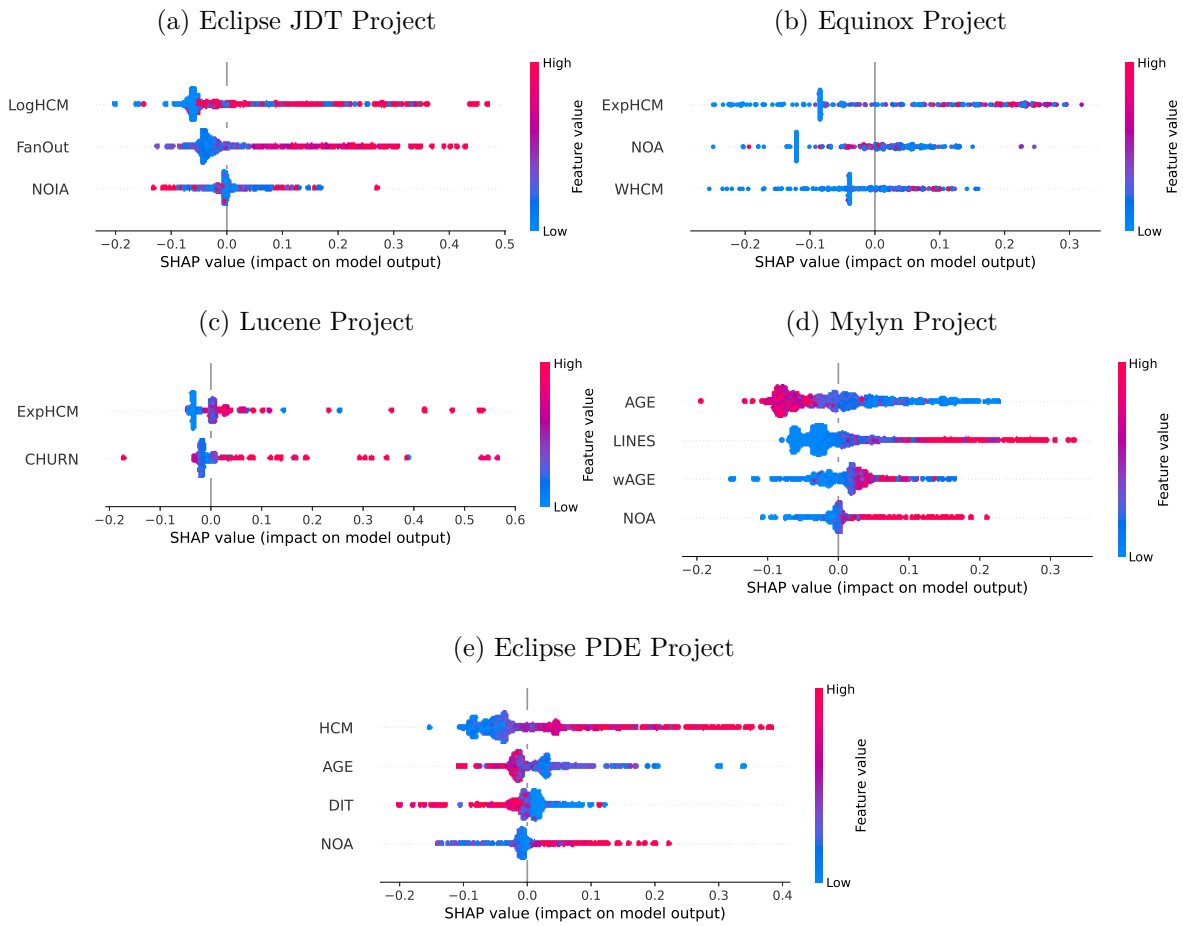
Where ϕ_i for $i = 0, 1, \ldots, m$ are parameters called Shapley values, m is the number of simplified input features, $z_i = \{z_1, z_2, \ldots, z_m\}$ is a binary vector in simplified input space where $z \in \{0, 1\}^m$. Shapley values measure how each feature contributes to the prediction. In fact, Shapley values are theoretically optimal and are unique consistent, and locally accurate attribution values. In this work, we use SHAP (SHapley Addictive exPlanation) values [Lundberg and Lee, 2017, Lundberg et al., 2018b, 2020] as an approximation of Shapley values to compute the importance of each feature in the prediction.

5.2 Model Understandability

In our exploration of model understandability, we investigate the models generated by US-XGB (previously discussed in the previous chapter). We assume that models with fewer features are more understandable than the ones trained on the entire set of features. We use SHapley Additive exPlanation (SHAP) [Lundberg and Lee, 2017, Lundberg et al., 2018a, 2020] to explain why the model made a specific prediction. SHAP assigns an importance value (positive or negative) to each feature in a particular defect prediction. The output value comprises the sum of the base value (average defect prediction over the validation set) and these important values. The SHAP value is relevant because we could find reasonable accuracy numbers with the unbiased algorithm. Besides, SHAP allows us to summarize important software features. Hence, we associate low and high feature values with an increase/decrease in output values, through color-coded violin plots built from all predictions. The color red shows high numbers, and blue shows low numbers. In this case, a high value indicates that a model predicts a defect in the class, while a low value registers a clean class. For these experiments, we present the results of applying SHAP to the target datasets (i.e., the Bug Prediction dataset, the Jureczko dataset, and the Unified dataset) [Jureczko and Madeyski, 2010, Jureczko and Spinellis, 2010, D'Ambros et al., 2010].

Figure 5.1 shows SHAP summary plots associated with the most superior models for the selected projects (Eclipse JDT and PDE, Mylyn, Equinox, and Lucene) in the Bug Prediction dataset. Figure 5.1 has a vertical line aligning the values that appear along the right, which are contributing to increase the odds of a defect. Values in red indicate a high influence on the defect prediction, while numbers in blue indicate less influence on the defects. Additionally, values appearing on the left side lead to decreasing the probability of a software defect. Hence, this reveals that for the Eclipse JDT project (bottom right), higher *fanOut* values increase the chances of the model predicting defects. In contrast, for the Mylyn project, second from the top left (d), the higher the *age* of the project, the lower the predicted defects in the project. Note that we explain model decisions for the Lucene project with only two features. Two of the selected projects used three software features (Eclipse JDT and Equinox), and the remaining two projects used four software features (Mylyn and Eclipse PDE). We also remark that important features may vary depending on the project. Some of the most relevant software features to understand a defect derived from entropy level features, such as the History Complexity Measure (HCM) and its variations (for example, the log and exponential versions of this feature). Features related to change features and class-level source features are also frequent in the target projects. For instance, the AGE of a software class in weeks is a feature that is relevant to the defect prediction of the Mylyn project.

Figure 5.1: SHAP Values that Influence Defects in the Bug Prediction Dataset.

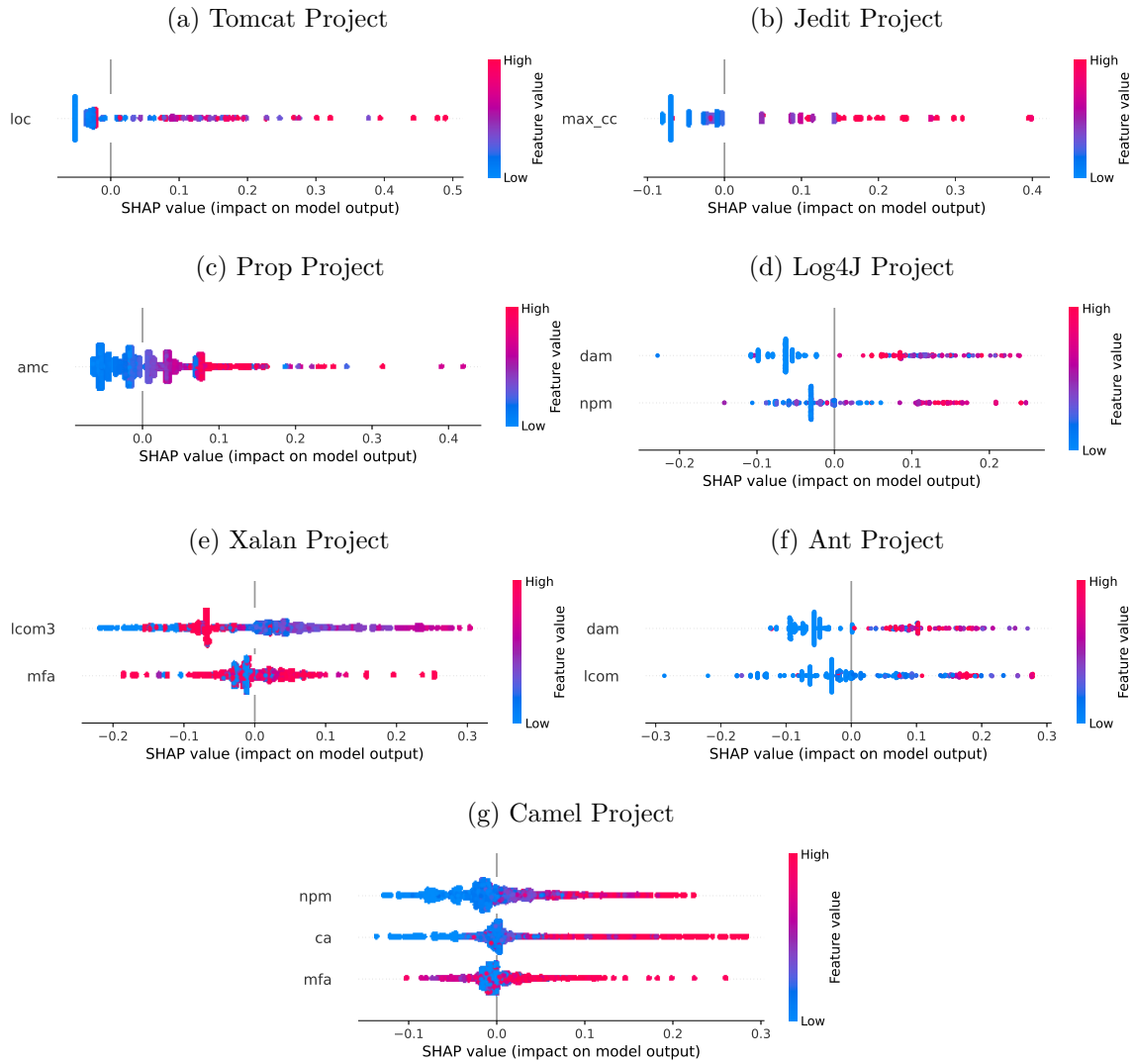


Source: Elaborated by the author.

Figure 5.2 shows the same experiments for the Jureczko dataset. Here, we note that the model demands up to only three features to understand the defects in the selected data. Hence, this reveals that for the tomcat project (Figure 5.2 top-left plot), higher Lines of Code (LOC) values increase the chance of our model predicting defects in that specific project. Model decisions for the Tomcat, Jedit, and Prop projects are explained with solely one feature. Four of the selected projects used only two software features (Log4J, Xalan, Ant, and Lucene), and the remaining project used three software features (Camel). We also remark that important features may vary depending on the project. Some of the most relevant features derived were LOC, AMC (Average Method Complexity), Data Access Metric (DAM), Response for a Class (RFC), and Number of Public Methods (NPM).

These results indicate that searching the feature space is not only able to slightly improve the AUC numbers but also to generate models with fewer features, thus making them more explainable (Figures 5.1 and 5.2). We produce models composed of up to only four features for the Bug Prediction dataset and three software features for the Jureczko dataset. Therefore, these models are simpler than the models generated for exploring the entire feature space. We argue that machine learning models composed of fewer software

Figure 5.2: SHAP Values that Influence Defects in the Jureczko Dataset.



Source: Elaborated by the author.

features are more explainable. Therefore, if a developer received the model explanation obtained from Figures 5.1 and 5.2, it would be easier for them to work on the features that have a high probability of causing defects. For example, a Mylyn developer could use our results to acknowledge that the age of a class (Figure 5.1d) contributes to the defects in that project. Thus, the developer could give more attention to software classes developed a long time ago.

5.3 Impact of Software Features

To confirm the hypothesis that smaller software feature values reduce the chance of having classes with defects, we designed an experiment to measure the limit of selected software features. For this purpose, we required a dataset with a wide range of software features. Therefore, we used the Unified dataset, which offers a broad set of software features (71 in total) that could help us visualize the contributions of all software features for a specific software class [Ferenc et al., 2018, 2020a] (Section 3.4). However, as most of the software features are not important to the model, we had to select the most relevant software features for the defect prediction task. Thus, we could find the software features for the defect prediction task and further threshold analysis. The main objective of our investigation was to examine the threshold of software features applied for the defect prediction task. Therefore, this section explores the following research question: *RQ. What is the limit value of software features that prevent the class from being defective?*

In this section, we present the results of applying the Unified dataset to measure the threshold of software features. First, we demonstrate the results of selecting the relevant software features for model understandability [Lundberg et al., 2020]. Here, we employed a recent implementation of a tree-based algorithm to choose the software features [Ke et al., 2017]. Second, we describe the results of the experiment to measure the threshold of software features based on Lundberg et al. [2018b]. These results indicate that developers should keep software features as small as possible to avoid defects in Java classes.

Selected Software Features. The current literature incorporates many software features to predict defects [Menzies et al., 2007, Zimmermann and Nagappan, 2008, Jing et al., 2014, Pornprasit et al., 2021]. In the Unified dataset (Section 3.4), we have 71 software features available to predict defects. We experimented with several methods to choose the optimal software features, and a recent implementation of the Gradient Boosting Machine (LightGBM, [Ke et al., 2017]) achieved better results compared to other baseline models. This technique automatically selects the most relevant features for predictive modeling, resulting in fourteen software features that we use to train the models. Although we apply all fourteen features to build the models, we individually evaluated each of them to measure their impact [dos Santos, 2023b].

Table 5.1 shows the fourteen software features selected by the tree-based algorithm. The first column shows the identifier of each feature, while the second column presents its acronym. The third column displays the feature's description, and the fourth column represents the quality attribute it belongs to. All features are class-level due to their scope. Most features relate to size (NA, NG, NLG, NS, and TNM) [Ferenc et al.,

2018, 2020b], while some are associated with documentation (CLOC, CK, AD, PUA), code coupling (CBOI, NOI), and code complexity (CC, WMC, NL) [Tóth et al., 2016, Ferenc et al., 2020a].

Table 5.1: Selected Software Features Unified Dataset.

	Acronym	Description	Quality Attribute
1	WMC	Weighted Method per Class	Complexity
2	NL	Nesting Level	Complexity
3	CC	Clone Coverage	Complexity
4	CBOI	Coupling Between Objects classes Inverted	Coupling
5	NOI	Number of Outgoing Invocations	Coupling
6	AD	API Documentation	Documentation
7	CD	Comment Density	Documentation
8	CLOC	Comment Lines of Code	Documentation
9	PUA	Public Undocumented API	Documentation
10	NA	Number of private attributes	Size
11	NG	Number of getters	Size
12	NLG	Number of local getters	Size
13	NS	Number of setters	Size
14	TNM	Total Number of Methods	Size

Source: Elaborated by the author.

Threshold of Software Features. We apply a unified model-agnostic approach known as SHAP [Lundberg and Lee, 2017] to explain the model decisions. This technique connects game theory and local explanations to interpret the output of machine learning models (using accuracy). Among the many possibilities that SHAP provides to explain a machine learning model, we focus on understanding how each feature contributes to pushing the model decision to a defective class. To be more specific, we follow the experimentation of Lundberg et al. [2018b], where we are able to visualize the relative contributions of all software features for a specific class. The replication package of this thesis contains the code to generate the plots presented in this section [dos Santos, 2023b]. Therefore, we discuss the impact of the selected fourteen software features on the explainability of the machine learning model. Furthermore, the order of importance of each feature is CLOC, CD, AD, NOI, WMC, CBOI, TNM, NL, NA, CC, NG, NLG, NS, and PUA.

1. **Weighted Method per Class (WMC)**: The machine learning model predicted that when the weighted methods per class are up to 35, the model has a 42% in predicting a defect. Higher numbers contribute consistently to increase the probability of predicting a defect.

2. **Nesting Level (NL)**: The model predicted that when the nesting level (which is counted by ifs structures [Ferenc et al., 2018]) is higher than 57, the model is more

likely to predict defects (nearly 71% of a chance in predicting a software defect in the source code). Low numbers contributed consistently less to find a software defect. For instance, when NL is 2, the model could predict a defect with only around a 20% chance.

3. **Clone Coverage (CC)**: If the clone coverage is up to 11%, the model predicted a defect with a 40% chance. In the case of clone coverage higher than 11%, the model predicts a software defect with only 21%. In this case, a higher number of clone coverage helped the model to predict a clean class.

4. **Coupling Between Objects classes Inverted (CBOI)**: For the number of coupling between objects inverted between 410 and 450, the model predicts a defect with a 70% chance. Higher numbers help the model to stabilize (dropping to close to 40%). Lower numbers (around 77) help the model in predicting a software defect with only a 20% chance.

5. **Number of Outgoing Invocations (NOI)**: A high number of outgoing invocations is considered above 22, and helps the model to predict defects (up to 70% chance of predicting a defective class). The limit to predict clean classes happens up to only twelve outgoing invocations (up to only 21% chance of being defective).

6. **API Documentation (AD)**: If the API documentation is around 98%, the model predicts a defect with a 69% chance. For low numbers of API documentation, such as 2%, the model predicted a defective class with only 29%. One argument for this finding is that documented APIs are from large source codes from more complex software systems. Thus, they may generate more defects by definition.

7. **Comment Density (CD)**: When the comment density is up to 47%, the model has a 49% to predict a defect (highest value). When the comment density numbers are low, for example, up to 3% helped the model very little in predicting a software defect (up to around 15%). Higher numbers affected less the prediction (around 30-40% to predict a software defect).

8. **Comment Lines of Code (CLOC)**: The peak of our prediction happens around 850 lines of comments in a class, where the model can predict a defective software class with a 78% chance. Lower values help the model less (between only 30% only). This finding may indicate that larger software classes are defective-prone. Hence, they have hundreds of lines of code and comment [Fowler, 1999].

9. **Public Undocumented API (PUA)**: For up to 284 lines of undocumented API lines of code, the model predicts a defect with a considerable chance (67%). If the lines of undocumented API lines are low such as only 12 lines, the model predicted

a software defect with only 17% (50% down the chance in predicting a software defect).

10. **Number of private Attributes (NA)**: Up to 66 attributes in a class, the model predicted a defect with approximately a 52% chance. Values higher than 66 attributes consistently increase the prediction of defects in the source code (up to 76% chance of predicting a defective class). Only up to 22 software attributes per class made the model predict a defect with a low possibility (around 22%).

11. **Number of Getters (NG)**: For up to 80 getters in a class, the model predicts a defect with a 77% chance. Thus, higher values than 80 do not increase the prediction of a software defect (stabilizes between 50-65%). Low numbers, up to 6 getters in a class, helped the model to predict a defect with only 23%.

12. **Numbers of Local Getters (NLG)**: The analysis is very similar to NG as if the number of local getters is up to 70. It helps the model consistently to predict a defect (around 68%). Low numbers of local getters decreased the possibility to find a software defect (up to 6 getters, only 22% to find a software defect in the class).

13. **Number of Setters (NS)**: In the case of the number of setters in the code, for up to 60-70 setters in the class, the model has a 70% to predict a defect. However, low values than 50 (for instance, between 10 and 30) decrease the chance of predicting a defect to around 40%.

14. **Total Number of Methods (TNM)**: For up to 46 methods per class, there is only a 36% chance of being a defective class. More than 46 methods consistently increase the possibility of a defect (up to 77% chance in predicting a software defect). Our study suggests that the number of software methods should not exceed 46.

From these findings analyzing each software feature, we could answer the research question. *RQ. What is the limit value of software features that prevent the class from being defective?*

> The threshold varies considerably among the target software features used in this work. For instance, developers should keep WMC around 35, and CBOI should not exceed 77 to help the class maintaining a clean state.

Using the explanations provided by SHAP, we could understand the limit of several software features for defect prediction. In the end, we could rank the quality attributes according to their impact: i. Documentation, ii. Size, iii. Complexity, and iv. Coupling. To the best of our knowledge, this is a unique use of SHAP [Lundberg et al., 2018b], as it was used to define the threshold of software features for defect prediction. As a result,

these thresholds may help developers to reason about the impact of software features on code defectiveness. Therefore, they can improve the quality of the code, tracking the features that introduced more defects in the source code.

5.4 Discussion

This section discusses our results and their impact on the software engineering community. First, the results of this work indicate that different projects have different features that may help the model predict defects. For example, the *Tomcat* project has a high number of lines of code per class, which is a feature that aids in predicting defects (i.e., LOC). Second, the results of this work are consistent with previous studies. For instance, Ferenc et al. [2018] found that the Total Number of Methods (TNM) is a good predictor of software defects. However, they did not identify a threshold for the TNM. In this work, we found that the TNM should not exceed 46, as higher values consistently increase the model's prediction of defects. We can extend this analysis to all fourteen software features discussed in Section 5.3. To the best of our knowledge, the identification of the thresholds for the software features is unique in the literature. This analysis can help developers reason about the impact of software features on code defectiveness and improve the code's quality by tracking the features that introduce more defects in the source code. Finally, although we have a wide range of software features to measure various aspects of the source code, only a limited number of features aid in predicting defects. Therefore, we can focus on these features to improve the code's quality. Moreover, as data collection is an expensive process [Menzies et al., 2007, Jureczko and Madeyski, 2010], the research community can focus on important features and ignore the ones that do not seem to indicate defects. Ultimately, this approach can reduce the cost of gathering data for defect prediction.

5.5 Implications for Practitioners

This section discusses the fundamental implications of our study for practitioners. We believe that the proposed investigation can serve as a guide for the research community to understand the potential of SHAP values for defect prediction.

(i) We found that a limited set of software features are relevant to predict defects in Java (considering that the datasets have 37 and 20 software features, respectively). Therefore, we do not need to concentrate on most of the features presented in the datasets to build explainable models for defect prediction. Consequently, we could use these models to propose a tool to predict whether a software class is likely to cause defects or not.

(ii) The SHAP explanations seemed to indicate that for a variety of software features (e.g., LOC, AGE, HMC, NOA, AMC, MFA), the higher the feature value, the higher the probability of defect prediction (as shown in Figures 5.1 and 5.2). The threshold analysis confirms that most software features should have small values to avoid defects.

(iii) Our analysis using SHAP values revealed that the software feature AGE, which represents the age of a class in weeks, has the highest impact on model accuracy compared to other features (as discussed in Chapter 4). This finding is particularly relevant for developers, as they can use this knowledge to prioritize and take special care of classes that have been developed a long time ago, which may be more prone to defects.

(iv) One of the main findings is related to the impact of each selected software feature. The results discussed in Section 5.3 could assist developers to mitigate software defects in Java projects. The thresholds of these features are publicly available in the replication package for further explorations of these software features by the research community in defect prediction.

5.6 Threats to Validity

This section presents potential threats to the validity of this study that could potentially impact our results. First, we discuss external threats to validity, followed by internal threats to validity. Next, we investigate construct threats to validity. Finally, we examine conclusion threats to validity.

External Validity: Wohlin et al. [2012] defines threats to external validity as conditions that limit our capacity to generalize the results of the research. The current literature accepts SHAP values as a method to explain machine learning models [Lundberg et al., 2018a]. However, other approaches may have different explanations based on a series

of characteristics. For instance, current literature considered techniques such as LIME [Mori and Uchihira, 2018] and BreakDown [Jiarpakdee et al., 2020] to understand defect models. At this moment, we cannot guarantee that the results are replicable using these tools. For this reason, we may use these mechanisms (i.e., Lime and BreakDown) in the concluding stage of this project to compare the results presented in this chapter [Mori and Uchihira, 2018, Jiarpakdee et al., 2020].

Internal Validity: According to Wohlin et al. [2012], threats to internal validity refer to rules that can influence the independent variable to causalities. Our explanations depend upon the defect labels of the Bug Prediction dataset [D'Ambros et al., 2010] and the Jureczko dataset [Jureczko and Madeyski, 2010, Jureczko and Spinellis, 2010]. However, we could not validate how the authors gathered the data available in the datasets. We only know that the Jureczko dataset uses a regular expression to extract the defect labels, and the Bug Prediction dataset applies an approach based on bug reports. More importantly, we cannot guarantee that the defects are correctly labeled. For instance, the authors may have categorized the software defects in the Jureczko dataset as "false positive" or "false-negative" instead of an actual software defect found during development.

Construct Validity: Wohlin et al. [2012] describes the construct threats to validity as the ability to assume the results of the experiments to the concept or theory. In this case, this threat correlates to the limited number of projects we investigated (twelve in total counting both datasets) and the use of only one programming language (i.e., Java) [Jureczko and Spinellis, 2010, D'Ambros et al., 2010]. Additionally, the threshold analysis has the limitation of only using the Java programming language. As a result, we cannot guarantee that the models generalize well for other programming languages different from Java, such as popular interpreted programming languages (e.g., Python and JavaScript). Further explorations should focus on the use of multiple programming languages from different projects. Moreover, it would be better to use a larger dataset with the same software features to compare the results. In this case, we did not compare the results of the two datasets because they do not share all the same software features. Another threat relates to the fact that although the current literature recognizes SHAP as a method to explain models, this technique is not the only option to understand these machine learning models. For instance, the current literature offers a set of other explainability techniques for this matter. As an example, LIME [Ribeiro et al., 2016] and InterpretML [Nori et al., 2019] are tools to explain any black-box machine learning model.

Conclusion Validity: According to Wohlin et al. [2012], a threat to conclusion validity refers to the study's ability to connect issues that affect the ability to express the correct conclusion between the treatment and the outcome. In this investigation, this threat

relates to how developers could use these software features to act upon their defective software code. We assume these software features are relevant to understanding the defects practitioners may find during development. However, we cannot conclude to what extent these software features are relevant to the development process. To validate these software features, we need to conduct new explorations with practitioners aiming at evaluating the usefulness of these features for the development process. Another solution would involve categorizing these features into categories that are more actionable by developers.

5.7 Final Remarks

This chapter concludes our investigation into the use of SHAP values for understanding defect models. We used SHAP explanations to understand model decisions and found that the best-performing models are simple to understand, as they use only a few features from the power-set (complementary to Chapter 4 results). Furthermore, we employed one of the many possibilities that SHAP offers to determine the threshold of software features for defect prediction. To the best of our knowledge, this is a unique use of SHAP [Lundberg et al., 2018b], as it was used to define the threshold of software features for defect prediction. In summary, our findings suggest that we can identify software defects with determined combinations of features. More importantly, the results display how difficult it is to generate a unique solution to understand defect models. Independent projects are subject to distinct software features that may cause software defects. In addition, we discovered that a limited set of software features is relevant to understanding software defects in Java. The SHAP explanations also suggest that a variety of software features (e.g., LOC, AGE, HMC, NOA, AMC, MFA) tend to lead to a higher probability of defect prediction if the feature value is high. Complementarily, the threshold analysis offered insights into the limit of these values for defect prediction. Overall, most features should have small values to decrease the probability of the model finding a defect in the class. Therefore, our findings could assist in the implementation of a tool to predict defects in Java projects based on the software features examined in this chapter. In the next chapter, we discuss the results of analyzing the developers' perceptions about SHAP explanations using a similar setting.

Chapter 6

Developers Perception

Understanding predictive models of software defects is important because this understanding may assist developers in finding the classes with defects in their source code. For this reason, this chapter presents the results of two survey studies with developers. The first study focuses on understanding models through the use of SHAP [Lundberg and Lee, 2017]. Our main goal is to check whether or not SHAP explanations are valuable for developers. In this case, we evaluate the results presented in Chapter 5. To do so, we employ an exploratory study using a survey with forty developers. The developers compose a mixed group with various backgrounds, including software engineers, data scientists, and researchers. Additionally, we want to validate how developers could act upon the SHAP explanations knowing that certain software features impact the software defects. The second study focuses on developers' perception of the quality attributes that may contribute to defective code. In this study, we invited developers to evaluate different scenarios that employ a set of quality attributes. We selected active developers from GitHub to participate in the survey, allowing us to survey developers from different parts of the world, not just limited to Brazil. Finally, we questioned developers about which static software features they perceived as defect predictors.

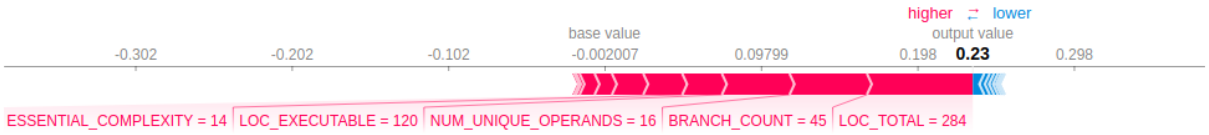
We divide the remainder of this chapter into five sections. First, we present the results of the first survey about SHAP explanations (Section 6.1). In this case, we introduce the survey design with the main steps executed to investigate the developers' perceptions. Then, we present the developers' backgrounds. Later, we discuss the results, focusing on the developers' perception of SHAP explanations. Section 6.2 describes the second survey study about developers' perception of quality attributes. We present the survey design and the background of the participants, then discuss the results, focusing on the developers' perception of the quality attributes. Section 6.3 explores the main implications of our findings for the defect prediction community. Section 6.4 presents the main threats to these studies. Finally, we summarize and conclude the main results reported in the chapter (Section 6.5).

6.1 Developers Perception on Explanations

This section describes how we designed the survey with developers to investigate their perceptions of SHAP explanations. First, we present the survey design and the background of the developers who participated. Next, we discuss the results of the study, focusing on the developers' perceptions of SHAP explanations.

Survey Design. We designed a survey with developers from various software development backgrounds to investigate a dataset containing features computed from nine NASA software projects (see Section 3.1). In Figure 6.1, we present an explanation for the decision of a model randomly selected from the model space composed of all features. This model represents one instance of one module in one of the nine NASA projects. The red features in the figure increase the model output, while the blue features decrease the output. We found that features LOC_TOTAL=284 (total number of lines of code) and BRANCH_COUNT=45 (number of branches) had the most significant impact on predicting a defective module, whereas features NUM_UNIQUE_OPERANDS (unique operands) and LOC_EXECUTABLE (number of lines of executable code) had a minor impact on the model prediction in the same direction. There were also blue features that contributed to the module not being defective, but their influence was too low to display in the figure. This explanation can be helpful for developers when evaluating a single defective module. Therefore, we used this figure as a reference for our survey study.

Figure 6.1: Local Explanation Randomly Selected from the Predictor.



Source: Elaborated by the author.

Based on this scenario, we developed an online survey with 40 developers. We invited the developers via mailing lists from two universities in Brazil, and we also publicized the survey on a professional social network. Therefore, we cannot assess exactly how many people accessed the survey link. However, due to the number of participants on these lists, we estimate that a couple of hundred people had access to the survey. The survey is in Portuguese, and we believe that most developers are from Brazil due to the disclosure of the list. The principal example used in the survey is the same as presented in Figure 6.1. The fundamental idea of the survey is to understand whether developers can comprehend the results of our models generated from SHAP. After briefly examining the local explanation (Figure 6.1), we check whether developers are taking the

proper actions on the defective module based on multiple-choice questions. Although the survey was multiple-choice, we provided the participants with a text box to express any opinions about the study design. Additionally, the first part of the survey focuses on the developers' backgrounds, as we discuss next.

Developers' Background. Among the developers that participated in the survey, we found that 18 (45%) held a master's degree in some computer science area, such as information systems, computer science, or computer engineering. Another 11 (27.5%) held an undergraduate degree in a computing-related area, and 7 (17.5%) were still undergraduates in computer-related fields. The remaining 4 (10%) had a Ph.D. in computer science. Twenty-seven developers (67.5%) studied computer science, eleven (27.5%) studied information systems, and two (5%) studied computer engineering in their most recent degree. We conclude that the majority of developers who participated in the survey hold a graduate degree in computer science-related fields (82.5% of developers hold an undergraduate or graduate degree in a computer science-related field). Moreover, most of the participants are masters in computer science and studied computer science in their degree.

The first part of the survey also questioned how long the participants had been developing code in their careers. Table 6.1 shows the results of the years of experience of the developers. The results show that 19 (47.5%) participants had developed software systems for over five years, ten (25%) had developed code between three and five years, and another ten (25%) had worked in the industry for less than three years. Only one participant (representing 2.5%) did not respond to this question, as it was not required to complete the survey. As the survey suggests, most developers who participated had worked in software development for over three years (72.5% of developers). Thus, we conclude that the participants were mostly at a mid-senior level, which is appropriate for analyzing the explanations provided in the study.

Table 6.1: Developers' Years of Experience in Software Development.

Years of Experience	# Developers
More than 5 years	19
Between 3-5 years	10
Less than 3 years	10
Did not respond	1

Source: Elaborated by the author.

Developers Understandability. The second part of the survey evaluates the extent to which the developers understood the importance of the SHAP features. To do so, we showed Figure 6.1 and asked the developers three questions to understand their perceptions of the results. As not all software features are trivial for developers, Table 6.2 provides an explanation of the features contained in Figure 6.1.

Table 6.2: Features Definition for the Survey with Developers.

Feature	**Definition**
Branch Count	Measures the number of branches of the software module.
Essential Complexity	Measures the degree of structure and quality of the code.
Lines of Code Executable	Measures the total lines of code inside the module subtracting the blank lines and comment lines.
Total Lines of Code	Measures all lines of code, regardless of whether they contain code, comments, or whitespace.
Total Number of Unique Operands	Measures the number of unique operands in a module. Operands are objects manipulated by a software system.

Source: Elaborated by the author.

To facilitate the analysis of the results, Table 6.3 displays the three survey questions we ask developers about their perceptions of the SHAP explanations. Since the survey is in Portuguese, we translated the questions from that language to English (Table 6.3).

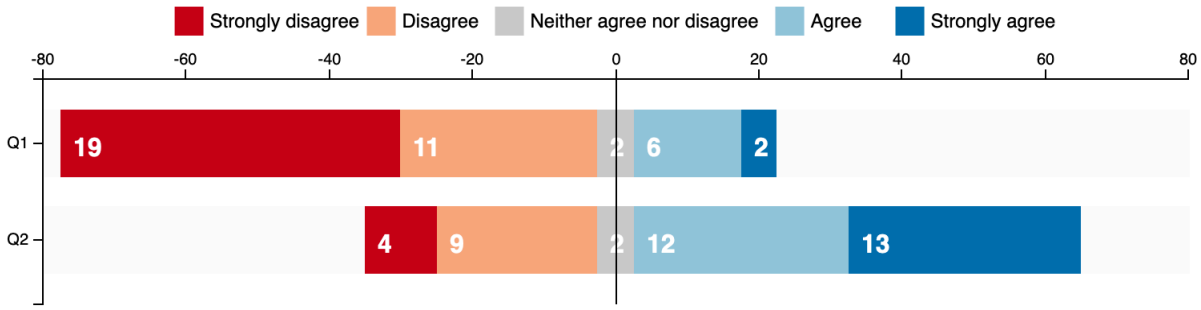
Table 6.3: Set of Questions Developers Answered.

Q1	Acknowledging the local explanation showed in Figure 6.1, do you agree that increasing the total lines of code avoids defects?
Q2	Based on Figure 6.1, do you agree that restricting the total lines of code is more important to avoid defects than restricting the total number of unique operands?
Q3	Pick the set of three features most relevant for causing defects based on the explanation shown in Figure 6.1?

Source: Elaborated by the author.

Q1 - In the first question (Q1), we wanted to understand if the participants would increase the number of lines of code based on the local explanation shown in Figure 6.1. As we can observe in Figure 6.1, the total number of lines of code is associated with defect-prone modules. Therefore, we expected that developers would not increase the size of the module that is already defect-prone (Table 6.3). To analyze the data, we applied a Likert-type scale with five options: (1) strongly disagree, (2) disagree, (3) neither agree nor disagree, (4) agree, and (5) strongly agree, as shown in Figure 6.2. Among the developers, 19 (or 47.5% of participants) strongly disagreed (1) with the increase of lines of code in the module. Eleven developers (27.5%) disagreed with the statement (2), and six developers (15%) agreed (4) to increasing lines of code. Other options had a minor impact on the developers, e.g., only two developers (5%) strongly agreed (5) in adding more lines of code, and two developers (5%) (3) could not give an opinion about the subject. From this question, we may conclude that most developers could understand the local explanation, as 30 developers (75%) disagreed or strongly disagreed that they should increase the lines of code in this module, which is already defect-prone.

Figure 6.2: Questions 1 and 2 about the Defectiveness of the Model.



Source: Elaborated by the author.

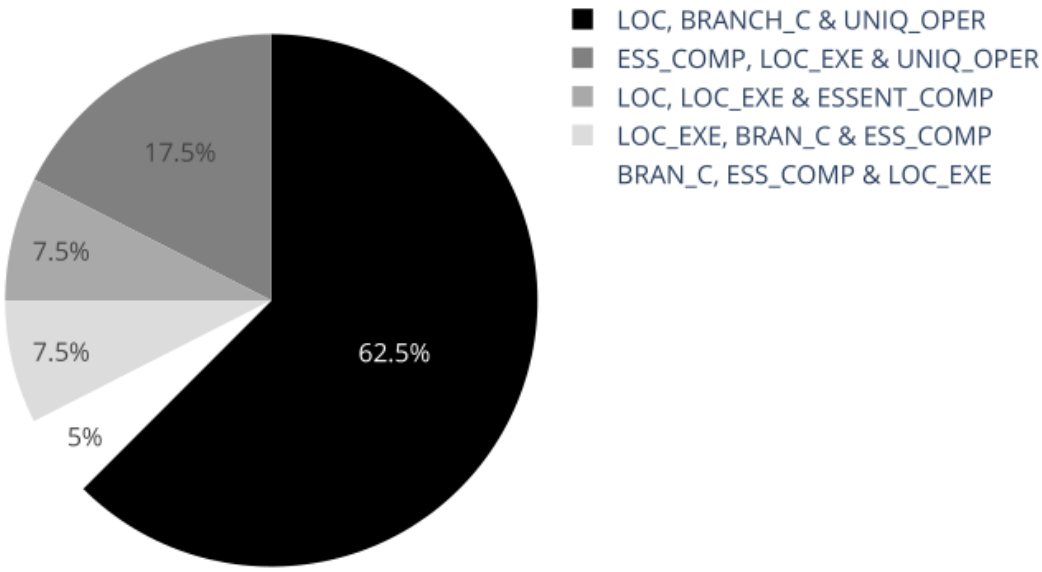
Q2 - In the second question (Q2), we wanted to explore whether the developers understood the order of importance of the features that affect the defectiveness of the module (Table 6.3). Figure 6.1 shows that the number of lines of code is the feature that contributes the most to classifying the module as defective. Thus, we asked if the developers agree that the total number of lines of code was more important than the number of unique operands. We chose this comparison because these two features are similar. This question applied a Likert-type scale [Liu et al., 2017], as in the last question (Q1). We also display the results in Figure 6.2. Thirteen developers (32.5% of participants) (5) strongly agree that the total number of lines of code is more important for the model. Twelve developers (30%) (4) agree that the total number of lines of code is more relevant to the model than the number of unique operands. Nine developers (22.5%) (2) disagree with the importance of the total lines of code. Participants chose other options, as four developers (10%) (1) strongly disagree that the total number of lines of code is more important to the model, while two developers (5%) (3) could not express an opinion about the question. We conclude that most developers (25 or 62.5%) understood the output and considered the total number of lines of code more important than the number of unique operands.

Note that little to no training was provided to the developers to respond to the survey. We adopted a basic textual description of the features and models as the only training for the survey. For this reason, we believe proper training would improve the developers' understanding of the SHAP's importance. However, the results from both questions (Q1 and Q2) show that most developers comprehend the local explanation.

Q3 - Finally, we asked developers to identify the top three features that contribute to a module being defective (Q3), based on Figure 6.1. We presented participants with several combinations of three options (Table 6.3), and the correct answer was the

combination of the total number of lines of code, number of branches, and number of unique operands. As shown in Figure 6.3, 25 developers (62.5%) chose the correct combination of defect-prone features. Overall, we concluded that most developers were able to understand the local explanation generated by SHAP. Considering that 25 developers (62.5%) chose the correct answer for all three questions (where "strongly disagree" or "disagree" and "strongly agree" or "agree" correspond to the correct answer for Q1 and Q2, respectively), we also believe that if developers had access to local explanations during the development of a module, they could anticipate problems that may arise based on the SHAP explanations.

Figure 6.3: Set of Important Features for Developers.



Source: Elaborated by the author.

6.2 Developers Perception of Quality Attributes

For the second study with developers, we created a survey to measure their perceptions of quality attributes. We invited developers to evaluate different scenarios that used a set of quality attributes, namely Documentation, Coupling, Complexity, and Size. These quality attributes were selected based on their importance for defect prediction (Chapter 5.3). Our aim was to explore the following research question: *Do developers agree with the machine learning models in terms of quality attributes and software features that indicate defective code?*

Furthermore, we invited developers from GitHub due to the large pool of developers we could reach with the survey. Additionally, all projects that we built the models on are open-source and mostly available on GitHub or similar tools (such as GitLab).

Developers Background. Before sending the survey to developers, we conducted a pilot study with members of the Software Engineering Lab (LabSoft) at UFMG. In total, ten researchers helped to enhance the survey with valuable feedback. The pilot survey was helpful in improving several aspects of the study, both related to the developers' background and their perspective on the quality attributes. Thus, we divided the study into two major groups: (i) developers' background and (ii) developers' understanding of the quality attributes and their impact on software defects.

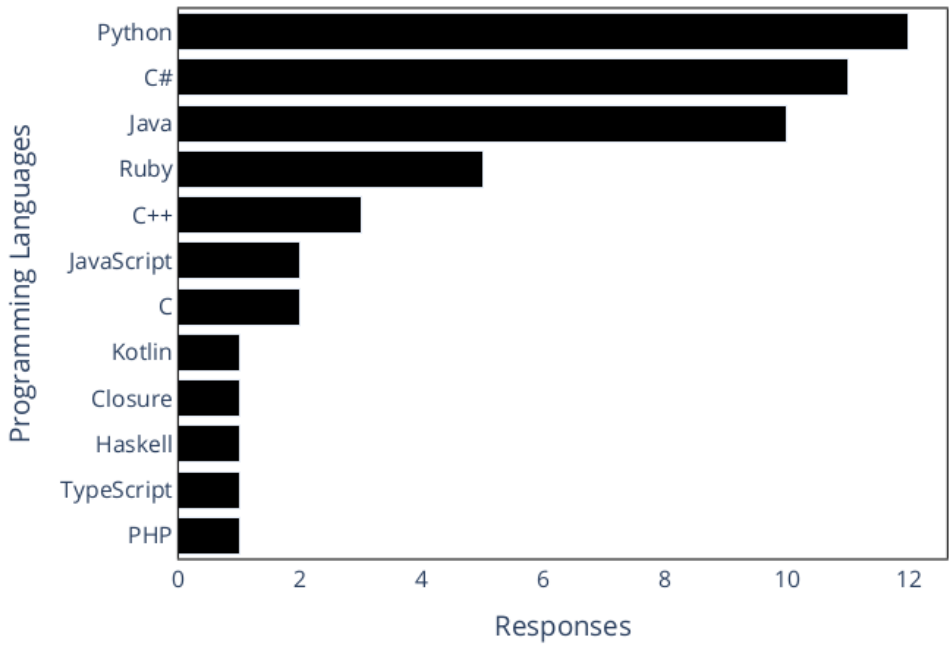
After the pilot survey, we executed the consolidated version of the study with a different set of developers. Furthermore, the selected developers had to meet some requirements to participate in the survey. First, we only sent the invitation email to developers who had contributed to at least ten repositories on GitHub in the past two years. Using this criteria, we focused on active developers on GitHub, which is one of the most popular version control systems widely adopted in the software community [Thung et al., 2013, Gousios et al., 2014, Constantino et al., 2021, Oliveira et al., 2021]. As a result, we were able to select more experienced developers who could provide relevant insights into defects and the relationship with the software features and their quality attributes.

We distributed the survey to the developers for three weeks in April 2021. In total, we sent invitations to 735 developers, and 54 responded to the survey. As a result, our acceptance rate was approximately 7.35%. Upon initial analysis, we found that among the 54 responses, we could not consider four of them because the participant submitted invalid or incomplete responses. In the end, we had 50 valid survey responses to analyze and compare with the machine learning model's results. In the first part of the survey, we aimed to collect information about the background of the developers. Therefore, we asked three questions about their experience with programming languages and software development. First, we asked the developers about their highest level of education at the present moment. In total, 22 developers (44% of the developers) hold a degree in some computer science-related field. Additionally, 13 developers (26%) have a master's degree in a computer science-related field, and 11 developers (22%) are undergraduates in computer science-related courses. Finally, two developers (4%) hold a Ph.D. in a computer science-related field. As we allowed developers to add more responses as they replied to the survey, two developers included unexpected degrees. One participant (2%) answered that they did some college but did not finish it, and one participant (2%) stated that they are a self-taught learner.

In the second background question, we wanted to check which programming lan-

guages the developers are more comfortable coding. In this case, the developers could choose more than one option. Although the models we based our analysis on apply to the Java programming language, we decided to allow developers with any programming language background to respond to the survey since programming languages are not essential to understanding the software quality attributes. In fact, the machine learning models may generalize to other programming languages, and we expect that programming languages share many similarities. As a result, any software development background can help the developers evaluate the software features and their impact on the defectiveness of the code. Figure 6.4 shows the principal programming languages developers use in their projects. We note that the Python programming language is the most common option among developers, with twelve developers (24% of the developers) programming in this language. Following that, eleven developers (22% of developers) develop code in C#. To complete the top 3 languages, ten developers (20% of developers) use Java in their projects.

Figure 6.4: Developers' Primary Programming Languages.



Source: Elaborated by the author.

Finally, we also asked developers about their years of experience in software development to complement their backgrounds. They could choose only one of three options. According to Table 6.4, 22 developers (44% of respondents) have between five and ten years of experience, while 19 developers (38% of respondents) have over ten years of experience. Only nine developers (18% of respondents) have less than five years of experience. From this part of the survey, we conclude that most developers have a significant amount of experience, as 41 developers (82% of respondents) have more than five years of experience.

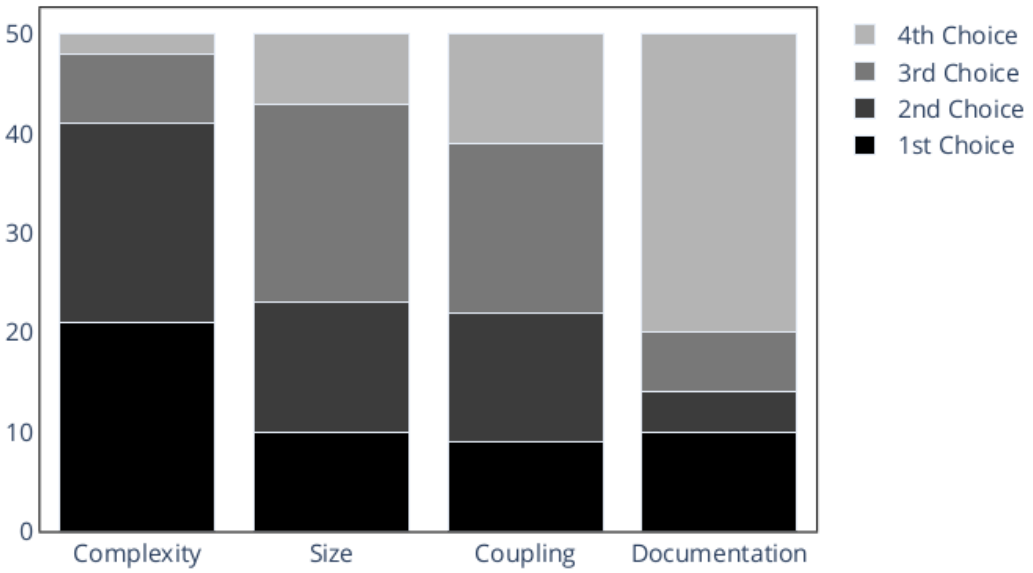
Table 6.4: Developers' Years of Experience with Software Development.

Years of Experience	# Developers
Less than 5 years	9
Between 5-10 years	22
More than 10 years	19

Source: Elaborated by the author.

Developers' Software Features Perception. For the second part of the survey, we questioned developers about how they perceived a list of software features and their relationship to defects in the source code. To do so, we present a scenario of use that focuses on API development not tied to on any programming language. Although we provide insights into the behavior of the API and specific classes that relate to each of the quality attributes explored in our investigation, we supply the developers with little information about the structure of the project. Thus, the primary goal of this part of the survey is to capture developers' perceptions about the potential impact of each quality attribute on software defects. We originally grouped these software features into four quality attributes [Ferenc et al., 2018, 2020a,b]: documentation, size, complexity, and coupling. Figure 6.5 shows how the developers rank each quality attribute assessing their impact to cause defects in the software class. The stacked bar chart represents how many developers ranked each quality attribute as the most important (i.e., 1st choice to the least important choice 4th). In this case, each developer had to pick a first, second, third, and fourth option.

Figure 6.5: Developers' Perception about the Quality Attributes.



Source: Elaborated by the author.

As stated previously, 54 developers took part in the survey, although we received four invalid responses. Among them, twenty-one developers (42% of the developers) consider code complexity the key quality attribute contributing to defects in the source code compared to the other quality attributes (documentation, size, and coupling). Another 10 developers (20% of the developers) classify code documentation and size as the most relevant quality attribute to indicate software defects in the source code. In addition, only nine developers (18% of the developers) think that coupling is the major category to predict software defects. This result differs from the machine learning models we discussed in Chapter 5.3 since the documentation and size quality attributes are the most relevant categories to predict defects in the source code according to our models. As a result, we conclude that the machine learning model contradicts developers' common sense.

To further explore the developers' perception of the quality attributes, we asked the developers four questions concerning each of the quality attributes that group the software features discussed in Section 2.3. Hence, we employed a software feature included in the quality attribute to exemplify the relationship between the group and software development. For instance, for the coupling quality attribute, we picked the Coupling Between Object classes (CBO) to represent the quality attribute. In this case, we opted to introduce the software feature with the most influence on defect prediction. For instance, we relied on WMC as an example of code complexity. We designed these questions to allow developers to analyze each category separately from the remaining ones. Besides, the software feature introduction in each question may help developers examine the effects of the quality attribute in their source code. Table 6.5 presents each of the questions developers had to evaluate.

Table 6.5: Questions for Developers about the Quality Attributes.

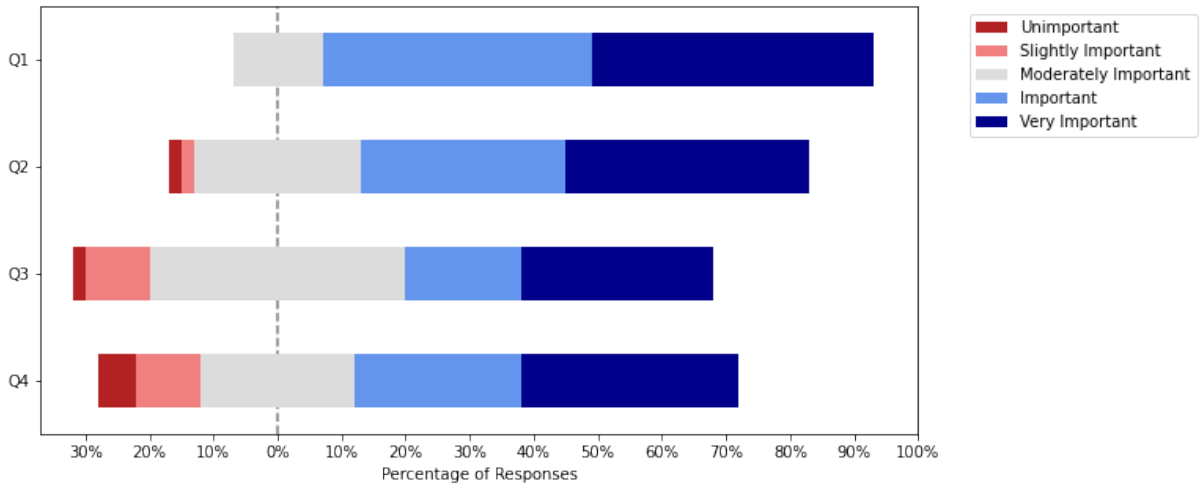
Q1	You notice that major classes have many responsibilities (WMC). Based on your experience, how would you classify the impact of modularization to create precise responsibilities for each class as a contributor to the defectiveness of the API?
Q2	You then notice that the coupling between the objects is also high (CBO). How would you classify the impact of loosely coupled objects to avoid defects in the API?
Q3	In the end, you notice that some classes have too many attributes (NOA). How would you classify the impact of smaller classes to avoid defects in the API?
Q4	You also noticed that the API has little to no documentation. How would you classify the impact of the lack of proper documentation to contribute to the defectiveness of the API?

Source: Elaborated by the author.

After analyzing each question, the developers ranked the quality attributes based on a 5-point scale ranging from "Very Important" to "Unimportant". Figure 6.6 shows the

Likert scale [Liu et al., 2017] used to evaluate the four questions. For the first question (Q1), developers rated the impact of modularization very highly compared to the other three questions. Thus, twenty-two developers (44% of developers) rated the complexity category (represented by the WMC) as "Very Important". Additionally, twenty-one developers (representing 42% of the developers) rated code complexity as "Important". Only seven developers (14% of the developers) believed the WMC was "Moderately Important" for defects. As a result, the complexity category is the most important category for causing defects in the code, based on developers' perceptions.

Figure 6.6: Likert Distribution of Questions.



Source: Elaborated by the author.

The second question (Q2) is similar to the first question (Q1) presented to developers. The only difference is that the developers rate the impact of loosely coupled objects less than the code complexity. In this case, nineteen developers (38% of the developers) rank the coupling quality attribute (represented by the CBO) as very important for the defectiveness of their code. Sixteen developers (32% of the developers) rank the coupling category as important. Another thirteen developers (26% of developers) believe the CBO is moderately important for defects, while one developer (2% of developers) believes the CBO is slightly important for software defects in the code. Additionally, one developer (2% of developers) considers the CBO unimportant for software defects. Hence, we conclude that the coupling category is the second most important category to cause defects in the code based on developers' perception.

The third question (Q3) is also similar to the first two questions (Q1 and Q2). Developers rate the impact of smaller classes (represented by NOA) as less important than code complexity and coupling between objects. The number of attributes is the most controversial quality attribute as many developers ranked the category as moderately important to cause a defect. In this case, twenty developers (40% of the developers) rank

the code size as moderately important for the defectiveness of their code, while fifteen developers (30% of developers) ranked the size as very important and nine developers (18% of developers) ranked the category as important. On the other hand, five developers (10% of developers) ranked the category as slightly important. Finally, only one developer (2% of developers) ranked the category as unimportant. As a result, the code size is not as important as the other two categories (complexity and coupling).

The final question (Q4) is similar to the remaining three questions (Q1, Q2, and Q3). In this case, developers rate the impact of the lack of proper documentation as the least relevant quality attribute to cause defects in the code. Thus, seventeen developers (34% of the developers) rank documentation as very important for the defectiveness of their code, while thirteen developers (26% of the developers) rank this quality attribute as important. Additionally, twelve developers (24% of developers) believe the documentation is moderately important for software defects in the code, while five developers (10% of developers) consider documentation slightly important, and three developers (6% of the developers) rank documentation as unimportant for code defects. Overall, the documentation category is the least relevant quality attribute to cause defects in the code based on developers' perceptions, although 60% of the developers still consider this quality attribute as very important or important. Therefore, we rely on these questions to answer the research question, *Do developers agree with the machine learning models in terms of quality attributes and software features that indicate defective code?*

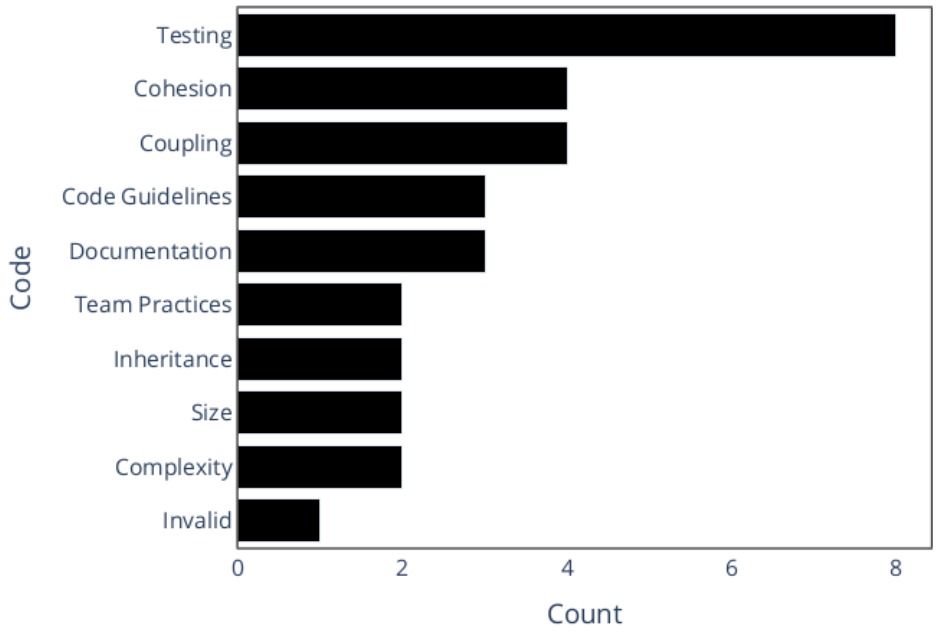
> The developers do not always agree with the models since they believe code complexity is the most relevant quality attribute to indicate software defects, while models indicate documentation as the most important quality attribute.

Thematic Analysis. The survey includes an open question that allowed us to collect qualitative information about quality attributes that are not present in the study. As a result, developers could enter specific software features that they believe are relevant to the defectiveness of software classes. Among the total number of developers (fifty developers), twenty-two (representing 44%) entered the software features that they believe are relevant to the defectiveness of the code and are not available in the static software features. As the field is open, we received all sorts of comments from the developers' perspectives. To analyze the responses, two research members, including the author of the thesis, conducted a thematic analysis of the comments. Thematic analysis is a qualitative method that allows researchers to analyze the data and identify the main themes [Flick, 2014]. Appendix B presents all the comments analyzed in the process. In the first stage of the thematic analysis, the researchers separately classified each of the comments into several categories. These two researchers then discussed the quality attributes with a third researcher to consolidate the comments into reliable categories. Hence, we elaborated on a set of ten quality attributes from the twenty-two developers' comments. The list of

quality attributes includes (i) testing, (ii) cohesion, (iii) coupling, (iv) code guidelines, (v) documentation, (vi) team practices, (vii) inheritance, (viii) size, (ix) complexity, and (x) invalid. We also concluded that the research members responsible for the categorization could use up to three labels (i.e., quality attributes) per comment.

Finally, in the second stage of the thematic analysis, the researchers categorized the comments based on the quality attributes. Figure 6.7 shows the quality attributes created after the thematic analysis. The main quality attribute that developers selected is Testing. In this case, eight developers pointed out that testing is important to avoid defects, and it was not included in the quality attributes. Although we are aware of the impact of testing the source code to guarantee high levels of software quality, this characteristic of the code is not a static software feature. Testing the code is a dynamic aspect of the source code that we have not used to predict defects. Second, four developers think that both cohesion and coupling are important quality attributes, but cohesion is not present in the quality attributes. Cohesion can be statically measured, and in fact, they were included in the unified dataset [Ferenc et al., 2018, 2020a].

Figure 6.7: Thematic Coding of the Open-Field.



Source: Elaborated by the author.

Complementarily, three developers believe that Documentation and Code Guidelines are relevant to avoiding defects in the source code. Code documentation is one of the quality attributes included in the survey. For example, one developer (D15) pointed out that *"Documentation should be intelligible around all development stages (QA, design, product owner). For me, is the most important issue raised on this survey"* (Appendix B, comment 15). On the other hand, we did not include code guidelines in the static feature set because they are not a static feature available in the dataset. We consider

code guidelines to be related to design choices, such as design patterns. For instance, a developer (D20) commented that *"Some well-defined design pattern for the project is also a very important thing"* (Appendix B, comment 22).

Yet, two developers believed that Team Practices, Inheritance, Size, and Complexity are relevant to the defectiveness of their code. Among these options, Inheritance is not included in the study, although we could add features representing it. For instance, DIT (Depth of Inheritance Tree) and NOC (Number of Children) are common features representing this quality attribute. We consider team practices to be code agreements between the team's members (i.e., developers and project managers, among others). For example, a developer (D17) pointed out that *"I would consider testing (at least unit), clean code and language standard conventions"* (Appendix B, comment 17). Note that the comment also mentions testing the code. We consider one invalid response (D1) in the open field as the developer did not answer the question (Appendix B, comment 1).

To sum up, the open-field thematic analysis was relevant to identifying quality attributes not included in the study. Although many quality attributes are related to dynamic aspects of the code (e.g., code testing) or agreements between the team (such as code guidelines and team practices), cohesion is a static quality attribute that we could incorporate in a future investigation of the quality attributes.

6.3 Implications for Developers

The end of these surveys with machine learning models for developers' understandability provided insights into the implications of our work for the community.

(i) Such investigations are relevant as models could benefit developers during software development. We conclude that developers can, for the most part, understand SHAP explanations. Furthermore, it is interesting to note that developers consider complexity as a significant quality attribute. These examinations demonstrate the potential to apply a technique similar to the one used in our experimental phase to build a tool or plugin that can reason about defects in software projects (as discussed in Chapter 4). The tool/plugin could prioritize complexity features, and we could tune the model with the static features that developers cited in the thematic analysis.

(ii) Even though we did not provide detailed training about the first survey (notably the first survey and SHAP explanations), developers were able, in most cases (75% in Q1, 62.5% in Q2, and 62.5% in Q3), to understand the output. We conclude

that we achieved considerable model understandability, as developers could assess the scenario by relying only on their knowledge about software development.

6.4 Threats to Validity

This section reviews the main limitations that could threaten the results discussed in this chapter. We base the threats on well-known categories [Wohlin et al., 2012].

External Validity: The main threat to external validity concerns the classic software features adopted in the first study. These features are based on McCabe or Halstead's work [McCabe, 1976, Halstead, 1977], and they do not capture modern software development. It would be better to employ software features associated with object-oriented design. Regarding the second study, the main threat to external validity concerns the programming language we used to explore the features and models. In this case, we only analyzed the Java programming language. Although the survey with developers had a general-purpose approach, where we did not require developers to have experience with that language, we cannot infer the effect of the programming language on the models and developers' perceptions.

Internal Validity: The main threat to the internal validity of both surveys is related to the features we used to generate the explanations. We cannot guarantee that the authors of these datasets collected the data correctly. Although we applied an extensive data cleaning process to mitigate most of the problems [Petrić et al., 2016], we retained the software feature values and the defect distribution as originally published [Menzies et al., 2007, Ferenc et al., 2020a]. Therefore, if these values are imprecise, the SHAP explanations may not accurately represent the understandability of a defect. Most importantly, we recognize that the number of responses obtained in the surveys may not generalize to the number of software professionals.

Construct Validity: The main threat to construct validity concerns the questions we asked developers about the importance of the software features (Table 6.3). We designed the questions to capture a quantitative overview of developers' perception of a limited local explanation or four quality attributes. Although there is an open field where developers could express their opinions about the study, it would be more appropriate to validate the SHAP explanations/quality attributes with qualitative research. For example, we believe the limitation of the questions may lead developers to the correct answer,

even though we cannot assess how they could apply SHAP explanations in their software development. As a result, we believe the limitation of the questions may result in a superficial comparison between the SHAP explanations and quality attributes.

Conclusion Validity: For the first survey, although developers generally understood the order of the features that influence defects, we noticed that some developers did not clearly understand the output of Figure 6.1 and the importance of the given features (around 37). For the second survey, one prominent comment that emerged in the open field was the inclusion of out-of-scope software features, such as code guidelines, team practices, and testing. We believe the open field was not clear enough in terms of suggestions for static software features (as in the case of coupling and cohesion). As a result, we could not obtain most of the developers' responses due to the out-of-scope software features.

6.5 Final Remarks

In this chapter, we conducted two investigations to examine the importance of understanding the software features that contribute to defects. First, we conducted a survey with 40 developers to evaluate their understanding of SHAP. We conclude that SHAP explanations are useful for developers and that they can understand the most impactful features using only their knowledge of software development. Although it is difficult to evaluate the extent to which it affects them, we conclude that developers could understand the local explanation because 30 developers (75%) disagreed or strongly disagreed that they should increase the lines of code, which are already defect-prone. We also explored whether developers understood the rank of importance of each software feature that affects the defectiveness of the module. Developers performed slightly worse than in the first question because only 62.5% agreed or strongly agreed with the correct order of software features. Finally, we measured developers' understanding of the order of importance of each software feature, and in this case, 62.5% of developers chose the correct combinations of software features contributing to a defective module.

For the second survey, we found that the developers' perception was quite different from that of the machine learning models. We noted that developers classified complexity as the main quality attribute that causes defects in their code, while the models classified documentation as the main quality attribute. This is an interesting result as it contradicts the common sense about the quality attributes and their impact on defects. In the upcoming chapter, we discuss the results of comparing defect models with code smells.

Chapter 7

Comparison with Code Smells

As we discussed in previous chapters, software defects can appear at different stages of the software system life-cycle, degrading software quality and impacting user experience [Haskins et al., 2004]. Sometimes, the damage caused by software defects is irreversible [Menzies and Zimmermann, 2013]. As a result, software costs increase as developers require time to fix defects [Menzies et al., 2010], and it is better to avoid them as much as possible. Several studies have shown that the presence of code smells and anti-patterns are usually related to defective code [Olbrich et al., 2010, Khomh et al., 2012, Hall et al., 2014, Palomba et al., 2018]. Code smells are indications of implementation decisions that may degrade code quality [Fowler, 1999]. Anti-patterns are the misuse of solutions to recurring problems [Brown et al., 1998]. For example, Khomh et al. (2012) found that classes classified as God Classes are more defect-prone than classes that are not smelly. In this chapter, we refer to code smells and anti-patterns simply as code smells.

One technique to mitigate the impact of defects and code smells is the application of strategies to detect problematic code [Nagappan et al., 2006], usually using machine learning models that predict defects or code smells [Menzies et al., 2004, Nagappan et al., 2006, Hassan, 2009, D'Ambros et al., 2010, Khomh et al., 2011, Palomba et al., 2013, Di Nucci et al., 2018, Tantithamthavorn et al., 2019, Cruz et al., 2020]. Training and evaluating machine learning models are difficult tasks, as (i) a large dataset is needed to avoid overfitting, (ii) obtaining labels and features for input is costly and requires different supporting tools, (iii) setting up the environment for training and evaluating models is time-consuming and computationally expensive, even though some tools can help automate the process, and (iv) understanding the importance of features and how they affect the model is complex [Lundberg and Lee, 2017].

With these difficulties in mind, our goal in this chapter is to identify a set of features that developers can use to simplify the process of predicting defects and code smells. We aim to reduce the number of features needed to predict or identify potential candidates with defects and code smells. To the best of our knowledge, no other study has investigated the similarities between defect and code smell models. Instead, most studies focus on proposing and evaluating models that predict defects or detect code smells [Khomh et al., 2011, He et al., 2012, Maiga et al., 2012b, Menzies and Zimmermann, 2013].

In this chapter, we fill this gap by analyzing which features are redundant or different in models built for predicting defects and seven code smells [Riel, 1996, Brown et al., 1998, Fowler, 1999]. Furthermore, we highlight which quality attributes are relevant to their prediction. This analysis is made possible by using the SHAP technique [Lundberg and Lee, 2017, Lundberg et al., 2018b], which determines the contribution of each feature to the prediction. Therefore, using SHAP (similar to the application presented in Chapter 5), we can verify which features contributed the most to the prediction and whether the features had high or low values.

We organize the remainder of this chapter as follows. Section 7.1 describes the study design. Then, Section 7.2 presents the results of our evaluation comparing the defect model with the code smells. Section 7.3 discusses the main threats to the validity of our investigation. Finally, Section 7.4 concludes this chapter.

7.1 Study Design

This section describes the study design employed to compare the software defects and code smell models.

Goals and Research Questions. To investigate the similarities and redundancies between the software features used to predict defects and code smells. We employed data preparation to find the software features for the defect and code smell models. Therefore, our main objective is to compare the software features and quality attributes applied for both predictions. We believe this information may simplify the prediction model and identify possible candidates for introducing defects and code smells. This chapter discusses the following research questions.

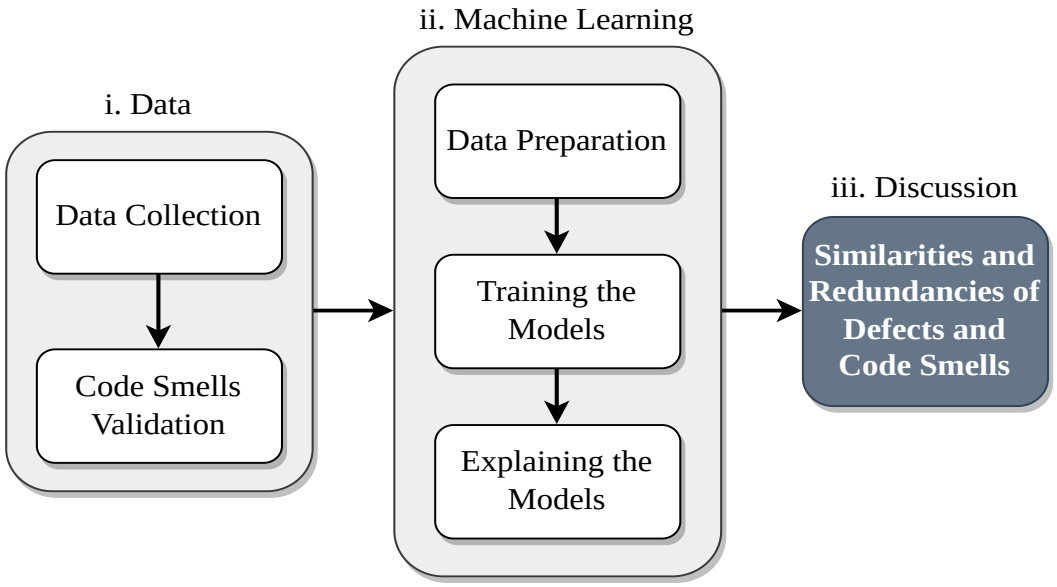
RQ1. Are the defect and class-level code smell models explainable with the data?

RQ2. Which software features are present in both defect and code smell models?

RQ3. Which software quality attributes are more relevant for the prediction of both defects and code smells?

To guide the remainder of the section. Figure 7.1 describes the study design in detail. We start by collecting data about code smells and defects. Then, we validate the code smells with developers. After that, we clean the data within the targets (i.e., code smells and defects). Finally, we train the models and explain the results.

Figure 7.1: Study Design Overview.



Source: Elaborated by the author.

7.1.1 Data

Predicting defects or code smells is a supervised learning problem that requires a dataset with values of independent and dependent variables for each sample [Khomh et al., 2009, Tantithamthavorn et al., 2019]. In this study, we used a dataset that combines several publicly available resources from the literature [Ferenc et al., 2018, 2020a,b, Tóth et al., 2016] (see Section 3.4). The dataset contains data from 34 open-source Java projects [Vale et al., 2021], and the features can be categorized into seven quality attributes: documentation, coupling, cohesion, clone, size, complexity, and inheritance. It is important to note that the dataset is imbalanced, with only around 20% of the classes having a defect and code smells affecting 4-16.2% of classes. Despite this, the dataset provides a rich set of software features for analyzing defects and code smells. Finally, the use of open-source data makes it easier to collect code smells.

Data Collection. We started by collecting data on code smells to join with the defect data [Ferenc et al., 2018]. We used the Organic tool [Oizumi et al., 2018] to detect the code smells. This tool has been proven to effectively detect code smells and does not require compilation. Since all projects are available on GitHub, we manually cloned the source code matching the project version included in the dataset. As most of the systems in the original dataset have fewer than 1,000 classes (20 systems), we collected data from those with more than 1,000 classes (14 projects): *Ant*, *Broadleaf*, *Camel*, *Elasticsearch*, *Hazelcast*, *JDT*, *Jedit*, *Lucene*, *Neo4J*, *OrientDB*, *PDE*, *POI*, *Titan*, and *Xalan*. We chose

to focus on these projects because they represent 75% of the entire data and are readily available on GitHub. Additionally, we matched the name of the detected instances of code smells to the class name present in our defect dataset. Therefore, regardless of whether a class had a smell or not, we only considered it if the match was found in both datasets (i.e., the one with the defects and the one with the code smells). In these cases, we did not consider the class for further investigation. We used this approach to avoid bias, as it would be unfair to determine that a class that Organic could not find in the defect dataset is non-smelly.

Organic collects a wide range of code smells, including method and class smells. However, as the defect dataset is class-level, we only use the code smells found in classes. For this reason, we obtained the ground truth of nine smells, as described in Section 2.2. After collecting the data, we merged three code smells: Brain Class (BC), God Class (GC), and Complex Class (CC) into one code smell. Despite their different definitions, we merged BC and CC into GC due to their low occurrence in the dataset. GC represents a large class with too many responsibilities that centralize an important portion of functionality [Riel, 1996]. Hence, we named the code smell as God Class (GC), since it is more widely used in the literature [Schumacher et al., 2010]. Consequently, we evaluate seven smells in total (God Class, Class Data Should be Private, Data Class, Lazy Class, Refused Bequest, Spaghetti Code, and Speculative Generality).

Table 7.1 shows a summary of the data for each project. The first column presents the project name and version included in the dataset. The second column presents the number of classes for each system. Columns 3 to 9 show the number of smells found for each project, and the last column shows the number of defects in the system. The "Total" row represents the absolute number of classes and the number of smelly/defective classes. The "Percentage" row presents the percentage of classes affected by smell/defect. We observe from Table 7.1 that the projects vary in size, with Lucene having the fewest classes (500) and Elasticsearch having the most (2605). Although we selected projects with more than 1,000 classes, not all classes were matched between the downloaded GitHub version and Organic. As a result, some projects have less than 1,000 classes. We also observe that the number of smells and defects varies greatly for each system. For instance, the Xalan system has 456 instances of God Class and 947 defects, while even though Neo4J is a large system, it has only 18 defects, i.e., 1% of its classes are defective.

Table 7.1: Summary of the Data for each Project.

Project	Class	CP	DC	GC	LC	RB	SC	SG	defect
Ant 1.7	1592	12	161	403	211	57	102	36	330
Broadl. 3.0	1303	3	231	168	97	66	36	36	277
Camel 1.6	2456	7	115	198	519	53	7	87	550
Elastic 0.9	2605	52	42	380	374	187	88	88	362
Hazelc. 3.3	1443	19	71	74	123	115	26	46	232
JDT 3.4	960	308	44	358	1	54	150	31	197
Jedit 4.3	1108	101	56	331	133	9	144	58	264
Lucene 2.4	500	51	13	96	67	66	36	15	208
Neo4J 1.9	1654	64	20	101	187	67	22	92	18
Orient. 1.6	880	54	30	181	141	40	58	53	171
PDE 3.4	1130	5	34	206	0	22	56	84	167
POI 3.0	822	6	103	58	130	219	18	17	434
Titan 0.5	765	28	11	75	96	18	29	54	66
Xalan 2.7	1794	102	113	456	298	211	159	60	947
Total	**19012**	**812**	**1044**	**3085**	**2377**	**1184**	**931**	**757**	**4223**
Percent.	**100**	**4.3**	**5.5**	**16.2**	**12.5**	**6.2**	**4.9**	**4**	**22.2**

Source: Elaborated by the author.

***CP**: Class Data Should be Private; **DC**: Data Class; **GC**: God Class; **LC**: Lazy Class; **RB**: Refused Bequest; **SC**: Spaghetti Code; **SG**: Speculative Generality.*

Code Smells Validation. To validate the code smells collected with Organic, we conducted a manual validation with developers. First, we selected three of the most frequent code smells from the dataset (God Class, Refused Bequest, and Spaghetti Code), since manual validation is costly and developers have to first understand the code. Then, we elaborated questions about each code smell based on the current literature: God Class (GC) [Schumacher et al., 2010], Refused Bequest (RB) [Lanza et al., 2005], and Spaghetti Code (SC) [Brown et al., 1998]. We then produced a pilot study with four developers that did not participate on the final survey to improve the questions using classes that Organic classified as one of the code smells. This allowed us to verify if the questions were suitable for our goals and whether the surveyed developers understood them. For each instance in our sample, we asked nine questions (three for each code smell). The developer was blind to which code smells they were evaluating and had four possible responses: “Yes”, “No”, “Don’t Know”, and “NA” (Not Applicable). Furthermore, to make our validation robust, we calculated the sample size based on the number of instances for each of the three smells in our dataset. We then set a confidence level of 90% and a margin error of 10%. As a result, the sample size should have at least eighteen classes of each target code smell. Finally, to avoid biasing the analysis, we determined that two developers should evaluate each instance in our sample. In this case, developers had to validate 108 software classes (54 unique). To validate the 108 software classes, we invited fifteen developers from distinct backgrounds.

Table 7.2 presents the questions for each code smell that developers had to answer. The first column represents the name of the code smell (three questions for each). The second column presents the question itself. Finally, the last column is the expected answer that developers should choose to agree with the Organic tool. As there were three questions for each smell, developers needed to reach an agreement on two out of three questions to consider the instance as truly containing the smell. In addition, if two developers that evaluated the same instance disagreed on the presence of the smell, a third and more experienced developer checked the instance to make the final decision. This tiebreaker evaluation was done by two software specialists who did not participate in the previous validation. In the end, the developers agreed that all God Class classifications made by the tool were correct (i.e., 18 out of 18 responses). For Refused Bequest, the developers agreed in 14 out of the 18 software classes (meaning that approximately 77% of developers agreed with the tool). Finally, Spaghetti Code was slightly worse, where the developers classified 13 out of the 18 classes as Spaghetti Code. Thus, Spaghetti Code classes achieved an agreement of 72% between the developers and the tool. The results demonstrate that Organic can identify code smells with an appropriate level of accuracy (around 84% of agreement between them). For this reason, we conclude that the Organic data is adequate to represent code smells, and we can proceed with the data for the experimentation.

Table 7.2: Questions to Manually Validate Code Smells with Developers.

Smell	**Question**	**Answer**
GC	Does the class have more than one responsibility?	Yes
	Does the class have functionality that would fit better into other classes?	Yes
	Would splitting up the class improve the overall design?	Yes
RB	Does the class use only a little of parent's behavior?	Yes
	Does the class have too many methods that overrides the parent behavior?	No
	Would refactoring the inheritance improve the overall design of this class?	Yes
SC	Is the class well-structured? For instance, would you be able to clearly state what the class is doing?	No
	Is the class difficult to maintain? For instance, are there many conditional branches in the code?	Yes
	Are most of the methods from this class interacting with other objects?	No

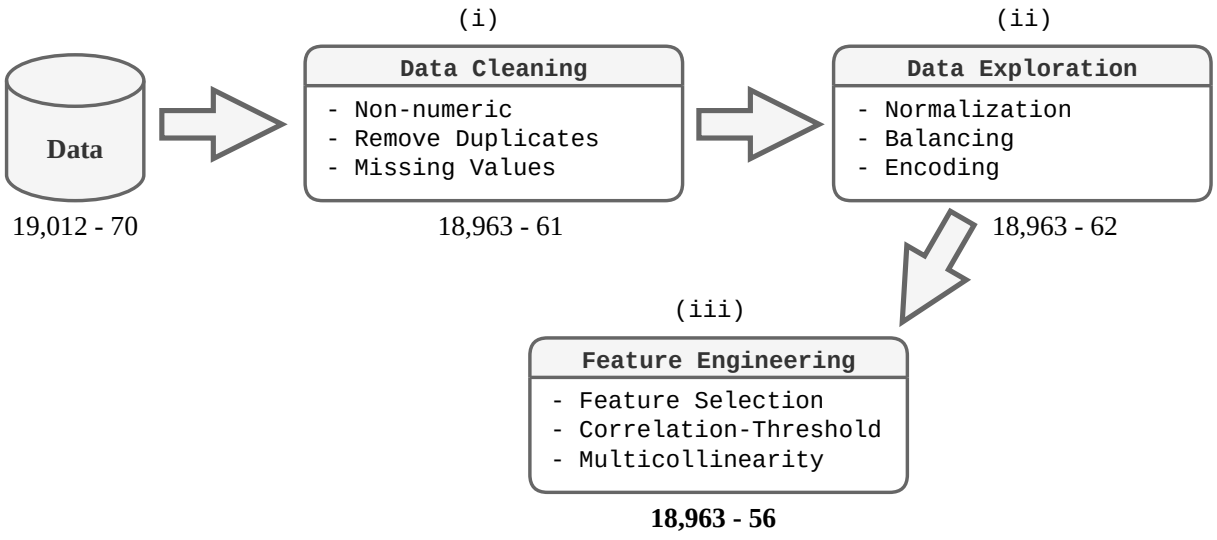
Source: Elaborated by the author.

7.1.2 Machine Learning

The predictive accuracy of machine learning classification models depends on the association between the structural software properties and a binary outcome [Caruana and Niculescu-Mizil, 2006]. In this case, the properties are the software features widely evaluated in the literature [Ferenc et al., 2018, 2020a], and the binary outcome is the prediction of whether the class is defective or non-defective or if the class presents any of the evaluated smells. To compare the defect and code smell prediction models, we rely on the same set of software features, i.e., the models are trained with the same 66 measures, except for the target representing the presence/absence of defect/code smell. We train a separate machine learning model for each target, where each code smell and software defect has its own model. To build these models, we employ a tool known as PyCaret [Ali, 2020] to assist in the different parts of the process, which are described later. Finally, to test the capacity of the models, we apply five evaluation metrics: accuracy, recall, precision, F1, and AUC [Cawley and Talbot, 2010].

Data Preparation. To prepare the data, we follow the fundamental steps described in Figure 7.2. The three rounded rectangles indicate the steps and the actions we performed to prepare the data. Furthermore, we show below each step how many classes and software features we had to process. First, we clean the data (i). Then, we explore the data to identify the best representation for our models (ii). After that, we prepare the features to avoid overfitting (iii).

Figure 7.2: Data Preparation Process Overview.



Source: Elaborated by the author.

Data Cleaning. We first applied data cleaning to eliminate duplicated classes, non-numeric data, and missing values [Petrić et al., 2016]. Hence, we were able to reduce the size of the dataset by removing a small number of repeated entries (61 classes) and over-represented software features, reducing the number of features from 70 to 65. The four removed features gathered information about the exact line and column of the source code where a class started and ended. Finally, we checked for missing values and executed data imputation, but the dataset had none.

Data Exploration. In the second step of the machine learning process, we executed data exploration. We used one-hot encoding [Lin et al., 2014] on the *type* feature, which stores information about the class type, to create two new features for class and interface types. Next, we applied data normalization using Standard Scaler [Raju et al., 2020]. Finally, we employed Synthetic Minority Oversampling Technique (SMOTE) [Tantithamthavorn et al., 2019] to deal with the imbalanced nature of the dataset. Table 7.1 summarizes the imbalanced nature of the targets compared to the data collection. For example, out of 19,000 classes, only 757 exhibit Spaghetti Code (almost 4% of classes). Therefore, oversampling was necessary to generate models that could generalize to unseen data.

Feature Engineering. In the final step, we applied feature engineering to select relevant software features. First, we executed feature selection, correlation analysis, and multicollinearity thresholds. The feature selection technique chose a subset of software features from various permutation importance techniques, including Random Forest, Adaboost, and Linear correlation [Jiarpakdee et al., 2021]. Second, we checked the correlation between the subset of software features (99% of threshold). We removed five software features (LLDC, TNLPA, TNA, TNPA, and TCLOC) because they were highly correlated with other software features (LDC, CLOC, NA, NLPA, and NPA). Additionally, we set the multicollinearity threshold to 85%, meaning that we removed software features with a correlation higher than the threshold. In the end, we were left with 56 software features.

Training the Models. To build our classifier, we employed a technique known as the ensemble model [Ali, 2020]. This technique learns how to best combine the predictions from multiple machine learning models. Thus, we used a stronger machine learning model in terms of prediction since the ensemble combines the prediction power of multiple models. To assess the performance of our models, we employed a method called k-fold cross-validation. This technique splits the data into K partitions. In our work, we used K=10 [Cawley and Talbot, 2010], and at each iteration, we used nine folds for training and the remaining fold for validation. We then permuted these partitions on each iteration. As a result, we used each fold as training and as the validation set at least once [Kohavi, 1995].

This method allows us to compare distinct models, helping us to avoid overfitting as the training set varies on each iteration.

To identify which models are suitable for our goal, we evaluated fifteen machine learning algorithms: CatBoost Classifier [Ali, 2020], Random Forest [Fukushima et al., 2014], Decision Tree [Ferenc et al., 2020a], Extra Trees [Ali, 2020], Logistic Regression [Jiang et al., 2013], K-Neighbors Classifier (KNN) [Xuan et al., 2015], Gradient Boosting Machine [Yatish et al., 2019], Extreme Gradient Boosting [dos Santos et al., 2020b], Linear Discriminant Analysis [Ali, 2020], Ada Boost Classifier [Pedregosa et al., 2011], Light Gradient Boosting Machine (LightGBM) [Ke et al., 2017], Naive Bayes [Turhan et al., 2009], Dummy Classifier [Pedregosa et al., 2011], Quadratic Discriminant Analysis [Ali, 2020], and Support Vector Machines (SVM) [Gray et al., 2009]. This is an extended list of the algorithms presented in Chapter 2. The use of PyCaret allowed us to easily employ these algorithms. Furthermore, to tune the hyperparameters of each model, we applied a technique called Optuna [Akiba et al., 2019]. Optuna uses Bayesian optimization to find the best hyperparameters for each model. After experimenting with all the targets, we observed that five models are able to achieve good performance independently of the target: Random Forest [Fukushima et al., 2014], LightGBM [Ke et al., 2017], Extra Trees [Bui et al., 2022], Gradient Boosting Machine [Tantithamthavorn et al., 2017], and KNN [Xuan et al., 2015]. The data on the performance of the evaluated models can be found in Appendix D. To evaluate our models, we focused on the F1 and AUC metrics. F1 represents the harmonic mean of precision and recall [Davis and Goadrich, 2006]. Additionally, AUC is relevant because we are dealing with binary classification, and this metric shows the performance of a model at all thresholds. For these reasons, both metrics are suitable for the imbalanced nature of data [Cawley and Talbot, 2010].

Explaining the Models. The current literature offers many possibilities to explain machine learning models in various problems. One of the most prominent techniques spread in the literature is the application of SHAP (SHapley Additive exPlanation) values [Lundberg and Lee, 2017], as discussed in Chapter 5. These values compute the importance of each feature in the prediction model. Therefore, we can reason why a machine learning model made specific decisions about the domain. For this reason, SHAP is suitable since machine learning models are challenging to explain [Tantithamthavorn and Hassan, 2018], and features interact in complex patterns to create models that provide more accurate predictions. Consequently, understanding the logic behind a software class is a critical factor that can help address the causes of a defect or code smell in the target class.

7.2 Results

This section presents the results of our study. First, we discuss the performance of the models. Second, we present the results of the comparing the models.

Predictive Capacity. Before explaining the models, we evaluate whether they can effectively predict code smells and defects. Even though we originally built models for the entire set of code smells, we observed that only three code smells (God Class, Refused Bequest, and Spaghetti Code) have comparable models to the defects. For this reason, we focus on the results of these code smells. We believe that some code smells are dissimilar to the defect model because they indicate simple code less likely to have a defect, such as Lazy Class and Data Class. As a result, these code smells do not seem to have similarities with the defects. Results of the remaining code smells are available in Appendix D.

Table 7.3 shows the performance of each ensemble machine learning model with our four targets (i.e., defects and the three code smells). The values in the columns represent the mean of the 10-fold cross-validation. We present the performance for the five evaluation metrics in each column. From Table 7.3, we can observe that the performance of the ensemble model for the four targets is fairly acceptable, with machine learning models presenting an F1 score ranging from approximately 65% (defect model) to 82% (God Class model). These numbers are similar to other studies with similar purposes [Ferenc et al., 2018, 2020a]. We conclude that the models can predict the targets with acceptable accuracy, as shown by the high AUC values in Table 7.3. Therefore, we may exploit these machine learning models to explain their predictions using the SHAP technique. By doing so, we can reason about the similarities of the software features associated with defects and code smells.

Table 7.3: Performance of the Machine Learning Models.

Target	Accuracy	AUC	Recall	Precision	F1
God Class	0.944	0.973	0.801	0.844	0.823
Refused Bequest	0.976	0.951	0.645	0.939	0.765
Spaghetti Code	0.971	0.977	0.715	0.692	0.705
Defect	0.843	0.865	0.701	0.609	0.652

Source: Elaborated by the author.

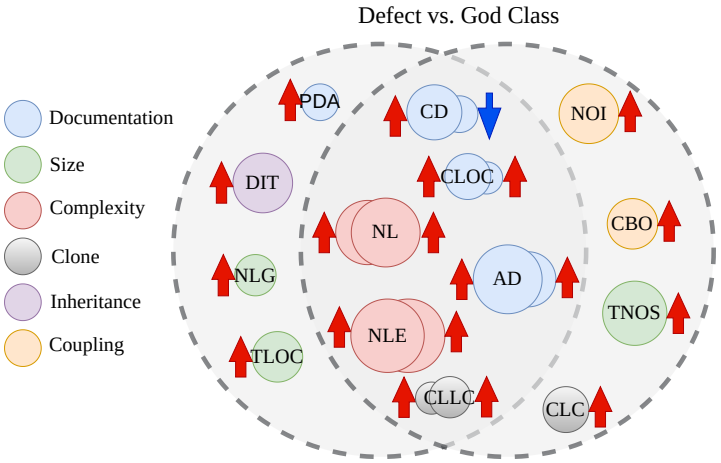
RQ1. The results show that the predictive accuracy of the defect and code smell models can be used to compare the models in terms of their features, with good F1 measures and high AUC. We also found that the class-level code smell models are slightly superior to the defect model in all five evaluation metrics.

Explaining the Models. This section discusses the explanation of each target model, and we rely on SHAP to support the model explanation [Lundberg and Lee, 2017]. To simplify our analysis, we consider the top 10 software features that have the most influence on each prediction model's targets. We then compare each code smell model with the defective one to find similarities and redundancies between the software features that help the machine learning model predict the target code smells and defects. We extract these ten software features from the four target models, which include the defect model and the three code smell models. To illustrate our results, we use a Venn diagram to check the intersection of features between the four models (Figures 7.3, 7.4, and 7.5). The Venn diagram displays two dashed circles: one for the code smell model and another for the defect model. Inside each dashed circle, we present the top-10 software features that contribute the most to the prediction of the target with inner circles. The color of these inner circles represents the feature's quality attribute (Section 2.2). Similarly, the size of the inner circle represents the feature's influence on the model, meaning that the larger the size, the more it contributes to the target prediction. On each side of the inner circles, we have an arrow that indicates the direction of the feature value. For instance, a software feature with an arrow pointing up means that the software feature contributes to the prediction when its value is high. On the other hand, a software feature with an arrow pointing down means that the feature contributes to the prediction when its value is low. The software features in the intersection have two inner circles because they have a different impact on each target, i.e., defect and the three code smells. For a better understanding of the acronyms, we provide a table on the right side of each diagram that lists the acronym and the full feature name of all the features that appear on the diagram.

God Class. Figure 7.3 displays the top-10 features that contribute to the Defect and God Class models, along with their feature intersection. From the figure, we can observe that the Defect model has an intersection with the God Class model of six out of ten features, which means that 60% of the top-10 software features that contribute the most to predictions are the same for both models. These features are CD, CLOC, AD, NL, NLE, and CLLC, with most of them related to documentation (three out of six) and complexity (two out of six). The only difference is for the CD feature, where low values help in predicting a God Class. For all the other features, high values predict a defect or a God Class (as indicated by the arrows pointing up). Additionally, in terms of importance, the largest inner circles for both models are for NLE, NL, and AD. However, the importance of AD is smaller for the God Class model than for the Defect model. Meanwhile, for NLE, the importance of the God Class model is slightly larger than that of the defect model. For the NL feature, its importance is equivalent in both models.

Refused Bequest. Figure 7.4 shows the top 10 features that contribute the most to the

Figure 7.3: Top-10 Software Features for the Defect and God Class Models.



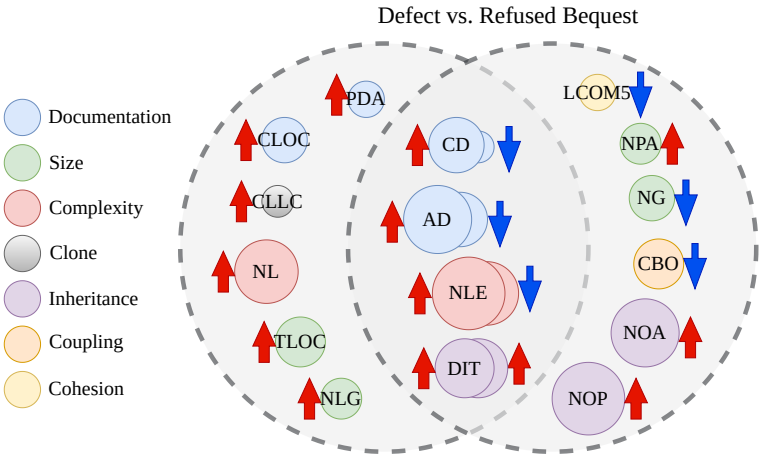
AD	API Documentation
CBO	Coupling Between Object Classes
CD	Comment Density
CLC	Clone Line Coverage
CLLC	Clone Logical Line Coverage
CLOC	Comment Lines of Code
DIT	Depth of Inheritance Tree
NL	Nesting Level
NLE	Nesting Level Else-If
NLG	Number of Local Getters
NOI	Number of Outgoing Invocations
PDA	Public Documented API
TLOC	Total Lines of Code
TNOS	Total Number of Statements

Source: Elaborated by the author.

Defect and Refused Bequest models. We can observe from the Venn diagram in Figure 7.4 that the Defect model has an intersection of 40% (4 out of 10 features) with the Refused Bequest model when considering their top 10 software features. The features that intersect are CD, AD, NLE, and DIT. It is interesting to notice that for three out of the four software features in the intersection, the values that help detect the Refused Bequest have to be low (see arrows pointing down), while for the Defect model, all of them require high values. Furthermore, most of the Refused Bequest features have to be low (6 or 60%). In terms of importance, DIT and NLE features have similar importance for both models. However, CD and AD's contribution to the Refused Bequest model was smaller. Additionally, two features that highly contributed to the Refused Bequest are not in the intersection (NOP and NOA), while one (NL) is outside the intersection for the Defect. We also note that three features are related to the inheritance quality attribute, but only one intersects for both models, the DIT one. We also observe that the size is relevant for both models. However, we do not have any size feature in the intersection of the models. The cohesion aspect was important only for the Refused Bequest model. The documentation attribute, which is relevant for the Defect model (4 out of 10), has two of them with small importance (CLOC and PDA). The complexity attribute, indicated by NLE, is also very relevant for both models. CBO is the only coupling feature in the Refused Bequest model.

Spaghetti Code. Figure 7.5 presents the ten features that are the most important to the Defect and Spaghetti Code models. We observe in Figure 7.5 that the Spaghetti Code model has a 50% intersection with the Defect model. They intersect with the CD, CLOC, CLLC, NL, and NLE software features. For both models, most features need high values, except for one for Spaghetti Code, the CD. The features NL, NLE, and CLOC had similar importance. On the other hand, the CD feature contributes less to the Spaghetti Code,

Figure 7.4: Top-10 Software Features for the Defect and Refused Bequest Models.



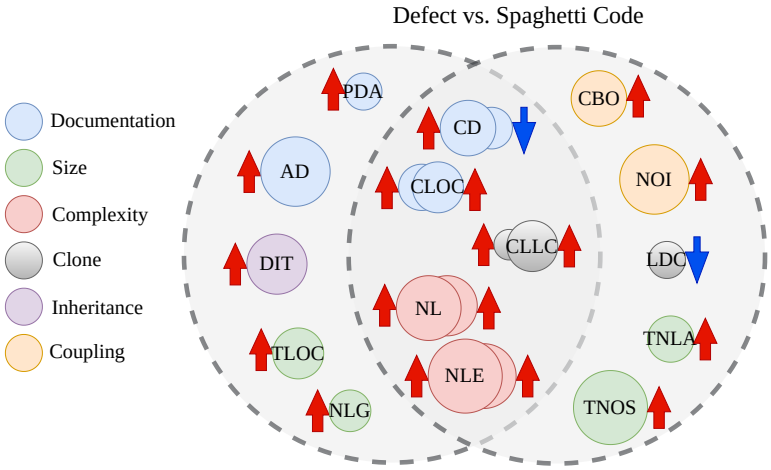
AD	API Documentation
CBO	Coupling Between Object Classes
CD	Comment Density
CLLC	Clone Logical Line Coverage
CLOC	Comment Lines of Code
DIT	Depth of Inheritance Tree
LCOM5	Lack of Cohesion in Methods 5
NG	Number of Getters
NL	Nesting Level
NLE	Nesting Level Else-If
NLG	Number of Local Getters
NOA	Number of Ancestors
NOP	Number of Parents
NPA	Number of Public Attributes
PDA	Public Documented API
TLOC	Total Lines of Code

Source: Elaborated by the author.

while the CLLC feature contributes less to the Defect model. It is interesting to notice that most features that highly contribute to the Spaghetti Code prediction are outside the intersection (NOI, TNOS, and CBO). Furthermore, the complexity quality attribute intersects both models (i.e., 2 out of 5). In addition, two of the documentation features on the Defect model are important for the Spaghetti Code model. In terms of clone duplication, it also intersects half of the features of the Spaghetti Code model (CLLC). The size is relevant for both models, but none of the features intersect (2 out of 10 for both models). The features TLOC and NLG appear on the Defect model, while the TNOS and TNLA appear on the Spaghetti Code model. The coupling is exclusive to the Spaghetti Code model, while the inheritance is exclusive to the Defect model.

Discussion. After observing the three figures (Figures 7.3, 7.4, and 7.5), we notice some intersections between the four models. For instance, CLOC is important for the Defect, God Class, and Spaghetti Code models, even though its importance for God Class was smaller (see inner circle sizes). For this trio, we also note that NL and CLLC were important for all three models, although CLLC made a smaller contribution in comparison to other features. Regarding the Defect, God Class, and Refused Bequest, we highlight that the AD feature was highly important for all three models. In addition, there were some intersections between the smell models. For the God Class and Spaghetti Code pair, we note that both NOI and TNOS were highly relevant to the models. Finally, CBO was moderately important for the God Class, Refused Bequest, and Spaghetti Code models.

Figure 7.5: Top-10 Software Features for the Defect and Spaghetti Code Models.



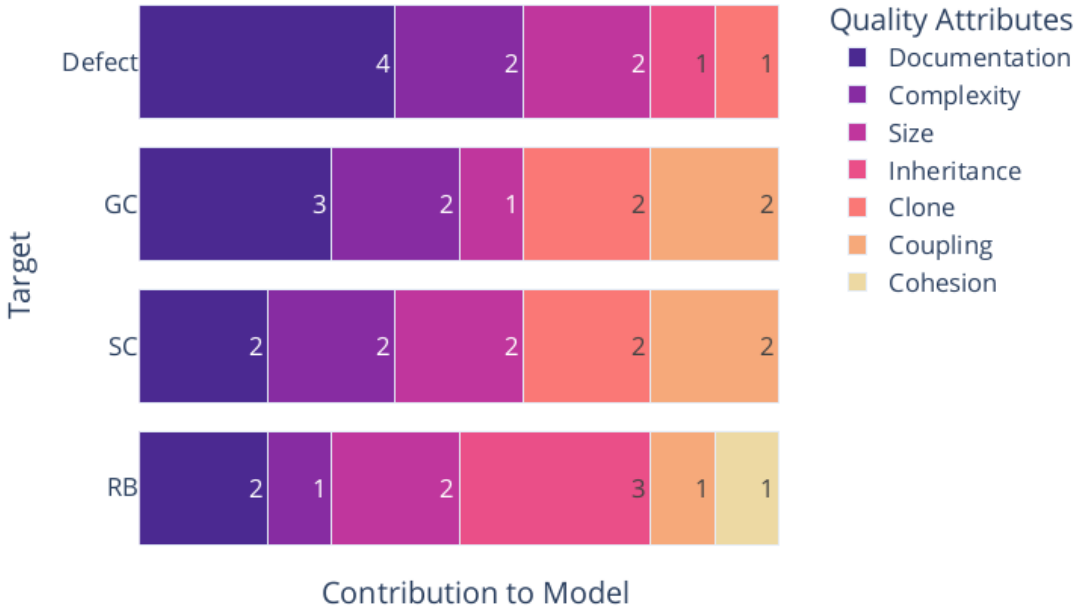
AD	API Documentation
CBO	Coupling Between Object Classes
CD	Comment Density
CLC	Clone Line Coverage
CLLC	Clone Logical Line Coverage
CLOC	Comment Lines of Code
DIT	Depth of Inheritance Tree
NL	Nesting Level
NLE	Nesting Level Else-If
NLG	Number of Local Getters
NOI	Number of Outgoing Invocations
PDA	Public Documented API
TLOC	Total Lines of Code
TLNA	Total Number of Local Attributes
TNOS	Total Number of Statements

Source: Elaborated by the author.

RQ2. There is a group of software features that intersect between the defect models and the three code smells. More importantly, Nesting Level Else-If (NLE) and Comment density (CD) appear in the four models, although the CD influence is considerably low for the evaluated code smells. Furthermore, CBO is important for all the code smells, but not the defect model.

Figure 7.6 presents the number of features corresponding to the evaluated quality attributes according to the top-10 features. We stack each quality attribute horizontally to facilitate comparison. We note that documentation, complexity, and size are the most important quality attributes contributing to the prediction of defects and code smells. Hence, our results suggest that researchers do not need to focus on all features to predict defects and code smells. The redundancies between the models imply wasted research effort, as despite the models dealing with very different symptoms of problematic code (i.e., code smells and defects), they are indeed very similar for at least three code smells. Furthermore, we note that a subset of features is sufficient to predict the targets. For instance, software features related to documentation are the most relevant for the Defect and God Class models, with 4 and 3 features in the top-10, respectively. The Refused Bequest model requires software features related to inheritance (3 features), but size and documentation are also relevant with two features each. Meanwhile, the Spaghetti Code model is the most comprehensive, requiring features linked to documentation, size, complexity, coupling, and clone duplication, all with two features each. Finally, based on the results discussed, we conclude that the four ensemble machine learning models require at least one software feature related to documentation (CD) and complexity (NLE) to predict the target.

Figure 7.6: Comparison between the Top-10 Features of each Target.



Source: Elaborated by the author.

RQ3. The most relevant quality attributes to predict defects and code smells vary greatly between them. For instance, documentation is more important for the Defect and God Class models, while Spaghetti Code has all of its five quality attributes with the same importance, and Refused Bequest prioritizes the inheritance. Documentation, complexity, and size contribute more to the prediction of the targets.

7.3 Threats to Validity

This section reviews the main limitations of this chapter [Wohlin et al., 2012].

Internal Validity: In our investigation, the chosen dataset is a potential threat to internal validity [Wohlin et al., 2012], as we used data documented in the literature [Ferenc et al., 2018, 2020a]. For this reason, we cannot assess data quality, as any storing process could insert erroneous data, which is common in a complex context such as software development. Furthermore, the use of Organic is also a threat; however, we validated the outcome by asking developers for a statistical sample of the results. Finally, the limited number of evaluated projects may hinder the model's generalization to other contexts, although we covered 75% of the defect data with the chosen projects.

External Validity: In this study, the external threat to validity [Wohlin et al., 2012] relates to the limited number of programming languages we examined to compare the targets. In this case, we restricted our analysis to the Java programming language to make it feasible. However, we selected relevant open-source systems that differ in domains, maturity, and development practices. Therefore, we cannot guarantee that our results generalize to other languages and contexts.

Construct Validity: The use of SHAP is a potential threat to construct validity [Wohlin et al., 2012]. Other techniques, such as Lime [Ribeiro et al., 2016], are available to explain a machine learning model.

Conclusion Validity: Our study only matched a portion of the data collected with Organic and the defect data. Even though we retrieved the same version from GitHub, some classes could not be located. One of the main reasons for unmatched software classes is likely due to class name and dependency refactoring. Therefore, we cannot guarantee how different the results would be if more classes could be matched. Additionally, our study focused on a diverse range of domains, which could potentially impact generalization.

7.4 Final Remarks

Building upon the research presented in the previous chapter, in this chapter, we investigated the relationship between defects and code smell models. To do so, we identified and validated the code smells collected with Organic. Then, we applied an extensive data processing step to clean the data and select the relevant features for the prediction. Subsequently, we trained and evaluated the models using an ensemble of models. In the end, as the models performed well, we employed an explainability technique known as SHAP to understand the models' decisions. We concluded that among the seven code smells initially collected, only three of them were similar to the defect model (Refused Bequest, God Class, and Spaghetti Code). In addition, we found that the features Nesting Level Else-If and Comment Density were relevant for the four models. Furthermore, most features require high values to predict defects and code smells, except for the Refused Bequest. Finally, we reported that the documentation, complexity, and size quality attributes are the most relevant for these models. We encourage the community to further investigate and replicate our results. For this reason, we made all data available [dos Santos, 2023b].

Chapter 8

Conclusion

Software defect prediction is a complex task due to the intrinsic characteristics of software that may hinder the complexity of the software system. For this reason, current literature focuses on a diverse range of techniques to predict software defects. Despite the undeniable importance of existing efforts for defect prediction, these studies are concerned with limited aspects of the source code. More importantly, these investigations lack an understanding of why the machine learning model has predicted a target software class as defective-prone. While explaining the decisions made by the models is key to understanding the arrangement of defects, the software engineering literature often leaves aside this topic.

In this thesis, we tackled the understandability of machine learning models for defect prediction. This chapter presents the conclusion of the thesis with insights about future research opportunities on the understandability of defect models. Therefore, we divide the chapter into three main sections. Section 8.1 summarizes the results obtained in the aforementioned chapters. Section 8.2 discusses the main findings of the thesis divided into topics. Finally, Section 8.3 examines the research opportunities left open for future explorations.

8.1 Thesis Summary

In the early stages of the thesis, we conducted an ad-hoc literature review to investigate the state-of-the-art on defect prediction. Our aim was not only to synthesize available evidence in the literature but also to identify gaps and research opportunities. Thus, we identified a research gap in the understandability of machine learning models for defect prediction. For this reason, this thesis aimed to provide a detailed investigation into understanding defects in software instances (i.e., classes or modules). To achieve this goal, we employed a variety of software features to validate our investigations. These software features represented various aspects of the source code, including size, complex-

ity, documentation, coupling, inheritance, and cohesion. We searched the feature space of machine learning models to investigate the influence of software features on the defectiveness of a class or module. This is important because understanding these machine learning models may assist developers in the decision-making process concerning software development and the identification of defective-prone classes or modules. To investigate these overarching goals, we introduced the following specific goals (SG).

- *SG1* Investigate the datasets commonly applied in the current literature to predict software defects.
- *SG2* Find a machine learning model that can search the space of software features comparing the predictive accuracy of these models with baseline machine learning models.
- *SG3* Understand the software features that may generate defects in several software projects for three datasets.
- *SG4* Evaluate developers' perceptions about the software features that contribute to their defective code.
- *SG5* Compare the similarities and redundancies between models for defect prediction and models for code smells detection.

For SG1, we identified three datasets from the ad-hoc literature review conducted at the beginning of this thesis. These datasets vary in software features, size, and defect distribution, among other characteristics. More importantly, these data differ in how they implement the concept of a software defect. We concluded that the datasets comprise four relevant sources for the research community because many studies employed these data to predict defects [Menzies et al., 2004, D'Ambros et al., 2010, Menzies et al., 2010, Jureczko and Spinellis, 2010, Jureczko and Madeyski, 2010, Ferenc et al., 2018, 2020a]. Hence, these data illustrated the importance of data quality and reproducibility for defect prediction investigations. Although most software features are unique to each dataset, the data shared the imbalanced nature frequently found in the current literature. In this case, the datasets usually have more clean instances (i.e., classes or modules) than defective ones (see Chapter 3). As a result, we concluded that the software features relate to many aspects of the source code. For instance, we identified five categories of software features: (i) class-level, (ii) entropy, (iii) change, (iv) McCabe and Halstead, and (v) additional features not correlated with the remaining groups. Additionally, the class-level software features are divided into seven quality attributes: (i) documentation, (ii) coupling, (iii) cohesion, (iv) complexity, (v) clone, (vi) inheritance, and (vii) size. Furthermore, most software features are associated with object-oriented programming.

For SG2, we employed a set of baseline machine learning models to predict software defects and evaluate the effectiveness of these models. As a result, we compared these baseline models with the proposed implementation of gradient boosting (named Unbiased Search XGBoost). The algorithm conveyed an exploratory examination that produced hundreds of thousands of random machine learning models from several software features. These machine learning models were random because they promptly selected the features from the entire pool of software features available for defect prediction. Finally, we investigated the predictive power of the machine learning models using the target datasets. This examination revealed how hard it is to detect defects, as only a minor fraction of the models (only 1.8%) achieved a performance higher than 83% based on the AUC numbers (see Chapter 4). We concluded that a limited set of features produced high accuracy numbers. We hope our effort may convert into a benchmark for other solutions to defect prediction using Java projects.

For SG3, we applied a technique described as Shapley Additive exPlanations (SHAP) to explain the machine learning models [Lundberg et al., 2018a]. Thus, we could reason about the model decision and how the target software features influence them (i.e., predicting whether a software class is defective or not). The results indicated how difficult it is to generate a unique solution to understand defect models (see Chapter 5). For this reason, we concluded that independent projects are subject to distinct software features that may cause software defects in different parts of the code. Moreover, the SHAP explanations suggested that some software features (e.g., LOC, AGE, NOA, AMC, among others) tend to lead to a higher probability of defects if the software feature value is high. We also concluded that the best-performing models are simpler to understand, as they employ fewer features from the pool of software features.

For SG4, we carried out a survey with 40 developers from different software engineering backgrounds. We examined the developers' perception of SHAP values [Lundberg et al., 2020]. We concluded that SHAP explanations are valuable for developers in two ways. First, developers could reasonably understand and reason about the most impactful software features with only their knowledge about software development. Second, the applied technique identified the software features the developers could eventually act upon in the source code (see Chapter 6). We drew these conclusions from the fact that 30 developers (75%) disagreed or strongly disagreed that they should increase the lines of code in the defective-prone module (where lines of code were the main factor for defect-proneness). In addition, developers performed slightly worse for the remaining two research questions as 25 developers (62.5%) agreed or strongly agreed with the correct order of software features that generated defects in the source code. However, we understand that the number of responses obtained in the survey may not generalize to the number of software professionals, characterizing a threat to internal validity.

For SG5, we examined the similarities and redundancies between defect and code

smell models. To do so, we employed an ensemble method that combines the predictions of multiple models. First, we identified code smells using a tool called Organic and then validated them with developers to ensure the precision of the tool. As a result, we concluded that the tool has an agreement of 84% with developers. Second, we cleaned and selected relevant features for our prediction models. Therefore, we trained and evaluated the models using the ensemble technique. As the models had good performance measures, we applied SHAP to understand the decisions. Our results showed that out of the seven code smells we initially identified, only three had similar models to the defect model. These code smells are Spaghetti Code, God Class, and Refused Bequest. Finally, we also concluded that certain software features, such as Nesting Level Else-If (NLE) and Comment Density (CD), were important for all four models. We observed that most features required high values to predict defects and code smells, except for Refused Bequest. Finally, we concluded that documentation, complexity, and size quality attributes are the most important for these models.

8.2 Main Results

This section presents the main results of the thesis. We distribute the thesis project into five fundamental steps.

Step 1 ⇒ Foundations and Ad-Hoc Literature Review

- Theoretical Foundation: This step resulted in an ad-hoc literature review to identify the most relevant studies about defect prediction and understandability of these machine learning models. Thus, we gathered the essential concepts and definitions concerning defect prediction.
- Description of the machine learning models: This step classified the machine learning models that the current literature applies to detect defects in source code.
- Association of the evaluation metrics: This step discovered the evaluation metrics that studies employ to evaluate the performance of the machine learning models for defect prediction.

Step 2 ⇒ Datasets for Defect Prediction Mapping

- Datasets literature mapping: This step involved identifying the three relevant data for defect prediction. We found that two datasets relate to object-oriented design

(measuring code complexity and size using CK features or other features), while one dataset relates to McCabe and Halstead features.

– Datasets exploration: This step involved exploring the three datasets, including the data distribution and the number of defects related to each software class or module. As the datasets were imbalanced (with more clean classes than defective ones), we explored the data to address this issue.

Step 3 ⇒ Empirical Studies over Defect Prediction Understandability

– Effectively predict software defects: This step resulted in an algorithm based on gradient boosting that explores the feature-space. This investigation reveals how difficult it is to detect defects, as only a small fraction of the machine learning models (1.8%) achieved a performance higher than 83% based on the AUC numbers.

– Understandability of defect prediction models: This step focused on the understandability of defect prediction machine learning models. We determined that the best-performing models are simple to understand, as they use only a few features from the power-set. Additionally, the explanations suggest that numerous software features lead to a higher probability of defect prediction if the feature value is high.

– Developers' perception regarding the defect models: This step aimed to evaluate developers' perception of SHAP explanations generated from the machine learning models. We conclude that the explanations are valuable to developers as they could reasonably understand and reason about the most impactful software. In addition, we identified software features that developers could act upon in the source code. However, we recognize that the number of responses obtained in the survey requires further investigation to complement the findings on developers' perception.

Step 4 ⇒ Comparison with Code Smells Models

– Code smells identification: This step resulted in the identification of code smells using the Organic tool. We focused on class-level code smells, such as Class Data Should be Private, Data Class, God Class, Lazy Class, Refused Bequest, Spaghetti Code, and Speculative Generality. We then merged the code smells with the defect dataset.

– Code smells validation with developers: This step focused on the validation of the code smells. To do so, we determined that fifteen developers should evaluate 108 Java software classes. We asked developers to evaluate classes related to three code smells (i.e., God Class, Refused Bequest, and Spaghetti Code). Developers were blind to the code smells related to each class. In the end, we conclude that the tool has an agreement of 84% with developers.

– Comparison between defect and code smell models: This step aimed at the comparison between software defect models and code smells focusing on similarities and redundancies. We concluded that, in fact, models for defect prediction are similar to models of two code smells: God Class (GC) and Spaghetti Code (SC). In addition, the model is also comparable to the Refused Bequest (RB) model, although the software features have opposite impacts on the target. For instance, Nesting Level If-Else (NLE) required a low value to predict RB while it demanded a high value to predict a defect. We also concluded that two software features are determinants to predict defects with the Unified dataset features, NLE and Comment Density (CD). In addition, we note that documentation, complexity, and size are the most important quality attributes to avoid defects.

Step 5 ⇒ Summarization of Understanding Software Defect Models

– Developers perception about quality attributes: This step focused on the evaluation of the developers' perception about quality attributes and their impact on software defects. We concluded that developers perceive that the complexity of the source code is the most important quality attribute to avoid defects. However, the machine learning model detected that the documentation quality attribute is the most important quality attribute to avoid defects. Furthermore, we noted that developers perceived a list of dynamic software features, such as testing and agreements between the software team (e.g., code guidelines and team practices) as important attributes left from the study.

– Threshold of software features to predict defects: This step aimed at the evaluation of the threshold of selected software features to predict defects. We concluded that most software features should have small values to decrease the probability of the model finding a defect in the class. For instance, developers should keep WMC around 35, and CBOI should not exceed 77 to help the class maintaining a non-defective state.

– Release of the replication package: This step focused on the release of the replication package that included the artifacts of the thesis [dos Santos, 2023b]. The replication package is a crucial resource that allows other researchers to replicate our work and build upon it [Yatish et al., 2019]. Replication is an important part of the scientific process, as it helps to verify and validate the results of a study. By making our data and methods available to the community, we are encouraging others to replicate our work and contribute to the advancement of knowledge in understandability of defect models. Furthermore, the replication helps to ensure the reliability and robustness of research findings, and can lead to the development of new insights. We encourage

the use of our replication package, and hope that it inspires further research and collaboration within the defect prediction community.

8.3 Research Opportunities

This section presents the research opportunities that we identified during the thesis project. We distribute the research opportunities into 5 themes.

- **Method-level Defect Prediction and Code Smells Detection:** This entire thesis evaluated defects or code smells in two specific scopes: class-level or modules. We can still explore the space of methods, decreasing the granularity and providing different insights about the software defects in terms of understandability. In fact, methods are the subject of other studies within the defect prediction [Tosun et al., 2010, Pascarella et al., 2020]. However, there is still room to explore the understandability of machine learning models at the method-level. Furthermore, we can explore the space of code smells in methods since there are several method-level code smells, such as Feature Envy, Long Method, Brain Method, Dispersed Coupling, Intensive Coupling, Long Parameter List, Message Chain, and Shotgun Surgery [Fowler, 1999, Oizumi et al., 2018]. Thus, we could compare the models for method-level defect prediction and code smell detection by replicating the experiments of Chapter 7.

- **Translate the Replication Package into a Tool/Plugin:** The replication package contains all the artifacts to replicate the experiments reported in this thesis. Researchers and practitioners could develop a tool/plugin to automate the process of running experiments and generating the results for unseen data. This tool/plugin could also generate the SHAP explanations for the unseen data. The replication package has a notebook designed for that purpose [dos Santos, 2023b]. However, the notebook is not fully automated. Future iterations of the tool could also automatically collect data from public repositories to retrain the models and allow practitioners to use the tool/plugin to predict defects in their code. Furthermore, a tool/plugin would motivate the open-source community to contribute and improve the tool. As a result, the tool/plugin would be more robust and reliable and potentially integrated with open-source Integrated Development Environments (IDEs) such as Visual Studio Code.

– **Integrate the Models into CI/CD Pipelines:** One great opportunity to apply the results discussed in this thesis is the integration into CI/CD pipelines. Other studies have already explored the use of defect models in real-time software projects [Turhan et al., 2009]. However, that study is relatively old for software development standards, meaning that there are new technologies and opportunities to integrate code checks [Zampetti et al., 2021]. In addition, the models could be integrated into the CI/CD pipelines to provide developers with SHAP explanations. We could do the integration in two ways: (i) developers could use the SHAP explanations to improve the code before the CI/CD pipeline runs the tests; (ii) the CI/CD pipeline could use the models to alert developers about the software features that may cause defects in their code.

– **Defect Definition:** One of the main problems with the defect prediction literature is the lack of standardization for the defect concept itself. For instance, the Jureczko datasets employ the "BugInfo" tool to identify defects, which analyzes the logs of each target repository to identify the commits that fix a defect [Jureczko and Spinellis, 2010]. However, the tool does not consider the number of defects fixed by each commit. Thus, the tool may identify a commit that fixes multiple defects as a single defect. One opportunity to explore is the use of data from GitHub to identify the commits that fix a software defect and the number of defects fixed by each commit. The current literature already considers some heuristics to find a corrective commit [Hindle et al., 2009] that could sustain these explorations. This is important because the definition of the defect concept is essential not only for the performance of the models but also for the understandability of the models. In fact, the understandability of the models is directly related to the definition of the defect concept [Lundberg et al., 2020].

– **Unsupervised Learning:** The current literature on defect prediction mostly employs supervised learning techniques to train the machine learning models. However, other techniques could be explored such as unsupervised learning. For instance, Yang et al. [2016] employed unsupervised learning techniques to predict defects and compared the results with widely applied supervised learning techniques. They conclude that for a set of projects the unsupervised learning techniques outperformed the traditional supervised learning techniques. For this reason, more studies should explore the use of unsupervised learning techniques to predict defects, especially because they do not depend upon the software defect definition discussed above.

Bibliography

M. Abbes, F. Khomh, Y. Guéhéneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *European Conference on Software Maintenance and Reengineering (CSMR)*, 2011.

E. Abdullah AlOmar, M. Wiem Mkaouer, A. Ouni, and M. Kessentini. Do Design Metrics Capture Developers Perception of Quality? An Empirical Study on Self-Affirmed Refactoring Activities. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019.

E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd. Software documentation: The practitioners' perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020.

A. Agrawal and T. Menzies. Is better data better than better data miners?: on the benefits of tuning smote for defect prediction. In *International Conference of Software Engineering (ICSE)*, 2018.

T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *International Conference on Knowledge Discovery & Data Mining (SIGKDD)*, 2019.

M. Ali. *PyCaret: An open source, low-code machine learning library in Python*, 2020. URL https://www.pycaret.org.

S. Amasaki. Cross-version defect prediction using cross-project defect prediction approaches: Does it work? In *Proceedings of the 14rd International Conference on Predictor Models in Software Engineering (PROMISE)*, 2018.

S. Amershi, A. Begel, C. Bird, R. DeLine, H. C. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. Software engineering for machine learning: a case study. In Helen Sharp and Mike Whalen, editors, *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, 2019.

L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *International Symposium on Software Reliability Engineering (ISSRE)*, 2015.

V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. In *IEEE Transactions on Software Engineering (TSE)*, 1996.

V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering (TSE)*, 1996.

Y. Bassil. A simulation model for the waterfall software development life cycle. *International Journal of Engineering and Technology (iJET)*, 2012.

I. Binanto, H. L. H. S. Warnars, F. L. Gaol, E. Abdurachman, and B. Soewito. Measuring the quality of various version an object-oriented software utilizing ck metrics. In *International Conference on Information and Communications Technology (ICOIACT)*, 2018.

D. Bowes, H. Tracy, and J. Petrić. Software defect prediction: do different classifiers find the same defects? In *Software Quality Journal (SQJ)*, 2018.

W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc., 1998.

X. Bui, H. Nguyen, and P. Soukhanouvong. Extra trees ensemble: A machine learning model for predicting blast-induced ground vibration based on the bagging and sibling of random forest algorithm. In *Proceedings of Geotechnical Challenges in Mining, Tunneling and Underground Infrastructures (ICGMTU)*, 2022.

A. Burkov. *The Hundred-Page Machine Learning Book.* Andriy Burkov, 2019.

R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, 2006.

G. C. Cawley and N. L. C. Talbot. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research (JMLR)*, 2010.

T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2016.

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. In *IEEE Transactions on Software Engineering (TSE)*, 1994.

J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement (EPM)*, 1960.

K. Constantino, M. Souza, S. Zhou, E. Figueiredo, and C. Kästner. Perceptions of open-source software developers on collaborations: An interview and survey study. *Journal of Software: Evolution and Process (JSEP)*, 2021.

C. Couto, C. Silva, M. T. Valente, R. Bigonha, and N. Anquetil. Uncovering causal relationships between software metrics and bugs. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012.

D. Cruz, A. Santana, and E. Figueiredo. Detecting bad smells with machine learning algorithms: an empirical study. In *International Conference on Technical Debt (TechDebt)*, 2020.

W. S. Cunha, G. A. Armijo, and V. V. de Camargo. Investigating non-usually employed features in the identification of architectural smells: A machine learning-based approach. In *Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*, 2020.

M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *7th IEEE Working Conference on Mining Software Repositories (MSR)*, 2010.

J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, 2006.

University of Szeged Department of Software Engineering. Openstaticanalyzer, 2022. URL https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer.

D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

G. E. dos Santos. Us-xgb models for defect prediction, 2023a. URL https://doi.org/10.5281/zenodo.7513974.

G. E. dos Santos. Replication package for the thesis "understanding software defects with machine learning", 2023b. URL https://github.com/gesteves91/phd-artifacts.

G. E. dos Santos and E. Figueiredo. Failure of one, fall of many: An exploratory study of software features for defect prediction. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2020a.

G. E. dos Santos and E. Figueiredo. Commit classification using natural language processing: Experiments over labeled datasets. In *XXIII Ibero-American Conference on Software Engineering (CIbSE)*, 2020b.

G. E. dos Santos, E. Figueiredo, A. Veloso, M. Viggiato, and N. Ziviani. Predicting software defects with explainable machine learning. In *19th Brazilian Symposium on Software Quality (SBQS)*, 2020a.

G. E. dos Santos, E. Figueiredo, A. Veloso, M. Viggiato, and N. Ziviani. Understanding machine learning software defect predictions. *Automated Software Engineering Journal (ASEJ)*, 2020b.

G. E. dos Santos, A. Veloso, and E. Figueiredo. The subtle art of digging for defects: Analyzing features for defect prediction in java projects. *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2022a.

G. E. dos Santos, A. Veloso, and E. Figueiredo. Understanding thresholds of software features for defect prediction. *36th Brazilian Symposium on Software Engineering (SBES)*, 2022b.

G. E. dos Santos, A. Santana, G. Vale, and E. Figueiredo. Yet another model! a study on model's similarities for defect and code smells. In *26th International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2023.

K. O Elish and M. O Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software (JSS)*, 2008.

U. Fayyad and K. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 1993.

R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy. A public unified bug dataset for java. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, 2018.

R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy. A public unified bug dataset for java and its assessment regarding metrics and bug prediction. In *Software Quality Journal (SQJ)*, 2020a.

R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy. Unified bug dataset. https://doi.org/10.5281/zenodo.3693686, 2020b.

E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2016.

U. Flick. *An Introduction to Qualitative Research*. SAGE Publications Ltd, 2014.

M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering (ICSE)*, 2011.

F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä. Code smell detection: Towards a machine learning-based approach (icsm). In *International Conference on Software Maintenance (ICSM)*, 2013.

F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. In *Empirical Software Engineering (EMSE)*, 2016.

M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Working Conference on Mining Software Repositories (MSR)*, 2014.

B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, 2015.

G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman. Lean ghtorrent: Github data on demand. *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 2014.

C. Goutte and E. Gaussier. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *Proceedings of the 27th European Conference on Advances in Information Retrieval Research (ECIR)*, 2005.

D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. Using the support vector machine as a classification method for software defect prediction with static code metrics. In *International Conference on Engineering Applications of Neural Networks (EANN)*, 2009.

D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. The misuse of the nasa metrics data program data sets for automated software defect prediction. In *15th Annual Conference on Evaluation Assessment in Software Engineering (EASE)*, 2011.

T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. In *IEEE Transactions on Software Engineering (TSE)*, 2005.

T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. In *IEEE Transactions on Software Engineering (TSE)*, 2012.

T. Hall, M. Zhang, D. Bowes, and Y. Sun. Some code smells have a significant but small effect on faults. In *Transactions on Software Engineering and Methodology (TOSEM)*, 2014.

M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Incorporation, 1977.

B. Haskins, J. Stecklein, B. Dick, G. Moroney, R. Lovell, and J. Dabney. Error cost escalation through the project life cycle. In *INCOSE International Symposium*, 2004.

A. E. Hassan. Predicting faults using the complexity of code changes. In *IEEE 31st International Conference on Software Engineering (ICSE)*, 2009.

Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang. An investigation on the feasibility of cross-project defect prediction. In *Automated Software Engineering (ASE)*, 2012.

S. Herbold. Crosspare: A tool for benchmarking cross-project defect predictions. In *30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 2015.

K. Herzig and A. Zeller. The impact of tangled code changes. *10th Working Conference on Mining Software Repositories (MSR)*, 2013.

A. Hindle, D. German, M. Godfrey, and R. Holt. Automatic classification of large changes into maintenance categories. In *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC)*, 2009.

IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990.

H. Jebnoun, M. Rahman, F. Khomh, and B. Muse. Clones in deep learning code: What, where, and why? In *Empirical Software Engineering (EMSE)*, 2022.

S. Jesus, C. Belém, V. Balayan, J. Bento, P. Saleiro, P. Bizarro, and J. Gama. How can i choose an explainer? an application-grounded evaluation of post-hoc explanations. In *Proceedings of the ACM Conference on Fairness, Accountability, and Transparency (FAccT)*, 2021.

T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013.

J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy. An empirical study of model-agnostic techniques for defect prediction models. In *IEEE Transactions on Software Engineering (TSE)*, 2020.

J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan. The impact of correlated metrics on the interpretation of defect models. *IEEE Transactions on Software Engineering (TSE)*, 2021.

X. Jing, S. Ying, Z. Zhang, S. Wu, and J. Liu. Dictionary learning based software defect prediction. In *International Conference of Software Engineering (ICSE)*, 2014.

M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering (PROMISE)*, 2010.

M. Jureczko and D. D. Spinellis. Using object-oriented design metrics to predict software defects. In *In Models and Methods of System Dependability (MMSD)*, 2010.

Y. Kamei and E. Shihab. Defect prediction: Accomplishments and future challenges. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. In *IEEE Transactions on Software Engineering (TSE)*, 2013.

A. Kaur, K. Kaur, and D. Chopra. Entropy based bug prediction using neural network based regression. In *International Conference on Computing, Communication Automation (ICCCA)*, 2015.

R. Kaur and S. Sharma. An ann based approach for software fault prediction using object oriented metrics. In *Advanced Informatics for Computing Research (ICAICR)*, 2019.

G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *31st Conference on Neural Information Processing System (NIPS)*, 2017.

F. Khomh, M. Di Penta, and Y. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*, 2009.

F. Khomh, S. Vaucher, YG. Guéhéneuc, and H. Sahraoui. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. In *Journal of Systems and Software (JSS)*, 2011.

F. Khomh, M. Di Penta, YG. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. In *Empirical Software Engineering (EMSE)*, 2012.

S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 2007.

P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, 2006.

R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

M. Kuhn. Caret: Classification and regression training. http://topepo.github.io/caret/index.html, 2015.

M. Kuhn and K. Johnson. *Applied Predictive Modeling*. Springer, 2013.

L. Kumar and A. Sureka. A comparative study of different source code metrics and machine learning algorithms for predicting change proneness of object oriented systems. In *Intelligent Software Engineering: Synergy between AI and Software Engineering (ISEC)*, 2017.

M. Lanza, R. Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2005.

S. Levin and Amiram Y. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proceedings of the 13rd International Conference on Predictor Models in Software Engineering (PROMISE)*, 2017.

C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E J. Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *International Conference of Software Engineering (ICSE)*, 2013.

Z. Lin, G. Ding, M. Hu, and J. Wang. Multi-label classification via feature-aware implicit label space encoding. In *International Conference on International Conference on Machine Learning (ICML)*, 2014.

C. X. Ling, J. Huang, and H. Zhang. AUC: A statistically consistent and more discriminating measure than accuracy. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

Z. C. Lipton. The mythos of model interpretability. *Workshop on Human Interpretability in Machine Learning (WHI)*, 2016.

Q. Liu, D. Basu, S. Goel, T. Abdessalem, and S. Bressan. How to find the best rated items on a likert scale and how many ratings are enough. *Database and Expert Systems Applications (DEXA)*, 2017.

S. M. Lundberg and S. Lee. A unified approach to interpreting model predictions. In *Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

S. M. Lundberg, G. G. Erion, and S. Lee. Consistent individualized feature attribution for tree ensembles. *Computing Research Repository (CoRR)*, 2018a.

S. M. Lundberg, B. Nair, M. S. Vavilala, M. Horibe, M. J. Eisses, T. Adams, D. E. Liston, D. K. Low, S. Newman, and J. Kim. Explainable machine-learning predictions for the prevention of hypoxaemia during surgery. *Nature Biomedical Engineering*, 2018b.

S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S. Lee. From local explanations to global understanding with explainable ai for trees. *Nature Machine Intelligence*, 2020.

B. Ma, H. Zhang, G. Chen, Y. Zhao, and B. Baesens. Investigating associative classification for software fault prediction: An experimental perspective. *International Journal of Software Engineering and Knowledge Engineering (SEKE)*, 2014.

L. V. D. Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research (JMLR)*, 2008.

L. Madeyski and M. Jureczko. Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal (SQJ)*, 2015.

A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, and E. Aimeur. Smurf: A svm-based incremental anti-pattern detection approach. In *Working Conference on Reverse Engineering (WCRE)*, 2012a.

A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, G. Antoniol, and E. Aïmeur. Support vector machines for anti-pattern detection. In *Proceedings of International Conference on Automated Software Engineering (ASE)*, 2012b.

R. V. R. Mariano, G. E. dos Santos, M. V. de Almeida, and W. C. Brandão. Feature changes in source code for commit classification into maintenance activities. In *Proceedings of the 18th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2019.

R. V. R. Mariano, G. E. dos Santos, and W. C. Brandão. Improve classification of commits maintenance activities with quantitative changes in source code. In *23rd International Conference on Enterprise Information Systems (ICEIS)*, 2020.

T. J. McCabe. A complexity measure. In *IEEE Transactions on Software Engineering (TSE)*, 1976.

T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 1989.

S. McIntosh and Y. Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018.

T. Menzies and J. S. Di Stefano. How good is your blind spot sampling policy. In *IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2004.

T. Menzies and T. Zimmermann. Software analytics: So what? In *IEEE Software*, 2013.

T. Menzies, J. Distefano, A. Orrego, and R. Chapman. Assessing predictors of software defects. In *In Proceedings, Workshop on Predictive Software Models (PROMISE)*, 2004.

T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. In *IEEE Transactions on Software Engineering (TSE)*, 2007.

T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. In *Automated Software Engineering (ASE)*, 2010.

T. Mori and N. Uchihira. Balancing the trade-off between accuracy and interpretability in software defect prediction. *Empirical Software Engineering (EMSE)*, 2018.

R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *2008 ACM/IEEE 30th International Conference on Software Engineering (ICSE)*, 2008.

N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering (ICSE)*, 2005.

N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, 2010.

J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering (TSE)*, 2018.

H. Nori, S. Jenkins, P. Koch, and R. Caruana. Interpretml: A unified framework for machine learning interpretability. *arXiv preprint arXiv:1909.09223*, 2019.

W. Oizumi, L. Sousa, A. Oliveira, A. Garcia, A. B. Agbachi, R. Oliveira, and C. Lucena. On the identification of design problems in stinky code: experiences and tool support. In *Journal of the Brazilian Computer Society (JBCS)*, 2018.

S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *IEEE International Conference on Software Maintenance (ICSM)*, 2010.

J. A. Oliveira, M. Viggiato, D. Pinheiro, and E. Figueiredo. Mining experts from source code analysis: An empirical evaluation. *Journal of Software Engineering Research and Development (JSERD)*, 2021.

M. Openja, M. M. Morovati, L. An, F. Khomh, and M. Abidi. Technical debts and faults in open-source quantum software systems: An empirical study. *Journal of Systems and Software (JSS)*, 2022.

F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013.

F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014.

F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman. A textual-based technique for smell detection. In *IEEE 24th international conference on program comprehension (ICPC)*, 2016.

F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. In *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.

L. Pascarella, F. Palomba, and A. Bacchelli. On the performance of method-level bug prediction: A negative result. *Journal of Systems and Software (JSS)*, 2020.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau,

M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research (JMLR)*, 2011.

F. Peters, T. Menzies, and A. Marcus. Better cross company defect prediction. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, 2013.

J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. Baddoo. The jinx on the nasa software defect data sets. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2016.

PMD. Pmd source code analyser, 2021. URL https://pmd.github.io/.

C. Pornprasit, C. Tantithamthavorn, J. Jiarpakdee, M. Fu, and P. Thongtanunam. Pyexplainer: Explaining the predictions of just-in-time defect models. In *Proceedings of the 36th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2021.

V. N. G. Raju, K. P. Lakshmi, V. M. Jain, A. Kalidindi, and V. Padma. Study the influence of normalization/transformation process on the accuracy of supervised classification. In *2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT)*, 2020.

W. Rhmann, B. Pandey, G. Ansari, and D K Pandey. Software fault prediction based on change metrics using hybrid algorithms: An empirical study. *Journal of King Saud University - Computer and Information Sciences*, 2020.

M. T. Ribeiro, S. Singh, and C. Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2016.

A. Riel. *Object Oriented Design Heuristics*. Addison-Wesley Professional, 1996.

A. Santana, D. Cruz, and E. Figueiredo. An exploratory study on the identification and evaluation of bad smell agglomerations. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC)*, 2021.

J. Sayyad S. and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005. URL http://promise.site.uottawa.ca/SERepository.

J. Schumacher, N. Zazworka, F. Shull, C. Budinger Seaman, and M. A. Shaw. Building empirical support for automated code smell detection. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010.

L. S. Shapley. A value for n-person games. In H. W. Kuhn and A. W. Tucker, editors, *Annals of Mathematical Studies*. Princeton University Press, 1953.

M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering (TSE)*, 2014.

E. Shihab, Z. Ming Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the eclipse project. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010.

B. Shuai, H. Li, M. Li, Q. Zhang, and C. Tang. Software defect prediction using dynamic support vector machine. In *Ninth International Conference on Computational Intelligence and Security (CIS)*, 2013.

A. Shvets. *Dive into Design Patterns*. Refactoring Guru, 2021.

D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå. Quantifying the effect of code smells on maintenance effort. In *IEEE Transactions on Software Engineering (TSE)*, 2013.

M. Sokolova, N. Japkowicz, and S. Szpakowicz. Beyond accuracy, f-score and roc: A family of discriminant measures for performance evaluation. *Lecture Notes in Computer Science (LNCS)*, 2006.

M. Staniak and P. Biecek. Explanations of model predictions with live and breakdown packages. *The R Journal*, 2019.

M. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, 2008.

E. Stroulia and R. Kapoor. Metrics of refactoring-based development: An experience report. *7th International Conference on Object Oriented Information Systems (OOIS)*, 2001.

Z. Sun, Q. Song, and X. Zhu. Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 2012.

M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, 2015.

C. Tantithamthavorn and A. E. Hassan. An experience report on defect modelling in practice: Pitfalls and challenges. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2018.

C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *International Conference on Software Engineering (ICSE)*, 2015.

C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering (TSE)*, 2017.

C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. In *IEEE Transactions on Software Engineering (TSE)*, 2019.

C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering (TSE)*, 2019.

F. Thung, T. F. Bissyandé, D. Lo, and L. Jiang. Network structure of social coding in github. *17th European Conference on Software Maintenance and Reengineering (CSMR)*, 2013.

H. Tong, B. Liu, and S. Wang. Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. *Information and Software Technology*, 2018.

A. Tosun, A. Bener, B. Turhan, and T. Menzies. Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. *Information and Software Technology*, 2010.

Z. Tóth, P. Gyimesi, and R. Ferenc. A public bug database of github projects and its application in bug prediction. In *Computational Science and Its Applications (ICCSA)*, 2016.

M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

B. Turhan and A. Bener. Analysis of naive bayes' assumptions on software fault data: An empirical study. *Data & Knowledge Engineering*, 2009.

B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering (EMSE)*, 2009.

B. Turhan, A. Tosun, and A. Bener. Empirical evaluation of mixed-project defect prediction models. In *Proceedings of the 37th Conference on Software Engineering and Advanced Applications (SEAA)*, 2011.

G. Vale, C. Hunsen, E. Figueiredo, and S. Apel. Challenges of resolving merge conflicts: A mining and survey study. In *Transactions on Software Engineering (TSE)*, 2021.

M. T. Valente. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. Independente, 2020.

B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015.

C. Vassallo, F. Palomba, A. Bacchelli, and H. C. Gall. Continuous code quality: Are we (really) doing that? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *International Conference of Software Engineering (ICSE)*, 2016a.

T. Wang and W. Li. Naive bayes software defect prediction model. In *International Conference on Computational Intelligence and Software Engineering (CiSE)*, 2010.

T. Wang, Zhiwu Z., X. Jing, and L. Zhang. Multiple kernel ensemble learning for software defect prediction. *Automated Software Engineering Journal (ASEJ)*, 2016b.

E. J. Weyuker. Evaluating software complexity measures. In *IEEE Transactions on Software Engineering (TSE)*, 1988.

C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer, 2012.

Z. Xu, J. Liu, X. Luo, and T. Zhang. Cross-version defect prediction via hybrid active learning with kernel principal component analysis. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

X. Xuan, D. Lo, X. Xia, and Y. Tian. Evaluating defect prediction approaches using a massive set of metrics: An empirical study. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*, 2015.

A. Yamashita and S. Counsell. Code smells as system-level indicators of maintainability: An empirical study. In *Journal of Systems and Software (JSS)*, 2013.

A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *20th Working Conference on Reverse Engineering (WCRE)*, 2013.

Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.

J. Yao and M. Shepperd. Assessing software defection prediction performance: Why using the matthews correlation coefficient matters. In *Proceedings of the Evaluation and Assessment in Software Engineering (EASE)*, 2020.

S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn. Mining software defects: Should we consider affected releases? In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta. Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021.

F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou. The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering (TSE)*, 2017.

S. Zhongbin, L. Junqi, and S. Heli. An empirical study of public data quality problems in cross project defect prediction. *Computing Research Repository (CoRR)*, 2018.

T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *International Conference of Software Engineering (ICSE)*, 2008.

T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE)*, 2007.

Appendix A

NASA Features

Table A.1: Description of NASA Features.

Feature	**Description**
Branch Count	Measures the number of branches of the module.
Cyclomatic Complexity	Measures the number of linearly independent paths through a program's source code.
Design Complexity	Measures the cyclomatic complexity of a module's reduced flowgraph.
Essential Complexity	Measures the degree of structure and quality of the code.
Halstead Content	Measures the number of unique operators and operands.
Halstead Difficulty	Measures the difficulty of a module. It is proportional to the number of unique operators.
Halstead Level	Measures the inverse of the defect proneness of the module.
Halstead Effort	Measures the effort to understand a module. It is proportional to the volume and to the difficulty level.
Halstead Error Estimation	Correlates with the overall complexity of the software.
Halstead Length	Measures the size of the module after removing everything except operators and operands.
Halstead Program Time	Measures the time to develop the module. It is proportional to the effort.
Halstead Volume	Measures the size of the implementation of an algorithm.
LOC Blank	Measures all whitespaces.
LOC and Comments	Measures all lines of code and comments.
Lines of Comments	Measures all lines of comments.
LOC Executable	Measures the total lines of code inside the module subtracting the blank lines and comment lines.
LOC Total	Measures all lines of code, regardless of whether they contain code, comments, or whitespace.
Number Operands	Measures the number of objects manipulated by the module.
Number Operators	Measures the actions of operands.
Number Unique Operands	Measures the number of unique objects manipulated by the module.
Number Unique Operators	Measures the unique actions of operands.

Appendix B

Thematic Analysis of Developers' Responses

This Appendix presents the developers' responses about the quality attributes study and the thematic analysis. The first column represents the ID of the developer (e.g., named from D1 to D22). Columns two to four represent the labels that each developer gave on their comment. On these columns, the 'NA' (Not Applicable) label means that the developer did not give any additional label on their comment. The pilot study revealed that we needed only up to three labels to represent the developers comments. Note that only one gave an Invalid label (D1).

Table B.1: Developers' Responses about the Quality Attributes Study.

Id	Label #1	Label #2	Label #3
D1	Invalid	NA	NA
D2	Testing	NA	NA
D3	Coupling	Inheritance	Cohesion
D4	Testing	NA	NA
D5	Documentation	NA	NA
D6	Cohesion	NA	NA
D7	Coupling	Inheritance	Cohesion
D8	Testing	NA	NA
D9	Testing	NA	Testing
D10	Documentation	NA	NA
D11	Team Practices	NA	NA
D12	Testing	Cohesion	NA
D13	Testing	NA	NA
D14	Code Guidelines	NA	NA
D15	Documentation	NA	NA
D16	Inheritance	Cohesion	Coupling
D17	Testing	Team Practices	NA
D18	Code Guidelines	NA	NA
D19	Testing	NA	NA
D20	Code Guidelines	NA	NA
D21	Size	Complexity	NA
D22	Complexity	Size	Coupling

Developers Responses:

D1: No

D2: Code coverage

D3: Big stack traces caused by the excess of module and entity dependencies also creates a difficulty to detect and solve bugs.

D4: I would consider automated tests (unit, integration, etc...) as an important tool for bug detection

D5: Information about the count of access would help to define the architecture to be used according to expected performance API

D6: I believe that it has many functions that could be one, because they interact with the same method. Just what I saw as a little redundant.

D7: The type of relationship among classes.

D8: Good testing coverage.

D9: Automated Tests; Tests Code Coverage;

D10: I work with Ruby on Rails and one of the language paradigms is that if the code needs comments or documentation to be understood then it is not code that has been well written.

D11: Pure functions. Immutable data structures. Automated regression tests. Generative tests. Small teams working on the code. Automated builds. Continuous integration.

D12: Lack of automated tests would be my #1. Also amount of code churn — if a module has to be changed many times, it's likely that there are bugs on the same module.

D13: Verification of test data in a database

D14: Yes - the best way to avoid bugs is NO CLASSES AT ALL. OOP and classes are a bug farm.

D15: Documentation should be intelligible around all development stage (QA, design, product owner), for my is the most important issue raised on this survey.

D16: This is geared around OOP. My defect level dropped considerably in projects where OOP was not used to the point where even looking at it from this perspective seems to be the fundamental issue.

D17: I would consider testing (at least unit), clean code and language standard conventions.

D18: Pretty much the same principles as Solid, exposed right here, with well semantic and self-explanatory method, class and variable names, I don't think any further comments are necessary, since there is a good distribution of responsibility and good modularity. Well-identified methods. Small projects for a small purpose

D19: Tests!!!

D20: Some well-defined design pattern for the project is also a very important thing.

D21: Repeated code/sentences, count line numbers, number of bucles, number of variables. All of this, if there are to much, most of the times is about the person don't understand the problem and start to create things based in trial and error. In that type of code is common to find bugs

D22: Code complexity, number of conditionals / loops Library usage. too many external libraries points at doing too much. No external libraries can also indicate writing everything. Method length, a 100 hundred line method also has a bad smell.

Appendix C

Quality Attributes

This appendix presents the complete list of class-level software features grouped into quality attributes.

C.1 Clone Duplication

Table C.1: Clone Duplication Class-Level Features.

Acronym	Feature	Status
CC	Clone Coverage	Kept
CCL	Clone Classes	Kept
CCO	Clone Complexity	Kept
CI	Clone Instances	Kept
CLC	Clone Line Coverage	Kept
CLLC	Clone Logical Line Coverage	Kept
LDC	Lines of Duplicated Code	Kept
LLDC	Logical Lines of Duplicated Code	Gone

C.2 Cohesion

Table C.2: Cohesion Class-Level Feature.

Acronym	Feature	Status
LCOM5	Lack of Cohesion in Methods 5	Kept

C.3 Complexity

Table C.3: Complexity Class-Level Features.

Acronym	Feature	Status
NL	Nesting Level	Kept
NLE	Nesting Level Else-If	Kept
WMC	Weighted Methods per Class	Kept

C.4 Coupling

Table C.4: Coupling Class-Level Features.

Acronym	Feature	Status
CBO	Coupling Between Object Classes	Kept
CBOI	Coupling Between Object Classes Inverted	Kept
NII	Number of Incoming Invocations	Kept
NOI	Number of Outgoing Invocations	Kept
RFC	Response for a Class	Kept

C.5 Documentation

Table C.5: Documentation Class-Level Features.

Acronym	Feature	Status
AD	API Documentation	Kept
CD	Comment Density	Kept
CLOC	Comment Lines of Code	Kept
DLOC	Documentation Lines of Code	Kept
PDA	Public Documented API	Kept
PUA	Public Undocumented API	Kept
TCD	Total Comment Density	Kept
TCLOC	Total Comment Lines of Code	Gone

C.6 Size

Table C.6: Size Class-Level Features.

Acronym	Feature	Status
LLOC	Logical Lines of Code	Kept
LOC	Lines of Code	Kept
NA	Number of Attributes	Kept
NG	Number of Getters	Kept
NLA	Number of Local Attributes	Kept
NLG	Number of Local Getters	Kept
NLM	Number of Local Methods	Kept
NLPA	Number of Local Public Attributes	Kept
NLPM	Number of Local Public Methods	Kept
NLS	Number of Local Setters	Kept
NM	Number of Methods	Kept
NOS	Number of Statements	Kept
NPA	Number of Public Attributes	Kept
NPM	Number of Public Methods	Kept
NS	Number of Setters	Kept
TLLOC	Total Logical Lines of Code	Kept
TLOC	Total Lines of Code	Kept
TNG	Total Number of Getters	Kept
TNLA	Total Number of Local Attributes	Kept
TNLG	Total Number of Local Getters	Kept
TNLM	Total Number of Local Methods	Kept
TNLPM	Total Number of Local Public Methods	Kept
TNLS	Total Number of Local Setters	Kept
TNM	Total Number of Methods	Kept
TNOS	Total Number of Statements	Kept
TNPM	Total Number of Public Methods	Kept
TNS	Total Number of Setters	Kept
TNA	Total Number of Attributes	Gone
TNLPA	Total Number of Local Public Attributes	Gone
TNPA	Total Number of Public Attributes	Gone
TCLOC	Total Comment Lines of Code	Gone

C.7 Inheritance

Table C.7: Inheritance Class-Level Features.

Acronym	**Feature**	**Status**
DIT	Depth of Inheritance Tree	Kept
NOA	Number of Ancestors	Kept
NOC	Number of Children	Kept
NOD	Number of Descendants	Kept
NOP	Number of Parents	Kept
TCD	Total Comment Density	Kept
TCLOC	Total Comment Lines of Code	Gone

Appendix D

Remaining Code Smells Performance

This Appendix shows the performance of the remaining code smells that do not have much similarity with the defects.

Table D.1: Remaining Class-Level Code Smells Performance.

Target	Accuracy	AUC	Recall	Precision	F1-Score
Class Data Should be Private	0.943	0.971	0.801	0.843	0.821
Data Class	0.979	0.988	0.829	0.811	0.821
Lazy Class	0.943	0.971	0.801	0.843	0.821
Speculative Generality	0.978	0.967	0.705	0.740	0.722