UNIVERSITY OF CALIFORNIA,
IRVINE

Feature Bias in Machine Learning Models: An In-depth Exploration for Software Engineering Tasks

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Jiri Gesi

Dissertation Committee:
Iftekhar Ahmed, Chair
Sam Malek
Ian Harris

2023

DEDICATION

This dissertation is dedicated to my wife, Jiahui Li, who has been a constant source of support and encouragement during the challenges of doctoral study and life. I am truly thankful for having you in my life. This work is also dedicated to my parents, Shuijin Hua and Yilatu Wu, who have always loved me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

Obtaining a Ph.D. degree is a tough journey. As I reflect on the past four years, I realize that I have made numerous mistakes, rejections, and self-doubt regarding my potential to become a competent researcher. However, I owe my perseverance and progress to the unwavering support and guidance of many people who have accompanied me throughout this journey. I am deeply grateful to all those who have played a part in my academic journey and the successful completion of this PhD dissertation.

First and foremost, I would like to express my heartfelt gratitude to my advisor, Iftekhar Ahmed, for his unwavering guidance, patience, and support throughout my doctoral study. His expertise, knowledge, and insightful feedback have been invaluable in shaping this dissertation. I especially admire his endless energy and resilience in pursuing higher standards in research and career, making him a perfect role model for my upcoming research career. I am extremely fortunate to have him being my advisor.

I am also indebted to the members of my dissertation committee, Sam Malek and Ian Harris, for their invaluable feedback, constructive criticism, and thoughtful suggestions, which have helped me to refine and improve this work.

I am especially grateful to my friend Hanzhang Wang, whose constant support and inspiration have played a significant role in my decision to pursue software engineering studies.

I extend my sincere appreciation to my fellow colleges in the ICS department at UC Irvine, Ningfei Wang, Junze Liu, Aodong Li, Zhe Wang, Yunhan Zhao, Ke Jin, Zhaoyuan Su, Xiangyi Yan, and Haoyu Ma, whose intellectual contributions, stimulating discussions, and camaraderie have enriched my academic experience.

I would also like to express my gratitude to the members of my research group and other students I have had the pleasure of working with at STAIRS lab: Jiawei Li, Xinyun Shen, Yunfan Geng, Andrew Truelove, Shiyue Rong, Qihong Chen, Mahan Tafreshipour, and Jina Chun.

I am also grateful to my family and friends for their unending love, encouragement, and understanding throughout my academic journey. Their support has been a constant source of motivation and inspiration.

Finally, I would like to acknowledge the financial support provided by the UC Irvine ICS department and eBay, which have enabled me to pursue my doctoral studies and carry out this research.

To all who have contributed, encouraged, and supported me throughout my journey, I extend my deepest thanks and appreciation.

VITA

Jiri Gesi

EDUCATION

Doctor of Philosophy in Software Engineering — **2019 - 2023**
University of California, Irvine — *California*

Master of Science in Computer Science — **2016 -2017**
University of Michigan — *Michigan*

Bachelor of Science in Mechanical Engineering — **2012 - 2016**
Xi'an Jiao Tong University — *Xi'an, China*

RESEARCH EXPERIENCE

Graduate Student Researcher — **2019 - 2023**
University of California, Irvine — *Irvine, California*

Graduate Student Researcher — **2021 - 2023**
eBay — *San Jose, California*

Applied Scientist Internship — **2022**
Amazon Science — *Palo Alto, California*

TEACHING EXPERIENCE

Teaching Assistant — **2019–2021**
University of California, Irvine — *California*

Teaching Assistant — **2016–2017**
University of Michigan — *Michigan*

REFEREED CONFERENCE PUBLICATIONS

Leveraging Feature Bias for Scalable Misprediction Explanation of Machine Learning Models **May 2023**
the 45th IEEE/ACM International Conference on Software Engineering, is the premier software engineering conference (ICSE)

An empirical examination of the impact of bias on just-in-time defect prediction **November 2021**
Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)

REFEREED Other PUBLICATIONS

Code Smells in Machine Learning Systems **October 2020**
arxiv

ABSTRACT OF THE DISSERTATION

Feature Bias in Machine Learning Models: An In-depth Exploration for Software Engineering Tasks

By

Jiri Gesi

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2023

Iftekhar Ahmed, Chair

The increasing popularity of machine learning techniques in software engineering research promises to improve software development practices by automating various tasks, such as defect prediction, code completion, bug localization and etc. However, the susceptibility of these models to feature bias can significantly affect their performance and reliability. This dissertation delves deep into the realm of machine learning in software engineering, focusing on the profound effects of feature bias on model performance. Feature bias, characterized by the uneven distribution of features in training datasets, can inadvertently skew the results of machine learning models, leading to potential inaccuracies in predictions.

In the first of the three studies, we embark on a journey to uncover the presence and implications of feature bias in software engineering tasks. Our findings are revelatory, indicating that feature bias is not just a theoretical concern but a tangible issue that can significantly hamper the performance of machine learning models. By analyzing both traditional statistical models and advanced deep learning algorithms, we underscore the pervasive nature of feature bias. The implications of these findings are vast, especially when considering the increasing reliance on machine learning models in software engineering. Ensuring the performance and reliability of these models is paramount, and as such, understanding the role

of feature bias becomes crucial.

In our second study, rather than viewing feature bias as a mere impediment, we harness its characteristics as an advantage. We delve into the mispredictions of machine learning models, using feature bias as a lens to interpret these inaccuracies. This unique approach allows us to gain deeper insights into the areas where models are most susceptible to errors. Building on these insights, we introduce a novel technique aimed at bolstering model performance, especially in regions that are traditionally vulnerable to mispredictions. This proactive approach not only mitigates the negative effects of feature bias but also leverages it to refine and enhance model accuracy.

In the third segment of our research, we delved into feature bias within Transformer-based models. These recent advancements in machine learning have set benchmarks in various software engineering tasks, such as code clone detection, code generation, and code translation. Central to their functionality is the attention mechanism, which allows them to focus on relevant input segments during training and prediction. Despite their impressive performance, we sought to determine if an 'attention bias' exists during predictions. Our findings highlighted a notable attention bias towards specific source code tokens, potentially affecting their efficacy in software engineering tasks. In response, we devised a strategy to enhance Transformer-based model performance by directing their attention to crucial source code tokens, aiming to bolster their reliability in real-world applications.

Our study underscores the criticality of recognizing and mitigating both feature bias and attention bias when crafting machine learning models for software engineering endeavors. The methodologies we introduced serve to enhance the efficacy and dependability of these models, making them more apt for deployment in practical software engineering scenarios.

Chapter 1

Introduction

Machine learning (ML) methodologies have increasingly been integrated into software engineering tasks, mirroring their adoption across diverse disciplines. Notable applications of ML within software engineering encompass defect prediction [78, 79, 62], automatic code completion [44, 178], merge conflict prediction [139], and program synthesis and repair [195, 189, 185, 159]. The efficacy of these ML models is intrinsically tied to the caliber of their training data [28, 38]. A recurrent challenge in data is the imbalanced class distribution, where dominant classes overshadow the dataset, leaving the minority class, often termed as few-shot classes, underrepresented [180].

This paucity of representation for few-shot classes during training is a recognized impediment, critically affecting the model's performance on such classes [76]. This bias permeates various domains, underscoring the complications introduced by imbalanced class distributions. For instance, within facial recognition, discerning individuals with a "normal-sized nose" from web images is markedly more straightforward than those with a "large nose". This imbalance is attributed to the challenges in procuring ample images of individuals with "large noses" during data collection [105]. Analogously, in vehicle recognition, pinpointing

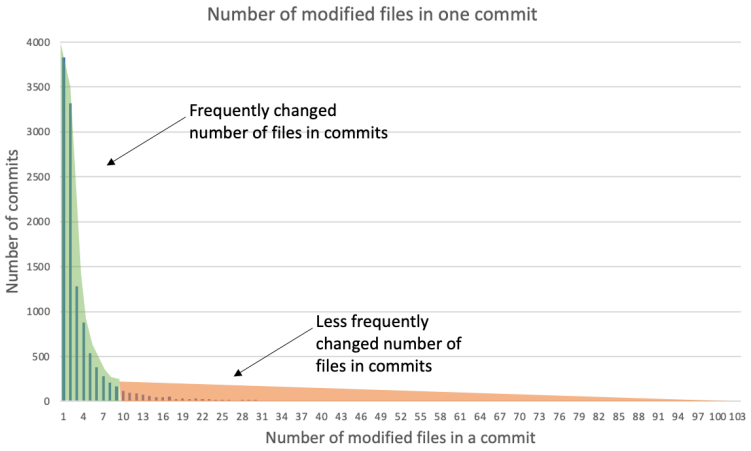


Figure 1.1: Long-tail distribution of number of changed files in a commit.

crashed vehicles is more arduous than identifying standard vehicles due to the dearth of training data encompassing crashed vehicle instances, thereby hindering the model's generalization capabilities [194].

Previous studies in software engineering have explored strategies to counteract data imbalance. For instance, Wang et al.[175] probed the enhancement of defect prediction through imbalance learning methods, encompassing resampling techniques, threshold adjustments, and ensemble algorithms. Their empirical findings suggest that these imbalance learning methodologies can bolster defect prediction performance. Consequently, subsequent defect prediction research began incorporating these imbalance learning techniques[164, 125].

Nevertheless, these studies predominantly concentrated on the class imbalance, such as buggy and non-buggy classes, cloned and not cloned classes . We contend that other dataset attributes might also exhibit imbalance, potentially undermining the prediction model's efficacy. For illustration, Figure.1.1 delineates the frequency in conjunction with the number of modified files in each commit from the OPENSTACK [15] defect prediction dataset. It's evident that a majority of modified files are fewer than 10, with only a handful of commits altering more than ten files. Our objective is to discern whether such attributes influence the performance of defect prediction models and to identify other potential characteristics

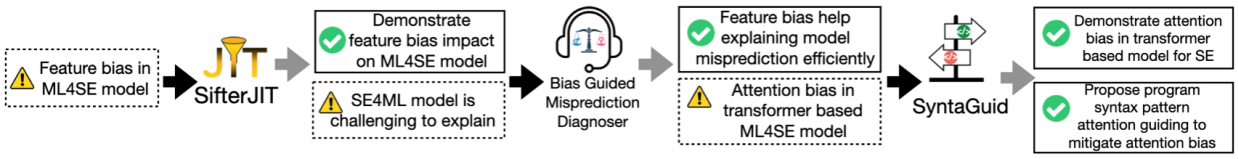


Figure 1.2: Overview of this dissertation

that might similarly impact defect prediction.

Grasping these nuances is paramount, as such attributes could curtail a prediction model's optimal performance by persistently mispredicting commits with specific attributes. Recognizing these limitations can pave the way for devising strategies and tools to address them. Sole reliance on feature importance measurement techniques, such as information gain [150, 119], is inadequate, as these methods are not specifically designed to measure bias. The significance of a metric doesn't necessarily imply its potential for imbalance.

The key insight of our research is the meticulous evaluation of feature bias and its implications on machine learning models tailored for software engineering tasks. Another pivotal insight is derived from the extensive body of research aimed at fortifying the robustness of these models. The technique of explanation generation has emerged as a particularly effective tool, as it elucidates the reasoning behind a prediction [151, 160, 47, 48, 61]. While numerous explanation generation techniques have been explored to illuminate a model's global behavior [151, 160], only a handful, such as [46, 45], have zeroed in on elucidating the mispredictions of an ML model.

Figure 1.2 provides a comprehensive overview of this dissertation's contributions. Drawing from the aforementioned insights, the initial segment of this dissertation empirically assesses source code commit characteristics that could introduce bias in the pivotal software engineering task of defect prediction. Our empirical findings underscore that biases stemming from commit characteristics can profoundly influence the performance of defect prediction models. For instance, in our analyzed dataset, the majority of commits modified fewer than

ten files, with only a scant few modifying over 20. Despite the overarching commendable performance of the defect prediction model, it faltered when assessing commits that modified in excess of 20 files. To counteract this, we introduced a Siamese-based network, SifterJIT. Our empirical results attest to its efficacy in enhancing the model's performance on data with under-represented features, without compromising its performance on other data [62].

From feature bias study, we found that this knowledge can be used to detect the weakness of prediction model. Cito et al. [46] proposed one model weakness explanation technique, i.e., misprediction explanation. However, the limitation of their study is that they blindly evaluating all available features for misprediction explanation. Thus, we utilized our previous perception that model tends to perform poorly on the biased features and proposed Bias Guided Misprediction Diagnoser. Through empirical comparison, the result shows that bias guided misprediction diagnoser not only generate better explanation rules that can cover most of mispredicted instances, but also use less than 90% generation time [63].

Recently, pre-trained Language Models (PLMs) such as BERT [54], GPT [145], and T5 [146] have exhibited notable performance gains in various Natural Language Processing (NLP) tasks [49, 107, 192]. This trend has been further extended to software engineering applications, including but not limited to code summarization [25, 122, 24], code translation [57, 142, 179], and code search [59, 71, 81]. These models are built on the Transformer network architecture [168], featuring a self-attention mechanism that learns the weight and interdependence of attention among tokens within an input sequence. The self-attention mechanism uses attention weight to capture inter-relationships and long-range dependencies among tokens in a sequence. Thus, a natural question raises: do transformer based models have attention bias on inputs source code? Following the similar study methodology with previous two studies, we analyzed the attention weight assigning bias on different part of input source code. And we found that when transformer based model assigns more attention on particular part of the code, the model can perform significantly better.

The main contribution of this dissertation are:

- **Analysis of Feature Bias for ML4SE models**: The paper offers a meticulous evaluation of feature bias, emphasizing its implications on machine learning models specifically designed for software engineering tasks. This foundational insight underscores the importance of understanding and addressing biases in the data to ensure model robustness.

- **Introduction of SifterJIT**: To counteract the observed biases, the paper introduces a novel Siamese-based network, SifterJIT. This architecture is designed to enhance the performance of machine learning models on data with under-represented features, ensuring that the model remains robust across diverse data sets.

- **Bias Guided Misprediction Diagnoser**: Building on the insights from the feature bias study, the paper proposes the Bias Guided Misprediction Diagnoser. This innovative approach is tailored to generate superior explanation rules for model mispredictions, achieving this efficiency in a reduced time frame compared to traditional methods.

- **Mispredicted Area Upweight Sampling**: The paper presents a novel technique, Mispredicted Area Upweight Sampling (MAPS), designed to make the original model allocate more attention to instances pinpointed by the Bias Guided Misprediction Diagnoser. By emphasizing these areas, MAPS ensures that the model is more attuned to potential pitfalls and areas of misprediction, thereby enhancing its overall accuracy and robustness.

- **Analysis of Attention bias in Transformer-based Models**: The research delves into the attention mechanisms of pre-trained Language Models (PLMs) like BERT, GPT, and T5. It scrutinizes the attention weight distribution across different segments

of input source code, revealing that heightened attention to specific code segments leads to superior model performance.

- **Program Syntax-based Attention Guiding Mechanism**: The research introduces a pioneering mechanism based on program syntax. This mechanism is designed to guide the model to allocate more attention to specific syntax tokens. By emphasizing these particular tokens, the mechanism ensures that the model captures essential syntactic nuances, further refining its prediction capabilities.

Chapter 2 provides an exhaustive examination of the Analysis of Feature Bias for ML4SE models and delineates the architecture and implications of SifterJIT. Chapter 3 meticulously presents the nuances of the Bias Guided Misprediction Diagnoser and explicates the mechanisms behind the Mispredicted Area Upweight Sampling technique. Chapter 4 is dedicated to a rigorous assessment of Attention Bias in Transformer-based Models and unveils the pioneering Program Syntax-based Attention Guiding Mechanism. At the last, the summative conclusions and reflections on the research are articulated in Chapter 5.

1.1 Background

We start by introducing the necessary backgrounds for this dissertation.

1.1.1 Data Bias in Software Engineering

In recent years, with the advent of ML, numerous studies have analyzed the biases within ML related software [40, 165, 39]. This type of biases mainly results from biased ML model training/evaluation processes and data distribution imbalance in the training or testing datasets [40, 23, 165, 39, 32, 74]. In addition, the data labeling, model train-

ing, and model evaluation may contribute to building a biased model [166, 28, 129, 171]. Data-driven decisions have the potential to negatively impact already disadvantaged populations [169, 149, 28, 129, 38] as they are relatively less represented in the training data.

1.1.2 Interpret Machine Learning Models

Interpreting *ML* models has been a popular topic for the past couple of years. Local interpretability techniques, such as LIME [151] and Integrated Gradients [160], use several simple, explainable models to simulate complex models. However, the problem with local interpretability techniques is that they can not completely represent the complex models. On the other hand, global interpretability techniques, such as GALE [167], DENAS [43] and BETA [106] help to understand the distribution of the target outcome based on the features. Some techniques try to explain the models by generating counterfactual explanations via modifying the inputs [47, 124, 153]. However, only a few studies investigated explaining model mispredictions. Cito et al. [46] proposed *EXPLAIN* based on rule generation. However, *EXPLAIN*'s rule deducing efficiency is low because it "blindly" analyzes all features. Thus, we proposed an efficient model Misprediction Explanation method that leverages data feature bias in this study.

1.1.3 Self-attention-based Transformer Model

The Transformer [168] architecture, which relies on the self-attention mechanism, has emerged as a popular choice for learning representations of source code. Let $c = \{t_1, t_2, ..., t_n\}$ denote a code snippet consisting of a sequence of n code tokens. A Transformer model comprises L layers of Transformer blocks that transform the code snippet into contextual representations at different layers, denoted by $H^l = [h_1^l, h_1^l, ..., h_n^l]$, where l denotes the l_{th} layer. The layer representation H^l for each layer is computed using the l_{th} Transformer block, i.e.,

$H^l = Transformer(H^{l-1}), l \in \{1, 2, ..., L\}$, where L is the total number of layers.

In each layer of the Transformer model, self-attention heads are utilized to aggregate the output vectors from the previous layer. Given an input sequence of code tokens $c = \{t_1, t_2, ..., t_n\}$, the self-attention mechanism computes a set of attention weights for each token w_i over the tokens in the input, represented as:

$$Atten(w_i) = (\alpha_{i,1}(c), \alpha_{i,2}(c), ..., \alpha_{i,n}(c))$$

Here, $\alpha_{i,j}(c)$ represents the attention that token w_i pays to token w_j, which is computed from the scaled dot-product of the query vector of w_i and the key vector of w_j, followed by a softmax. The general form of the attention mechanism is expressed as the weighted sum of the value vector V, using the query vector Q and the key vector K:

$$Att(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_{model}}}) \cdot V$$

Here, d_{model} denotes the dimensionality of the hidden representation. For self-attention, the query, key, and value vectors are obtained by mapping the previous hidden representation H^{l-1} using different linear functions, i.e., $Q = H^{l-1} \cdot W_Q^l$, $K = H^{l-1} \cdot W_K^l$, and $V = H^{l-1} \cdot W_V^l$, respectively. Finally, the encoder produces the final contextual representation $H^l = [h_1^l, h_2^l, ..., h_n^l]$, which is obtained from the output of the last Transformer block.

To further clarify, the positional encoding of each token is calculated using sine and cosine functions, as shown below:

$$w_i = e(w_i) + pos(w_i)$$

where e denotes the word embedding layer, and pos denotes the positional embedding layer. Typically, the positional encoding implies the position of the code token based on sine and cosine functions.

Overall, the combination of self-attention mechanism, multi-head attention, and positional encoding enables Transformer models to effectively capture both the syntactic and semantic features of source code, making them a popular choice for many software engineering tasks.

1.2 Dissertation Structure

This dissertation is based on the following papers:

Chapter 2 is based on our paper "An Empirical Examination of the Impact of Bias on Just-in-time Defect Prediction", which was published at the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement in 2021. This is the first study to empirically validate a set of commit characteristics that potentially bias the defect prediction performance. In addition, we propose an end-to-end DL framework (SifterJIT) aimed towards improving the prediction performance for few-shot classes.

Chapter 3 is based on our paper "Leveraging Feature Bias for Scalable Misprediction Explanation of Machine Learning Models", which was published at the 45th IEEE/ACM International Conference on Software Engineering in 2023. This study introduces a scalable ML model misprediction explanation rule generation technique named *BGMD*, and introduces a new upweight sampling method that improves model performance on data prone to be mispredicted without requiring extra annotated training data named *MAPS*.

Chapter 4 is based on our paper "Beyond Self-learned Attention: Enhancing Transformer-based Models Using Attention Guidance", which was submitted to the 46th IEEE/ACM

International Conference on Software Engineering in 2024. In this study, we firstly provide the first empirical evidence of attention weight bias towards source code syntax tokens and AST elements in fine-tuned language models. Secondly, we propose a novel attention-guiding technique, SyntaGuid, which enables PLMs to focus attention weight on critical source code syntax tokens and AST elements. Thirdly, we demonstrate the effectiveness of the proposed attention guiding mechanism across multiple software engineering datasets and tasks, establishing its potential as a generalizable solution for improving fine-tuned PLMs performance.

Chapter 2

Feature Bias in Software Engineering Machine Learning Model

2.1 Introduction

Assuring the reliability of software is very important due to its omnipresence. However, it is also inherently a resource-constrained activity. Real-world software systems have more bugs than developers can identify and fix [110]. Moreover, bug fixing is effort-intensive; Kim et al. [98] reported that the time to fix a bug ranges from 100 to 200 days. Therefore, any technique that allows developers to reliably identify buggy parts of the code to guide their bug-fixing efforts is helpful. One such technique is defect prediction. In the last decade, researchers have investigated a wide range of defect prediction models based on different types of metrics, such as metrics about the code [200], historical data [75, 133], and developers' interaction information [147, 112]. These defect prediction techniques aim to isolate the parts of the code that are likely to be buggy so as to facilitate bug-fixing efforts. Just-in-time (JIT) defect prediction [87] is one such technique to predict if a commit

will introduce defects in the future. Such commit level predictions are useful in allocating resources to prioritize fixing the riskiest commits.

Researchers have been investigating ways to improve the JIT defect prediction model's effectiveness. Applying Deep Learning (DL) to automatically extract the semantic and syntactic structure of the actual code changes has been the focus of one such effort [191, 115, 78, 79] and is the state-of-art in terms of performance. For example, Yang et al. [191] utilized Deep Belief Network, Hoang et al. [78] proposed "DeepJIT" which implements Convolutional Neural Network (CNN) to extract features from both commit messages and code changes for defect prediction. Hoang et al. also introduced a hierarchical attention network to construct distributed representations of code changes for JIT defect prediction [79].

All defect prediction techniques, including JIT, relies on the quality of data [28, 38]. However, data often exhibit highly skewed class distribution, i.e., most data belong to majority classes. In contrast, the minority class only contains a small number of instances, also known as few-shot classes [180]. For example, in JIT defect prediction, defect inducing commits would fall in the few-shot class, and non-defect inducing commits would be in the majority class.

Since the few-shot class is under-represented during the training phase [76], trained models perform poorly on the few-shot class. Such bias is well-known in various domains. For example, in face recognition, it is comparatively easier to detect humans with a "normal-sized nose" from web images compared to someone with a "big-nose" since it is easier to obtain face images of "normal-sized nose" than faces with "big-nose" during data collection [105]. For vehicle recognition, it is more difficult to detect crashed vehicles than regular vehicles since training data rarely contain crashed vehicles [194].

Prior studies have investigated ways to deal with data imbalance. Wang et al. investigated how to benefit defect prediction from such imbalance via implementing imbalance learning methods, such as resampling techniques, threshold moving, and ensemble algorithms [175].

Their experimental result shows that these class imbalance learning methods could improve overall defect prediction performance. From there on, more defect prediction studies started using imbalance learning techniques [164, 125].

However, these works only focused on the class imbalance between the buggy and non-buggy classes. We posit that other characteristics of the dataset can also be imbalanced and can have an adverse effect on the prediction model's performance. For example, Figure.1.1 shows the frequency along with the number of modified files in each commit from OPENSTACK [15] defect prediction dataset. We can observe that the major number of modified files is less than 10, and only a few commits modified more than ten files. Our goal is to understand whether characteristics such as these impact defect prediction models' performance and whether other characteristics can affect defect prediction. Understanding this is important because such characteristics can limit the maximum performance of a prediction model by consistently incorrectly predicting commits with certain characteristics. Being aware of them can help alleviate the issue and help devise techniques and tools to deal with them. Simply applying feature importance measuring techniques (such as information gain [150, 119]) will not suffice since these techniques are not tailored for measuring bias (i.e., if a metric is important, it does not mean that the metric must be imbalanced).

In order to identify characteristics that may make the defect prediction dataset biased, we conduct experiments to answer the following research questions:

RQ1: Do commit characteristics have an impact on defect prediction performance?

RQ2: Considering different commit characteristics, which one affects defect prediction performance the most?

RQ3: How well can DL techniques predict defects by explicitly considering few-shot classes?

To improve the prediction performance for the few-shot class, we relied on the Siamese network, which is the most efficient technique for few-shot learning, and we call our new Siamese-based few-shot learning "SifterJIT". We also compared SifterJIT's result with state-of-art JIT techniques. Specifically, this paper makes the following contributions:

- This is the first study to empirically validate a set of commit characteristics that potentially bias the defect prediction performance.
- An end-to-end DL framework (SifterJIT) aimed towards improving the prediction performance for few-shot classes.

The remainder of the paper is structured as follows. Section 2.2 describes the related work. Section 2.3 presents details of our methodology. Section 4.7 reports the findings. Section 4.6 places our results in the broader context of work to date and outlines the implications for practitioners and researchers. Section 4.8 is the threats to validate our results. Section 4.9 concludes with a summary of the key findings and an outlook on our future work.

2.2 Related Work

In this section, we first give an overview of ML-based defect prediction studies. Then, we describe the biases studied in software engineering, ML, and defect prediction. Finally, we provide background about the few-shot learning and Siamese Network.

2.2.1 ML for Defect Prediction

ML based defect prediction techniques have been proposed to predict software defects to reduce the manual effort for identifying defects and reduce software development and main-

tenance cost [75, 85]. A large number of research studies were performed to boost the performance of ML defect prediction models. In these studies, ML models were built from past software data (e.g., software codebase, issue tracking systems, etc.) and then used to predict whether new instances of code regions (e.g., files, changes, and functions) contain or introduce defects [100, 75, 199]. Researchers have investigated on how to manually design new features or combinations of features to represent defects [128]. Prior research also looked into using DL algorithms to learn features or new representations automatically [78, 79]. Besides these approaches, researchers also explored transfer learning [86], Personalized [84] for defect predictions.

To further reduce the costs of software development by identifying defects as soon as they are introduced, research studies [97, 90, 191] in recent years proposed JIT defect prediction techniques. JIT techniques can predict whether a particular code region (e.g., file, code line, and function, etc.) involved in a code change (e.g., commit) will introduce defects in the future. JIT defect prediction allows developers to check and resolve defects as soon as they are introduced. In an ideal scenario, JIT helps to pinpoint the most likely defective commits [89] before those commits are introduced into the codebase. The convenience of providing early feedback to software developers allows them to prioritize and optimize efforts for code review and testing, especially when they are restricted by limited resources [78]. As a result, JIT defect prediction research has gained much attention in recent years [58, 163, 93].

ML techniques such as Support Vector Machine [65], Random Forest [29] and Nearest-Neighbor [154] have been widely used in existing work for building JIT defect prediction models. Similar to regular defect prediction, a common theme of existing JIT defect prediction work is to rely on manually crafted features/metrics to characterize a code change and use them to predict defects [131, 90, 143]. DL techniques have also been adopted in JIT defect prediction [191, 183, 144, 78, 79]. Yang et al. [191] integrated DL in JIT defect prediction by constructing a Deep Belief Network-based approach. Qiao et al. [144] employed a DL

neural network for JIT defect prediction to overcome the difficulty of selecting useful change metrics and mapping between the input (metrics of code changes) and the output (defective or non-defective). Hoang et al. proposed [78, 79] techniques based on deep representation learning to extract semantic feature representations from both commit message and commit code change.

Among the aforementioned DL based JIT defect prediction techniques, we picked the state-of-the-art DeepJIT [78] and CC2Vec [79] with respect to performance. DeepJIT [78] is a DL-based JIT defect prediction technique. It trains on the information of both commit message and code change [78]. DeepJIT uses two separate Convolutional Neural Networks (CNN) for feature extraction and concatenation [111]. Using the resulting vector, the output layer computes the probability of a commit being defective. CC2Vec [79] is an improvement over DeepJIT which uses a Hierarchical Attention Network (HAN) for extracting features. The resulting features are concatenated to form a representation of the code change, which is then concatenated with the commit message vector and the code change vectors generated by DeepJIT. Concatenated vector is then fed into DeepJIT's feature combination layers to predict whether the given commit is defective.

Although the two techniques mentioned above achieved fairly good performance, these frameworks did not improve the prediction performance for classes with a small number of instances (few-shot classes). Since under-represented classes with particular characteristics may negatively impact the JIT defect prediction model's overall performance, improving prediction performance for under-represented classes can boost the overall performance. In our study, we implement the SifterJIT approach to improve the overall performance of DL-based JIT defect prediction.

In the software defect prediction field, there are mainly two popular research branches of analyzing data bias. We provide detailed explanations of these two branches below.

One type of bias in defect prediction datasets stems from the construction of the datasets [31]. Identifying defect-fixing changes is a key to the identification of defect code regions in the codebase to construct a historical dataset for defect prediction models [128, 148]. However, multiple factors (e.g. severity of the defect or the experience of the fixer) can impair the automated identification of defect-fixing changes and further impact the performances of defect prediction models [187]. For instance, suppose only experienced developers annotate their changes as defect-fixing or not. Automated tools only identify defect-fixing changes made by experienced developers. Therefore, there will be an under-representation of the code regions fixed by inexperienced developers in defect prediction datasets; and the resulting bias may negatively affect the performances of the models. Rahman et al. [148] proposed a set of bias-influence metrics to measure the aforementioned bias in file-level defect prediction techniques. Inspired by their work, we identify several commit characteristics to see how does bias among these commit characteristics affects the performances of JIT defect prediction techniques.

Another category of bias studied in defect prediction is mostly the class imbalance problem. Previous studies on defect prediction demonstrate that most of the defects occur in very few modules [200], which indicates that the number of defective instances is much less in number compared to non-defective instances, which results in imbalanced datasets. In such cases, the imbalanced distribution of classes may result in incorrect predictions of the minority class instances. Therefore, handling imbalanced datasets to obtain improved results has received much attention among Software Engineering researchers. Various methods have been developed to deal with imbalanced data like data sampling, cost-sensitive learning, and ensemble methods [88, 157, 60, 121, 175, 103]. To the best of our knowledge, researchers have used the methods mentioned above to mitigate the class imbalance issue [191, 143]. However, in this study, our goal is to investigate the effect of biased commit characteristics on JIT defect prediction and, in addition, we aim to propose an approach to overcome such effect on the performance.

2.2.2 Few-shot learning and Siamese Network

The performances of ML models may be hampered when there are few training instances. However, in some certain areas (e.g. drug discovery [27], image classification [101]), labelled data instances may be difficult or impossible to acquire. Few-shot Learning is proposed to tackle this issue with the help of prior knowledge [180].

As a representative method of few-shot learning, the concept of Siamese networks was proposed by Bromley et al. [34]. Koch [101] and Neculoiu et al. [135] pointed out that the Siamese network is a type of twin framework with two or more identical sub-networks and every sub-network has the same parameters and weights. The parameters of Siamese networks are updated based on the joint performance of all sub-networks. Its classification powers are learned through similar and dissimilar information between data pairs [130]. Moreover, they proved that Siamese networks are good at learning on a dataset where a small amount of data is available [101, 135, 173].

Siamese network has also been used in defect prediction field, Zhao et al. [198] proposed Siamese Dense neural networks (SDNN) based defect prediction model, which integrates similarity feature learning and distance metric learning. After comparison experiments, SDNN outperformed other state-of-the-art defect prediction models on NASA datasets [70]. However, their approach does not leverage the true notions of DL as they still employ the numeric features/metrics that are manually engineered. Our goal is to improve the learning efficiency of the few-shot class with respect to commit characteristics. While SDNN is a few-shot learning model dealing with a lack of sufficient training data, they did not consider the bias existing in the dataset and its impact on the model's performance. In this study, we investigate such bias to fill this gap in existing research.

2.3 Methodology

We use the following process during our study: (A) First, we select state-of-art JIT defect prediction techniques, (B) we select characteristics to investigate; (C) we explore if commit characteristics have an impact on the state-of-the-art DL defect prediction techniques; (D) we measure how much these characteristics can impact the prediction technique by splitting the data based on characteristics; (E) we propose a new DL framework called SifterJIT and compare our framework's performance with existing techniques.

2.3.1 Prediction Technique Selection

Among many available DL-based JIT defect prediction techniques, we picked DeepJIT [78] and CC2Vec [79] since they are state-of-the-art. These two studies use the same training/testing datasets originally curated by McIntosh et al. [128] and have been widely used in JIT defect prediction [78, 79, 143] literature. In the dataset, McIntosh et al. manually filtered and analyzed commits from two well-known software projects QT [19] and OPENSTACK [16]. The dataset contains 25,150 commits from the QT project and 12,374 commits from the OPENSTACK project. Table 2.1 presents summary statistics of the dataset.

Table 2.1: Summary of the dataset

Dataset	**Timespan**		**Commits**	
	Start	**End**	**Total**	**Defect**
OPENSTACK	11/2011	02/2014	12,374	1,616 (13%)
QT	06/2011	03/2014	25,150	2,002 (8%)

2.3.2 Characteristics Selection

JIT defect prediction techniques predict whether a commit will introduce defects in the future by identifying the most likely defective commits [143]. We focused on extracting character-

Table 2.2: Commit characteristics used in this study

Commit characteristics	**Definition**
File Count	Number of changed files that contain non-comment and non-blank-line edits
Edit Count	Number of lines edited that are non-comment or non-blank
Multiline Com--ments Count	Number of new added multiline comment chunks
Outward Depen--dency Sum	Total number of dependents modified files are depended on.
Inward Depen--dency Sum	Total number of dependents depending on the modified files.

istics relevant to a commit since JIT prediction is performed after every new commit. We relied on the study conducted by Motwani et al. [132] for this purpose. They conducted a comprehensive study to identify characteristics that are important for developers while fixing a bug. Some of the characteristics are not available when a new commit is pushed into the code base, such as *Time to fix, Priority, Reproducible, Triggering test count* etc. Therefore, all defect characteristics proposed by Motwani et al. [132] can not be directly used in our analysis. The first and second authors carefully examined each characteristic mentioned by Motwani et al. and selected the characteristics applicable to JIT defect prediction after reaching a complete agreement. Our selected characteristics are listed in Table 2.2.

To extract *Edit Count* and *File Count*, we first collected the information (e.g., added code lines, deleted code lines, the names of the changed files) of commits in the dataset from Github by using Github API[1]. However, the information gathered contains modified files that only modified comments. According to [132], comment changes play a negative role in characterizing changes since they do not affect program behaviors. Therefore, we designed *regular expressions* to filter out the modified files that only edited comments. The resulting *Edit Count* is the sum of the number of non-comment added and deleted code lines, while *File Count* is the number of changed files in a commit that contain at least one line of non-comment code change. In addition, *git diff -w* command is used to ignore whitespace

[1]https://docs.github.com/en/rest

Table 2.3: Regular expression implemented to filter out comments

Single Line Comment (C/C++)	"(^[+-][[:blank:]]*\/\/)—(^[+-][[:blank:]]*$)"
Multi Line Comment (C/C++)	"\s*(\/*)(.*?)*\/"
Single Line Comment (Python)	"(^[+-][[:blank:]]*#)—(^[+-][[:blank:]]*$)"
Multi Line Comment (Python)	"\s*([\'\"])\1\1(.*?)\1{3}"

differences between commits and their parents to ensure blank line changes do not impact the counting of *Edit Count* and *File Count*. The detailed implementation of the *regular expressions* is showed in Table 2.3.

Prior research identified comments when modified with source can act as a significant feature for defect prediction [97]. Thus, we also want to explore if multiline comment blocks in the source code can impact the performance of JIT defect prediction. So we extracted the number of multiline comment blocks as *Multiline Comments Count*.

To ensure the reliability of results for *Inward Dependency Sum* and *Outward Dependency Sum*, we used a widely adopted static analysis tool, Understand[2]. The tool analyzes every reference in a project and builds dependency data structures for every file and architecture. This includes the nature of the dependency and the references that cause the dependency. Therefore, we summed all files that the edited files depend on and summed files that depend on the edited files as *Outward Dependents Sum* and *Inward dependents Sum*, respectively.

2.3.3 Investigating Difference in Characteristics between Correct and Incorrect Prediction

Next, we investigated the impact of the selected characteristics on prediction. We replicated DeepJIT [78] and CC2Vec [79] with their original training and testing datasets so that we can compare the characteristics' impact on these two techniques.

[2]https://www.scitools.com/

DeepJIT [78] relies on both commit message and code change for prediction. We encoded commit messages and code changes and fed them into the input layer of two separate Convolutional Neural Network (CNN) [111] for feature extraction. Then, the two extracted feature vectors were concatenated to form a unified feature representation. The new vector is then fed into a fully connected layer, which outputs a probability score for a given commit being defective.

We followed the process described by Hoang et al. [79] to replicate CC2Vec. Specifically, we took information from the code change of the given commit as an input and output a list of files, including a set of removed code lines and added code lines. Each changed file is then encoded as a three-dimensional matrix to be given as input to a hierarchical attention network (HAN) for extracting features. The resulting features are then concatenated to form a vector representation of the code change. Then, we map the vector representation of the code change to a word vector extracted from the log message; the word vector indicates the probabilities with which various words describe the commit. Finally, we concatenated the vector representation of the code change extracted by CC2Vec with two embedding vectors extracted from the commit message and code change to form a new feature, which is fed into DeepJIT's feature combination layers to predict if a commit is buggy.

Next, we split the classification results into correctly and incorrectly classified groups. Then for each characteristic, we compare between correctly and incorrectly classified groups. Since we perform multiple tests, we have to adjust the significance value accordingly to account for multiple hypothesis correction. We use the Bonferroni correction [33], which gives us an adjusted p-value of 0.01. For all five characteristics, we find significant differences (Mann-Whitney test, α¡0.01) between the means of correctly and incorrectly classified commits. We use the non-parametric Mann-Whitney test since our population is not normally distributed. We also calculated Cliff's Delta between the mean values to check the effect size, where a delta less than 0.147 is considered "negligible", less than 0.33 is considered "small", less than

0.47 is considered "medium", and a delta greater than 0.47 is considered "large" [152].

2.3.4 Investigating Impact of Characteristics on Prediction

To measure how much the characteristics affect the performance of DL defect prediction techniques, we leveraged the mean values of characteristics identified for correctly and incorrectly classified commits in the previous step.

We posit that for any characteristic, a value close to the mean value of the wrongly classified group will have a more negative impact on prediction performance. Whereas a commits characteristic value close to the mean value of a correctly classified group does not have a significant negative impact on DL predictions.

Following the above intuition, we divided OPENSTACK and QT datasets using the previously calculated mean values for correctly and incorrectly predicted groups, separately for each characteristic. In this study, we tried multiple threshold calculation approaches, such as calculating the mean of mean or median of mean values. However, through our empirical investigation, we found that predictions based on different threshold calculation approaches were similar because the difference between the mean values of characteristics for correctly and incorrectly classified groups was big. So we use the median of mean values for calculating the threshold using the following equation:

$$Threshold = median[\sum_{i}\sum_{j} mean(CVC_{m^i}^{d^j}) + \sum_{i}\sum_{j} mean(CVW_{m^i}^{d^j})] \tag{2.1}$$

In the equation above, CVW is a Characteristic Vector for Wrongly classified data, CVC is a Characteristic Vector for Correctly classified data, m is DL model (i.e., DeepJIT, CC2Vec),

Table 2.4: Training and testing data of OPENSTACK dataset calculated based on thresholds

Characteristics	Divide Threshold	Smaller than threshold (train/test)	Bigger than threshold (train/test)
Edit Count	143.35	84.29%/82.34%	15.71%/17.66%
File Count	5.68	85.84%/84.07%	14.16%/15.93%
Multi-line Comments Count	8.84	87.92%/86.03%	12.08%/13.97%
Inward Depen--dency Sum	22.81	81.45%/79.79%	18.55%/20.21%
Outward Depen--dency Sum	46.78	77.75%/74.98%	22.25%/25.02%

d is dataset (i.e., OPENSTACK, QT). For example, mean values calculated for *File Count* characteristic in OPENSTACK data is 2.71 for correct classification, and 8.58 for wrong classification on DeepJIT (Table 2.6). For CC2vec, the values are 3.09 for correct classification and 8.28 for incorrect classification. Thus, the threshold of *File Count* characteristic in the OPENSTACK dataset is 5.68. The thresholds for each characteristics for OPENSTACK is shown in second column of table 2.4, and for QT is in Table 2.5.

Table 2.5: Training and testing data of QT dataset based on thresholds

Characteristics	Divide Threshold	Smaller than threshold (train/test)	Bigger than threshold (train/test)
Edit Count	247.35	92.25%/93.38%	7.75%/6.62%
File Count	13.22	94.39%/94.32%	5.61%/5.68%
Multiline Comments	58.13	93.37%/93.81%	6.63%/6.19%
Inward Depen--dency Sum	71.71	88.23%/86.84%	11.77%/13.16%
Outward Depen--dency Sum	69.24	82.41%/84.11%	17.59%/15.89%

After splitting, we observe that for each characteristic, the group with values smaller than the threshold always occupies the majority parts of data (third column in table 2.4 and table 2.5). For example, in the case of *File Count*, the group below threshold in OPENSTACK contains 84.29% of training data and 82.34% of testing data. The majority group based on *Outward Dependency Sum* occupies a relatively low percentage of total data, but it still occupies

77.75% of training and 74.98% of testing data. A similar pattern is observable for the QT dataset. Table 2.5 shows that the data below the *File Count* threshold occupies 94.39% of all QT training data and 94.32% of all QT testing data. Thus, we call the divided data below the threshold as *Majority Class*. Furthermore, the group with a value bigger than the threshold is named as *Few-shot Class* since a smaller part of the dataset belongs in this group. Next, we trained both DeepJIT and CC2Vec for both of the *Majority Class* and *Few-shot Class*. We did the training for each characteristic and compared the AUC score to investigate the impact of characteristics on prediction. To deal with variance in DL prediction results, we validated each dataset 15 times and reported the mean values in the results.

2.3.5 Improving Defect Prediction Considering Few-shot class

SifterJIT aims to improve the state-of-the-art DL defect prediction models, specifically by improving the prediction on the few-shot class. The intuition behind our approach is that focusing on the few-shot class will help improve the overall performance since DL models tend to perform worse for the few-shot class compared to the majority class when trained together.

The SifterJIT schema is shown in Figure 2.1. First, we divide the training dataset into majority and few-shot classes based on their characteristics (Section 2.3.4). Then, we trained a Siamese network on the few-shot class since state-of-the-art DL models (DeepJIT, CC2Vec) perform poorly on the few-shot class as they have fewer training instances compared to the majority class. We selected the Siamese network since Koch et al. [101] and Neculoiu et al. [135] showed that Siamese networks are suitable for few-shot learning where a little data is available. The trained Siamese network was then used for testing commits that belonged to the few-shot class. For the majority class, we continue using DeepJIT since it is a state-of-the-art technique. Below we present the details of Siamese networks and SifterJIT.

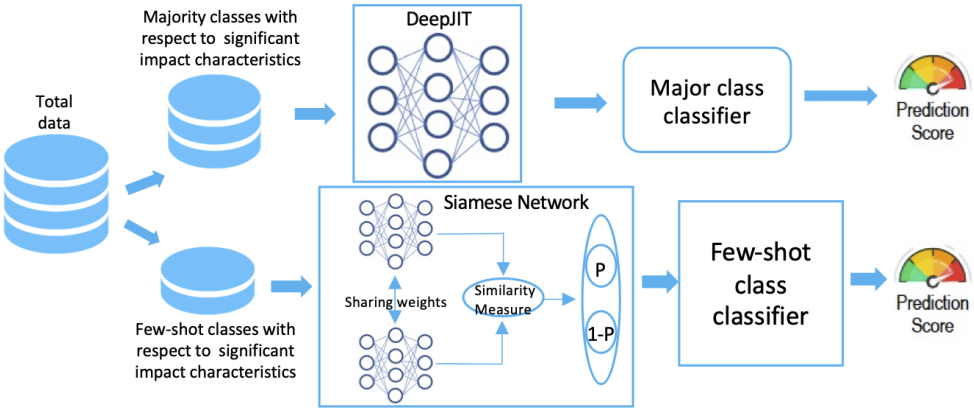


Figure 2.1: Overview of SifterJIT

Siamese network

The Siamese network consists of two identical base networks which process the same training instances in pairs. However, the weights of the two networks are shared. This model accepts distinct inputs and joins them by a similarity measure function. This similarity measure function measures the distance d_i between the learned features h_1 and h_2 on each side. Figure 2.2 shows a Siamese network with three hidden layers, and each layer contains two neurons. The depicted Siamese network performs binary classification with a similarity function $s = \sum_{i=1}^{n} d_i$, where n is the number of learned attributes.

Similarity measure

Siamese network measures the distance between learned features on each side. If X_1 and X_2 are two input vectors, w represents shared parameter vector, and the mapping of X_1 and X_2 in the feature space are represented by $H_w(X_1)$ and $H_w(X_2)$. Then the Siamese networks can be considered as a scalar similarity function $D_w(X_1, X_2)$ to measure the distance between X_1 and X_2, and the distance is defined as:

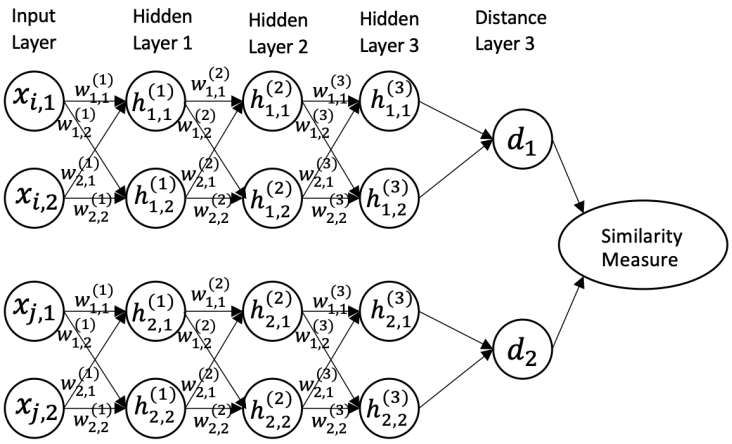


Figure 2.2: Siamese network with three hidden layers

$$D_w(X_1, X_2) = ||H_w(X_1) - H_w(X_2)||$$

SifterJIT uses Euclidean distance to learn the metric of similarity features from input pairs of data.

Loss function

: For the Siamese network loss function, we use the most popular Contrast loss function [182]. The loss function is defined as:

$$L_{contr}(w, y, X_1, X_2) = \frac{1}{N}\sum_{i=1}^{N}((1 - y_i) * (D_w^{(i)2} + y_i * (max(m - D_w^{(i)}, 0))^2) \quad (2.2)$$

where y is a binary label, $y = 0$ if a pair of data(X_1, X_2) belongs to the same class and $y = 1$ if it is different. $m > 0$ is a pre-set threshold, D_w is the Euclidean distance. The minimum of $L_{contr}(w, y, X_1, X_2)$ will decrease D_w when pairs of data come from the same character class and increase D_w when pairs of data come from a different class. More concisely, the

minimization of $L_{contr}(w, y, X_1, X_2)$ would result in low values of D_w for similar pairs and high values of D_w for dissimilar pairs. Using this loss function, two commits will have low Euclidean distance if they introduce similar defect, and non-defect introducing commits have large Euclidean distances with a defect introducing commits.

SifterJIT Model Training

The SifterJIT's base network is similar to DeepJIT [78], which includes a convolutional layer with multiple filters and a nonlinear activation function (i.e., Relu). This paper uses a normal distribution with zero mean and a standard deviation 0f 10^{-2} to initialize all neural network weights. We set the dimension of word vectors and the number of filters to 64 since Hoang et al. [78] showed they got the best performance at these values. We set the batch size to 32 and the size of the fully connected layer to 512. Since our goal was to compare performance, we used same hyperparameters settings that were used by prior work [80, 78].

Data Oversampling:

To evaluate if SifterJIT can outperform state-of-the-art techniques, we compared with original DL defect prediction techniques and applied oversampling on the dataset for original DL techniques. Previous studies [125, 64] show that oversampling is an effective approach to improve performance on imbalanced data. Thus, we also compared SifterJIT with models trained on over-sampled data. SMOTE [41] has been proved as one of the most popular oversampling techniques. SMOTE works by selecting examples close in the feature space, drawing a line between the examples in the feature space, and drawing a new sample at a point along that line [41]. However, the input data we used is the representation of code changes and commit messages, and it is difficult to draw a meaningful line between data samples of this type. Thus, we did not use SMOTE for oversampling; instead, we imple-

mented a random oversampling scheme [64]. The random over-sampling technique randomly duplicates examples from the few-shot class and adds them to the training dataset [64]. We applied oversampling on DeepJIT and CC2Vec but not on SifterJIT. Because SifterJIT compares pairs of inputs and predicts via calculating their distance. If random over-sampling with SifterJIT is used, the duplicates will have no distance between them, and it will not improve SifterJIT's performance. On top, it will increase training time. So we did not use over-sampling with SifterJIT.

Evaluation Metric:

We report the standard precision, recall, and AUC (Area Under the receiver operating characteristic Curve) to assess the performance of the prediction models because it is independent of prior probabilities [30]. Also, AUC is a better measure of classifier performance than accuracy because it is not biased by the size of test data. Moreover, AUC provides a "broader" view of the performance of the classifier since both sensitivity and specificity for all threshold levels are incorporated in calculating AUC. Other work related to JIT prediction have used AUC for comparison purposes [55, 66, 67, 196, 78, 79].

We list the formula used for calculating precision, recall, and F-measure below. AUC Computes the area under the curve plotting the true positive rate against the false positive rate, while applying multiple thresholds to determine if a commit is defective or not. The AUC curve is created by plotting the recall against the false positive rate (FPR) at various threshold settings.

- **Precision (P)**: A measure of whether the commits classified as *defect* are *actually defective commits*.

$$precision = \frac{t_p}{t_p + f_p} \tag{2.3}$$

- **Recall (R)**: A measure of the percentage of *defect* instances that the approach managed to correctly predict.

$$recall = \frac{t_p}{t_p + f_n} \tag{2.4}$$

- **F1 score (F1)**: The F1 score is the harmonic mean of the precision and recall.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{2.5}$$

- **False positive rate (FPR)**: A measure of the ratio of the number of *defects* wrongly categorized and the total number of actual *defect* commits.

$$FPR = \frac{f_p}{f_p + t_n} \tag{2.6}$$

2.4 Results

Here we discuss the results of our study by placing them in the context of three research questions, which investigate the impact of characteristics on prediction performance (RQ1), which characteristic affects prediction performance the most (RQ2), and whether we can improve the prediction of defects by explicitly considering few-shot classes identified using the aforementioned characteristics (RQ3).

2.4.1 RQ1: Do commit characteristics have an impact on defect prediction performance?

To answer this question, we replicated the work of DeepJIT and CC2Vec using their original training and testing data. Then we split the classification results into two groups based on whether they were correctly or incorrectly classified. Next, we investigate if characteristics (i.e., *Edit Count*, *File Count*, *Multiline Comments Count*, *Inward Dependency Sum*, and *Outward Dependency Sum*) have any impact on classification performance. The calculated mean characteristic values for correctly and incorrectly classified instances are shown in table 2.6 and in table 2.7. For example, in terms of *File Count* characteristic, the mean number of the modified files is 2.71 for the correctly classified group and 8.58 for the incorrectly classified group.

The next column on table 2.6 and 2.7 shows the P-values from the Mann-Whitney test, indicating whether there is a statistically significant difference between the characteristic's value between correct and incorrect classification groups. Our results indicate that for OPENSTACK, the distribution of mean values between correctly and incorrectly classified groups are significantly different for all characteristics (Table 2.6). The Ciff's Delta of the these characteristics' mean values between correctly and wrongly classified data is 0.42, when using DeepJIT and 0.38, when using CC2Vec, both of which show a medium difference in effect size. However, for the QT dataset, only *Outward Dependency Sum* has a statistically significant difference between correctly and incorrectly classified groups (Table 2.7).

Table 2.6: Comparison between correct and incorrect prediction's mean values of characteristics in OPENSTACK dataset. * indicates statistical significance.

Dataset	OPENSTACK					
Model	DeepJIT			CC2vec + DeepJIT		
	Correct Classifi--cation	Wrong Classifi--cation	P-value	Correct Classifi--cation	Wrong Classifi--cation	P-value
File Count	2.71	8.58	**9.64e-14***	3.09	8.28	**1.95e-09***
Edit Count	65.71	227.26	**3.38e-13***	82.03	204.68	**2.36e-10***
Multiline Comments	3.50	15.04	**3.40e-08***	4.94	12.74	**3.26e-05***
Inward Dep--endents Sum	12.56	33.30	**1.92e-08***	14.28	31.34	**1.99e-05***
Outward Dep--endents Sum	27.67	64.36	**3.6e-12***	29.41	64.16	**1.97e-09***

Table 2.7: Comparison between correct and incorrect prediction's mean values of characteristics in QT dataset. * indicates statistical significance.

Dataset	QT					
Model	DeepJIT			CC2Vec + DeepJIT		
	Correct Classifi--cation	Wrong Classifi--cation	P-value	Correct Classifi--cation	Wrong Classifi--cation	P-value
File Count	4.88	21.57	0.03	4.79	27.22	0.12
Edit Count	168.98	325.72	0.14	153.90	585.26	0.14
Multiline Comments	41.80	74.47	0.28	34.83	150.52	0.06
Inward Depend--ents Sum	66.55	198.44	0.08	62.49	76.88	0.21
Outward Depen--dents Sum	48.13	151.16	**1.98e-07***	40.53	90.36	**6.79e-09***

2.4.2 RQ2: Considering different commit characteristics, which one affects defect prediction performance the most?

To answer this research question, we needed to measure how much the characteristics affect the defect prediction performance. After training the state-of-the-art DL defect prediction techniques using their original training data, we evaluated the models on all testing data. Their AUC score is shown in table 2.8. Then, we divided training and testing data into two groups using the threshold shown in table 2.4 for OPENSTACK and table 2.5 for QT. We call the group with values less than the corresponding threshold as *Majority class* since they

Table 2.8: The AUC results on all testing data

	OPENSTACK	QT
DeepJIT	75.1	76.8
CC2vec + DeepJIT	80.9	82.2

always occupied most of the data. Another group that is bigger than the threshold is named as *few-shot class* since they always contain a small portion of the data.

From Table 2.9, we can observe that few-shot classes with respect to *Edit Count, File count, Multiline Comment Count, Inward Dependency Sum* and *Outward Dependency Sum* have a significant performance drop. When using the DeepJIT technique on OPENSTACK, the AUC score on few-shot classes is between 52.4 to 65.3, a 22.7 drop compared to the AUC achieved when all data is used (shown in table 2.8). Interestingly, the AUC scores on majority classes increased, ranging between 76.8 to 79.1. When using CC2vec, for the majority classes AUC score ranged between 80.1 and 83.2 (Table 2.9). Similar to DeepJIT, in the case of CC2vec, few-shot classes saw a significant performance drop with AUC score ranging between 64.9 to 68.3. This is a 10.2 average drop compared to the AUC achieved when all data is used in table 2.8. We also checked whether the difference in AUC score between the majority class and the few-shot class is statistically significant. The results are statistically significant for both DeepJIT (Mann-Whitney test, p¡0.011) and CC2vec (Mann-Whitney test, p¡0.008).

Table 2.10 shows that for QT, DeepJIT have similar drop in AUC scores on few-shot classes for *Inward Dependency Sum* characteristic. For *Edit Count, File Count, and Multiline Comment Count* the drop is comparatively lower. However, both techniques have significantly worse performance on the few-shot class than the majority class with respect to *Outward Dependency Sum* characteristics. We also checked whether the difference in AUC score between the majority class and the few-shot class is statistically significant. The results are statistically significant for CC2vec (Mann-Whitney test, p¡0.007). However, for DeepJIT the results are not statistically significant(Mann-Whitney test, p¡0.119).

Table 2.9: AUC variance of divided classes on OPENSTACK

	DeepJIT			**CC2vec + DeepJIT**		
	Few-shot class	Majority class	Delta	Few-shot class	Majority class	Delta
Edit Count	55.2	76.8	21.6	65.1	80.1	15
File Count	59.1	78.2	19.1	67.2	81.2	14
Multiline Com--ment Count	52.4	78.3	25.9	66.8	80.3	13.5
Inward Depend--ents Sum	65.3	79.1	13.8	68.3	82.9	14.6
Outward Depend--ents Sum	58.2	78.3	20.1	64.9	83.2	18.3

Table 2.10: AUC variance of divided classes on QT

	DeepJIT			**CC2vec + DeepJIT**		
	Few-shot class	Majority class	Delta	Few-shot class	Majority class	Delta
Edit Count	65.9	73.2	7.3	74.5	83.2	8.7
File Count	64.8	73	8.2	73.2	82.9	9.7
Multiline Com--ment Count	70.1	74.3	4.2	75.5	84.2	8.7
Inward Depend--ents Sum	76.7	74.8	-1.9	80.2	84.5	4.3
Outward Depend--ents Sum	59.3	74.8	15.5	64.1	83.5	19.4

From our results, we see that DL techniques are effective on majority classes since their AUC scores are close to aggregated classes AUC in table 2.8. However, few-shot classes have poor classification performance, and several of them even close to random classification since AUC scores of few-shot classes for *Multiline Comment Count* and *Edit Count* are 52.4 and 55.2 when using DeepJIT, which are close to 0.5.

When ranking the characteristics based on the total drop in AUC, we see that *Edit Count, Multiline Comment Count, and Outward Dependency Sum* are the top three characteristics affecting the performance most for OPENSTACK. However, for QT, *Edit Count, File Count, and Outward Dependency Sum* are the top three characteristics affecting the performance negatively.

Table 2.11: Prediction Performance Comparison on OPENSTACK few-shot classes

	DeepJIT + CC2vec (Original)	DeepJIT + CC2vec (Oversampling)	SifterJIT	SifterJIT - Original	SifterJIT - Oversampling
AUC Score (%)	57.88	60.39	69.19	11.31	8.80
Precision (%)	32.09	33.33	42.57	10.48	9.24
Recall (%)	95.56	95.56	65.15	-30.41	-30.41
F1 Score (%)	47.43	48.99	51.50	4.07	2.51

2.4.3 RQ3: How well can DL techniques predict defects by explicitly considering few-shot classes?

Previous research questions show that both DeepJIT and CC2vec perform poorly on few-shot classes. Thus, to improve prediction performance on these few-shot classes, we propose a Siamese network-based few-shot learning framework for JIT defect prediction (SifterJIT).

Table 2.11 shows the comparison of defect prediction results on the OPENSTACK dataset using CC2vec with and without random oversampling and SifterJIT framework. From this table, we can see that SifterJIT improves the AUC score on few-shot classes from 57.88% to 69.19% (an improvement of 11.31%). SifterJIT also improves precision and F1 score by 10.48% and 4.07%. Oversampling also improves compared to the original performance, but the improvement is only 2.51% for AUC, 1.24% for precision, and 1.56% for F1 score. The SifterJIT's recall is worse than the original and oversampling. A closer look reveals that SifterJIT is more conservative than original and oversampling methods, and since it predicts less number of samples as "defect", the recall is lower. Prior study shows that too many false positive warnings can discourage developers from using a tool [77]. Thus, it is important to reduce the false positives.

To better understand the improvement, we looked into the AUC distribution of few-shot classes for individual characteristics, shown in Figure 2.3. From Figure 2.3 we can observe that SifterJIT outperforms original and oversampling results for most characteristics.

Table 2.12: Prediction Performance Comparison on QT few-shot classes

	DeepJIT + CC2vec (Original)	DeepJIT + CC2vec (Oversampling)	SifterJIT	SifterJIT - Original	SifterJIT - Oversampling
AUC Score (%)	63.14	64.71	69.12	5.98	4.41
Precision (%)	25.53	25.79	35.48	9.95	9.69
Recall (%)	88.33	90.00	63.16	-25.17	-26.84
F1 Score (%)	38.74	39.20	45.28	6.54	6.08

Table 2.12 shows the aggregated defect prediction results on QT dataset. From the table, we can see that the SifterJIT classification AUC score improves from the original's 63.14% to 69.12% (5.98% improvement), and SifterJIT also improves the precision by 9.95% and F1 score by 6.54%. Compared to oversampling, the AUC score increased by 4.41%, and it improves precision by 9.69% and F1 by 6.08%.

Figure 2.4 shows the AUC score distributions for few-shot classes for individual characteristics. From the figure, we can see that SifterJIT outperforms for characteristics *Outward Dependency Sum* and *File Count*. However, unlike the OPENSTACK dataset, SifterJIT did not outperform in the case of the other three characteristics, even though the overall improvement using SifterJIT was non-trivial as shown in Table 2.11 and Table 2.12.

2.5 Discussion

To the best of our knowledge, we are the first to investigate whether and to what extent the imbalance of commit characteristics impacts JIT defect prediction. We find that characteristics such as *File Count, Edit Count, Inward Dependency Sum, Outward Dependency Sum, Multi-line Comment Count* are some of the characteristics that impacted the performance of the classifiers significantly, up to 25.9%.

Our analysis finds that along with other characteristics *Inward Dependency Sum, Outward Dependency Sum* impact the performance of the classifiers. These two factors are related to

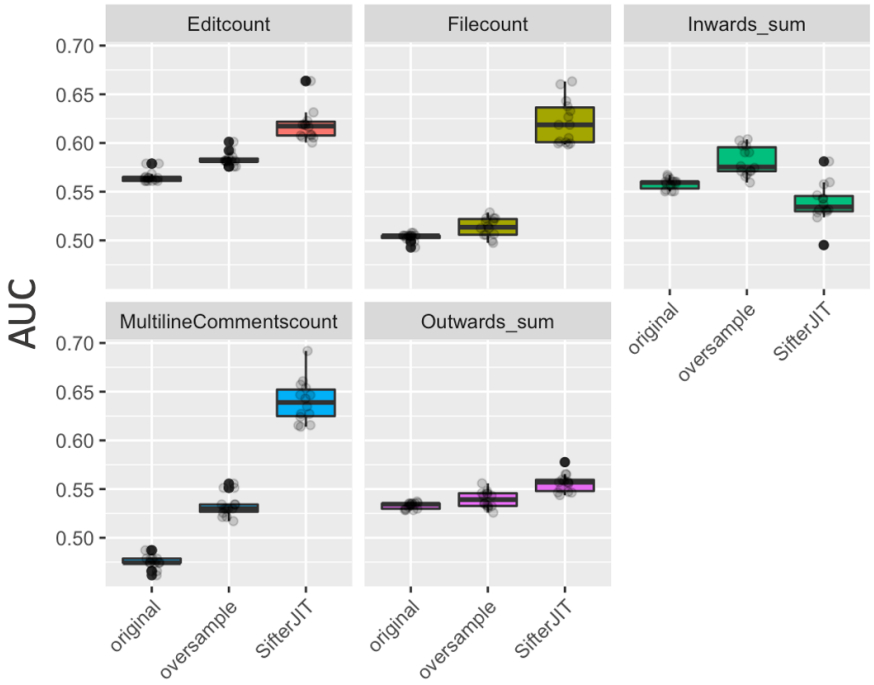


Figure 2.3: AUC distribution for different characteristics using OPENSTACK

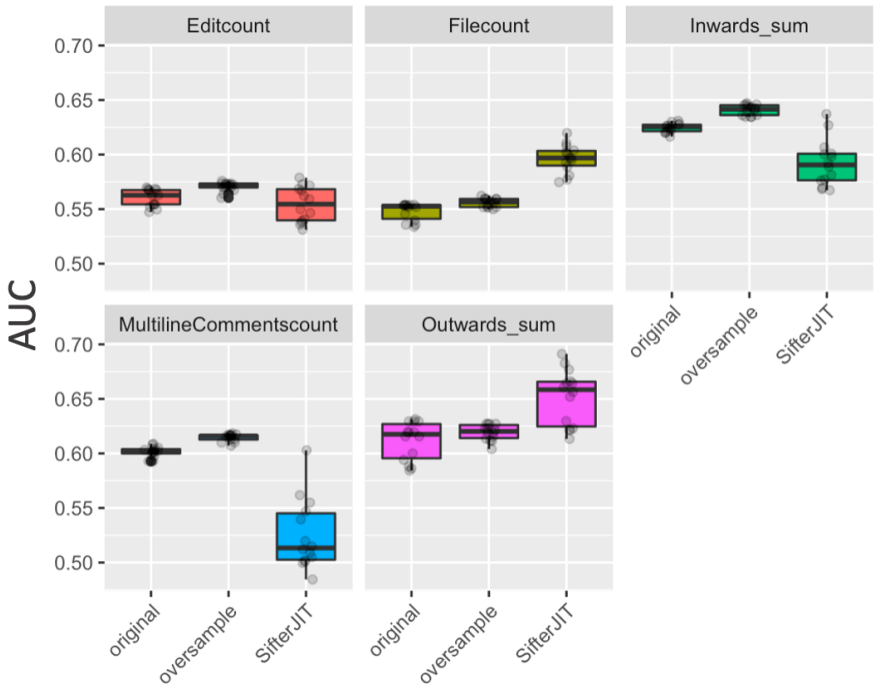


Figure 2.4: AUC distribution for different characteristics using QT

dependency. We posit that a commit that modifies files with high dependency is likely to be highly coupled with other parts of the code and has a high impact. Our results also identified that the *Edit Count* significantly impacts the performance of the classifiers. Since classifiers are sensitive to change size [131, 90, 128], such impact of *Edit Count* is not surprising. Since we studied a small number of characteristics, one important direction for researchers is to identify other characteristics and investigate their impact.

One interesting finding is that not all characteristics equally impact the classifier's performance across all datasets. Our results in RQ2 indicate that a characteristic can have varying levels of impact, even for the same technique depending on the dataset. For example, *Multi-line Comment Count* for DeepJIT applied on OPENSTACK resulted in a 25.9% drop in AUC, however, for QT dataset, it resulted in a 4.2% drop, as shown in Table2.10. Another interesting observation is that the impact is not always negative. For example, in case of *Inward Dependency Sum*, few-shot class had a higher AUC compared to majority class as shown in Table2.10. Further investigation is required to understand the underlying reason for this. Also, the varying impact of characteristics can be leveraged to examine different ranking schemes. An effective ranking scheme can help practitioners prioritize their effort to more impactful characteristics when trying to minimize the imbalance.

Our results also highlight that different DL techniques have varying resistance to the imbalance of commit characteristics. Table2.9 and 2.10 show that CC2vec on an average is more resistant to the imbalance which is backed by the Mann-Whitney test (U = 22, p-val = 0.03756). This similar to other researchers' findings where they showed that different machine learning classifiers have varying resistance to noise [99].

Our findings have implications for software practitioners and tool builders as well. Practitioners should pay more attention to the imbalance of commit characteristics to achieve the best prediction performance, which would allow them to save their effort while sifting through incorrect predictions. Also, it will reduce the number of bugs making it to the

production system.

There are tools to detect data imbalance issues for normal machine learning tasks to avoid unfairness issues, such as in computer vision and natural language processing [21, 1]. However, to the best of our knowledge, no such tools exist for defect prediction tasks. Thus, it is also necessary to build tools to detect data imbalance in the defect prediction dataset.

2.6 Threats to Validity

We have taken care to ensure that our results are unbiased and have tried to eliminate the effects of random noise, but it's possible that our mitigation strategies may not have been effective.

Bias Due to Dataset: Our findings may not generalize to all software projects since we evaluated our approach on two datasets (QT and OPENSTACK). However, we evaluated our approach on a publicly available dataset that has been used in previous JIT defect prediction research [78, 79, 143, 128]. On top, our considered projects are large and significantly different in size, programming language, complexity, and revision history. So we believe that the selected projects adequately address the concern.

Bias Due to Characteristics: Our set of characteristics are selected from literature [132]. However, results may differ depending on the characteristics used for evaluation. Also, we did not compare our results with the effects of other widely-used software metrics for ML defect prediction models.

Bias Due to Threshold Selection: Our threshold selection in RQ2 may threaten the internal validity. In order to mitigate this, we empirically investigated different formulas for threshold calculation and found that the results do not significantly differ.

Bias Due to Implementation: To mitigate this bias, we reused existing implementations of the *DeepJIT* and *CC2Vec* techniques whenever possible. We also tested our code and data to ensure that there are no implementation errors; however, errors may remain. In addition, the *regular expressions* used to identify comments might fail to identify all types of comments in the source code.

2.7 Conclusions and Future Works

In this paper, we investigated whether and to what extent commit characteristics can impact JIT defect prediction. Our results show that the performance of DL techniques got negatively impacted by the imbalance of commit characteristics. DeepJIT's performance on OPENSTACK dropped down to 52.4% compared to the original performance of 75.1% (22.7% drop). A similar pattern was observed for CC2Vec, where performance dropped down to 65.1% compared to the original 80.9 (19.53% drop). On the QT dataset, DeepJIT's performance dropped down to 59.3% from 76.8% (17.5% drop), CC2vec's performance dropped to 64.1% from 82.2% (18.1% drop)

To improve their overall performances, we propose a Siamese-based Few-shot learning framework named SifterJIT. Our choice of investigating the Siamese network was motivated by the Siamese network's power to learn from a limited number of training instances, which can help to boost the model performance on few-shot classes with respect to commit characteristics and boost the overall model performance. Our results show that SifterJIT outperforms state-of-the-art CC2vec by an improvement of 11.31% AUC score, 11% improvement in precision, and 5% improvement in F1-score on OPENSTACK dataset. Similar improvements were seen for the QT dataset with a 5.98% improvement of AUC score, 10% improvement of precision, 6% improvement of F1-score.

In this work, we analyzed five characteristics. However, prior defect-prediction research has identified a plethora of characteristics which is yet to be investigated. Our results identify the need for further research to understand the impact of the imbalance of these characteristics on the prediction model's performance and understand the underpinning of why different characteristics have varying levels of impact.

Chapter 3

Leveraging Feature Bias to Interpret Model Misprediction

3.1 Introduction

Machine learning (ML) techniques, similar to other fields, have been gaining popularity in software engineering tasks. Defect prediction [78, 79, 62], automatic code completion [44, 178], predicting merge conflicts [139], and synthesizing and repairing programs [195, 189, 185, 159] are some examples. While these models' overall performance is good, interpreting and debugging them is a challenge, which also impedes the real-world usage of these models [52, 114].

Specific characteristics of ML systems make them difficult to debug. The opacity of the learned models, high dimensionality of the input data, dependence on the data quality [28, 38] are a few of them. Data often exhibits highly-skewed class distributions (class imbalance), i.e., most data belong to the majority class, and the minority class only contains a small number of instances [180]. To complicate things even more, imbalance not only

happens at class level but also on data features [62]. Since ML models are usually trained by minimizing average training loss on all data, which is also known as Empirical Risk Minimization (*ERM*), a feature imbalance can lead to models that achieve low test error but still incur high error on instances that contain under-represented features. For example, Gesi et al. [62] showed that in software defect prediction tasks, comparing with most commits, the prediction model often performs significantly worse for the commits, which involve a large number of modified files since the number of training instances with a large number of modified files is very few during training. The similar situation has been observed in other fields as well, such as a vehicle recognition model usually fails to detect crashed cars as a car because of very few crashed car instances in the training dataset [194]. These varying granularities of imbalance (i.e., class vs. feature) severely impact the robustness of models.

To ensure the robustness of the models, the explanation generation technique has been proven to be one of the most effective ways as it can help in explaining the rationale for a prediction [151, 160, 47, 48, 61]. Researchers have been trying various explanation generation techniques [151, 160] to shed light on the global behavior of a model either by highlighting which features are the most important or by constructing a surrogate and simpler model that emulates a complex model. However, except for [46, 45], none of the work focused on explaining the mispredictions of a ML model.

In the most recent work, Cito et al. [46] proposed a technique named *EXPLAIN*, which generates a set of decision rules based on features and mispredicted instances to explain the reasons for mispredictions, i,e., Misprediction Explanation (*ME*) rules. However, one of the limitations of *EXPLAIN* is that its generated *ME* rules are deduced "blindly" from all features. Since ML models can have thousands or even millions of features [20], without guiding the ME rule generation by incorporating some form of prior knowledge, techniques like *EXPLAIN* will suffer from scalability issues. Furthermore, due to data and model drift over time [104], models must be retrained, and ME rules also must be regenerated. Additional

time requirements for approaches "blindly" relying on all features for rule deduction would quickly add up when done many times over the lifetime of a model.

In another related line of work, researchers introduced various methods to improve the model's prediction performance on the instances containing under-represented features [62, 194]. However, previous approaches typically require additional annotations [46]. For example, adding additional annotated code commits that modify a large number of files or adding annotated crashed car pictures in the vehicle detection dataset. While these approaches have been successful at improving the model's performance for instances containing under-represented features, the required additional annotated training data is often expensive [117].

Having Observed these limitations of the existing techniques, in this paper, we propose a technique called **B**ias **G**uided **M**isprediction **D**iagnoser (**BGMD**), which leverages feature imbalance as prior knowledge for generating rules to explain misprediction. Then, we use generated rules from *BGMD* to guide a novel upweight sampling method that can improve ML model's performance on mispredicted data without requiring additional annotated instances, named ***MAPS*** (**M**ispredicted **A**rea U**P**weight **S**ampling).

Figure 3.1 shows the high-level overview of how *BGMD* and *MAPS* work together to resolve the aforementioned limitations of existing techniques. Figure. 3.1-(a) presents a trained model that classifies black and white points based on two features (x-axis and y-axis coordinates). The model predicts points in green region as black points and white in blue region. Next, *BGMD* identifies two regions (red square area in Figure. 3.1-(b)) that contain instances that are prone to misprediction. Then, *MAPS* improves the weight of instances within the identified regions (Figure. 3.1-(c)) so that the retrained model pays more attention to these part of instances. The retrained model result presents in Figure. 3.1-(d), which could perform better on the instances that were identified by *BGMD*. A detailed description of the *MAPS* algorithm is in Section 3.4.

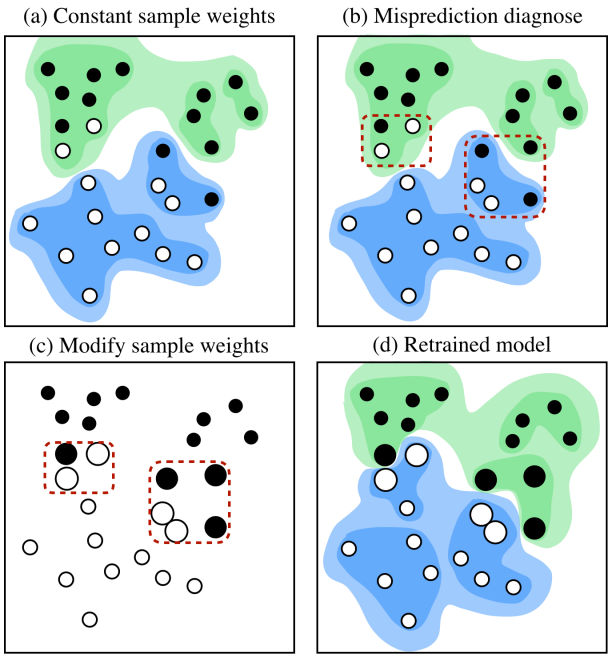


Figure 3.1: Overview of Mispredicted Area Upweight Sampling

We empirically compared *BGMD* with the state-of-the-art *EXPLAIN* [46] technique and the result shows that *BGMD* not only outperformed *EXPLAIN* in generating *ME* rules in trms of rule coverage but also reduced 92% in rule generation time. Furthermore, we also investigated if *MAPS* can successfully improve the ML model's performance, specifically for instances containing under-represented features that are prone to misprediction. We empirically evaluated *MAPS* on three software engineering tasks and five general classification tasks and the result shows that *MAPS* can significantly improve the model's performance without requiring extra annotation data.

The *key contributions* of this study are:

- Introduces a scalable ML model misprediction explanation rule generation technique named *BGMD*.
- Introduces a new upweight sampling method that improves model performance on data prone to be mispredicted without requiring extra annotated training data named

Table 3.1: Samples from a dataset used to train a *ML* model that predicts whether a merge commit is likely to lead conflict

commit num	added file num	parallel changed file num	developer num	...	conflicted	pred
3	7	0	12	...	True	True
1	2	3	4	...	False	False
1	3	2	3	...	True	False
5	13	0	8	...	False	True

MAPS.

- Empirically evaluates new proposed techniques with corresponding state-of-the-art techniques.

The rest work is structured as follows. In Sec. 3.2, we introduce the necessary preliminary information. Then, in Sec. 3.3, we introduce *BGMD*. In Sec. 3.4, we describe how *MAPS* works. In Sec. 3.5, we show empirical evaluations and results. Then, In Sec. 4.6, we make further discussions. In Sec. 3.7, we review some of the related works close to our problem. In Sec. 4.8, we present threats to validity, and finally, in Sec. 4.9, the conclusions are drawn.

3.2 Preliminaries

In this section, we describe what a misprediction explanation (*ME*) rule generation technique is and how the generated rules can be used to explain mispredictions of a *ML* model.

Imagine training a model to predict whether a merge commit is likely to cause a conflict. The model may be based on features such as number of commits ("commit num"), number of added files ("added file num"), number of changed files parallelly ("parallel changed file num"), number of involved developers ("developer num") and potentially dozens of additional features. Table 3.1 provides a small subset of the entire dataset, including the true

label ("conflicted") and the model's prediction ("pred"). We will use it as a running example for the rest of this section.

We use instances $x \in \mathcal{X}$ and corresponding labels $y \in \mathcal{Y}$ to train a *ML* model. Let $\mathcal{D} : \mathcal{X} \to \mathcal{Y}$ be the ground truth for the dataset. Given instances $(x_1, y_1), ..., (x_n, y_n) \in \mathcal{X}$, a trained *ML* model $\mathcal{M}_\theta : \mathcal{X} \to \mathcal{Y}$ parameterized by θ, we define a misprediction indicator $\mathcal{I} : x \to \{0, 1\}$:

$$\mathcal{I}(x) = \begin{cases} 1 & \text{if } \mathcal{D}(x) \neq M_\theta(x) \\ 0 & \text{if } \mathcal{D}(x) = M_\theta(x) \end{cases} \tag{3.1}$$

In other words, $\mathcal{I}(x)$ is 1 iff when the *ML* model $\mathcal{M}_\theta$ predicts the wrong label for instance x.

Misprediction coverage: *ME* technique's goal is to generate a decision list Φ, i.e., *ME* rules. These rules are generated based on model training features. In the case of our running example, these features would be all available features shown in table 3.1. Then *ME* technique generated rules identifies a sub-dataset $\Phi(x)$, in which most of the instances are prone to be mispredicted by the trained model:

$$P\left(\Phi(x) = 1 \mid \mathcal{I}(x) = 1, x \in \mathcal{X}\right) \tag{3.2}$$

We refer to the value of Equation 3.2 as the *ME* coverage of rule Φ where $\Phi(x) = 1$ when the *ME* rule covers an instance x that is mispredicted by the model. The larger value of Equation 3.2 means the more mispredicted instances are explained by decision list Φ. And decision list Φ is composed of a set of rules:

$$\Phi = \{\phi_1 \wedge \phi_2 \wedge ...\phi_n\} \tag{3.3}$$

where ϕ_i is a predicate based on feature i and defined as:

$$\phi \rightarrow x_c = c \,|\, x_c \neq c \,|\, x_n \leq c \,|\, x_n > c \tag{3.4}$$

Where each condition is a conjunction of the atomic predicate of the form "$x\, op\, c$" where x is a feature and c is a variable. The notation x_c indicates categorical features, and x_n indicates numeric features. For example, in the running example, the best rule list is when $\Phi = \{$*commit num > 28 & added file num > 15 & developer num <= 15 & developer > 9*$\}$ which has a precision of 82% and a recall of 46%. This means that 82% of the instances identified by the above-mentioned rule are mispredicted by the model, and the identified instances contain 46% of all mispredicted instances. A good *ME* rule should have a higher misprediction coverage, which means both high precision and recall.

3.3 BGMD: Bias Guided Misprediction Diagnoser

In this section, we present our proposed *ME* rule generation technique *BGMD*. First, we show an example of feature imbalance that occurs in merge conflict prediction datasets [137]. We then introduce how *BGMD* exploits feature imbalances in ML models to achieve scalable *ME*.

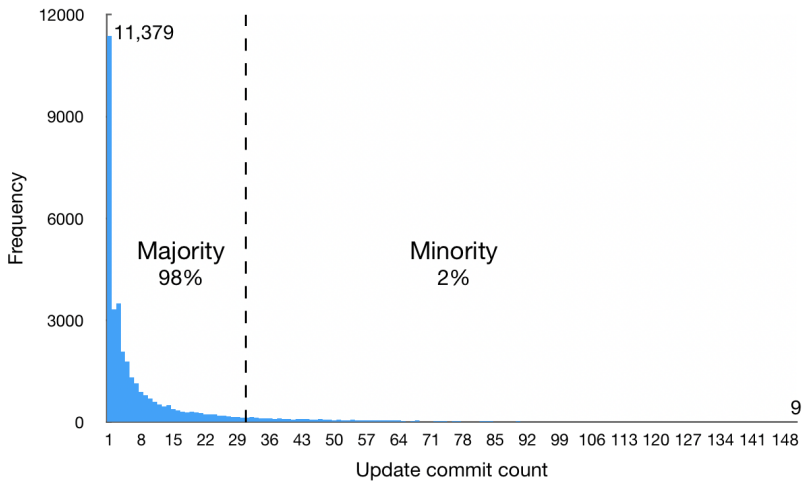


Figure 3.2: Commit count frequency for dataset [137]

3.3.1 Data Feature Imbalance

ML model performance heavily relies on data quality [28]. However, data often exhibit highly-skewed feature distribution. For example, figure 3.2 shows the frequency of the *Updated commit count* feature in a merge conflict prediction data set (we only present the *Updated commit count* between 1 and 150 because of the space limitation).

From figure 3.2, we observe that 11,379 merge commit instances contain one update commit, but only nine merge commit instances have 150 update commits. Additionally, instances with less than 30 *Updated commit count* accounted for 98% of all data. Thus, in the merge conflict prediction dataset, instances with *Updated commit count* less than 30 belong to the majority group with respect to *Updated commit count* feature, while instances with *Updated commit count* over 30 belong to the minority group. The minority group of data is usually under-represented during model training, and as a result, the trained model is biased towards the majority group, causing the model to perform poorly on data containing under-represented features [62, 194]. Despite such bias, these features should not be removed because that might negatively impact the model's overall performance. For example, in case of the data shown in figure 3.2, *Updated commit count* is one of the most important features for merge

conflict prediction [113]. So removing the *Updated commit count* will adversely impact the overall model's performance. Therefore, directly removing the biased features is not advised in literature [41, 117].

3.3.2 Bias Guided Misprediction Diagnoser

The general *ME* rule generation for *ML* model is formulated in Section 3.2. At a high level, the first step of *BGMD* is to select a subset of features whose part of data are prone to be mispredicted based on imbalanced features, such as the *Updated commit count* feature in the merge conflict prediction data set (Section 3.3.1). Then, *BGMD* deduces a list of explanation rules to explain when a data contains what particular features that the model tends to mispredict. Note that, to the best of our knowledge, *BGMD* is the first method to use the feature imbalance for model *ME* rule generation.

Algorithm 1 presents the procedure of *BGMD* method. This procedure takes labeled data set D containing ground truth label, all attributes $\mathcal{A}$, an ML model M, and a target *ME* coverage δ (percentage of mispredicted data) as inputs. We now explain the procedure of BGMD.

Construct misprediction indication vector. The first step (line 1 in Algorithm 1) is to build a misprediction indication vector $\mathcal{I} : \mathcal{X} \rightarrow 0, 1$ for the model M, such that:

$$\mathcal{I}(x) = 1 \Leftrightarrow (D(x) \neq M(x))$$

In other words, the extracted indication vector $\mathcal{I}$ maps each input in D to a boolean value indicating whether the instance is mispredicted by the given model M.

Extract biased features. Next, our algorithm calls a procedure named *ExtractBiasFea-*

Algorithm 1 BGMD $(D, \mathcal{A}, M, \delta)$

Input: Labeled dataset $D : \mathcal{X} \rightarrow \mathcal{Y}$;
ML model $M : \mathcal{X} \rightarrow \mathcal{Y}$;
Data attributes: $\mathcal{A}$;
Target coverage: δ.
Output: Misprediction explaination for model M.

1: $\mathcal{I} \leftarrow I\{D, M(\mathcal{A})\}$
2: $\mathcal{BA} \leftarrow ExtractBiasFeatures(\mathcal{A}, \mathcal{I})$
3: $Atom \leftarrow GenAtoms(\mathcal{BA})$
4: $\Phi \leftarrow [\,]$
5: $cvg \leftarrow 0$
6: $cur \leftarrow D$
7: **while** $cvg \leq \delta$ **do**
8: $\quad \Phi \leftarrow LearnRule(Atom, cur)$
9: $\quad cur \leftarrow Filter(\Phi, cur)$
10: $\quad cvg \leftarrow ComputeCoverage(\mathcal{I}, cur)$
11: **end while**
return Misprediction Explanation Φ

tures (line 2 in Algorithm 1) to select a subset of features that the trained model is biased on, i.e., the model M performs significantly better on the feature's majority group than its minority group. The detail of the procedure *ExtractBiasFeatures* is in Algorithm. 2.

First, *ExtractBiasAttributes* separates all data into *mispredicted* and *correctly predicted* groups. Then, iterate each feature in $\mathcal{A}$ and evaluate whether there is a significant feature distribution difference (Mann-Whitney test, $\alpha < 0.05$) between the *mispredicted* and *correctly predicted* groups. We use the non-parametric Mann-Whitney test since the data population usually is not normally distributed. We consider the model is biased towards a feature if the Mann-Whitney test shows there is a significant difference ($\alpha < 0.05$) between *mispredicted* and *correctly predicted* instances.

Generate atomic predicates. Next, *BGMD* calls *GenAtoms* procedure (line 3 in Algorithm 1) to generate candidate atomic predicates of the form "*x op c*", where x is a feature and c is a constant value. If x is a categorical variable, we generate predicates of the form

Algorithm 2 ExtractBiasFeatures $(\mathcal{A}, \mathcal{I}, \alpha)$

Input: Data features: $\mathcal{A}$;
Mispredict indicator: $\mathcal{I}$;
Significance threshold: α.
Output: Biased feature list $\mathcal{BA}$.

1: $\mathcal{BA} \leftarrow [\,]$
2: mispredicted $\leftarrow \mathcal{I}(x) = 1$
3: correctly-predicted $\leftarrow \mathcal{I}(x) = 0$
4: **for** $feature$ in $\mathcal{A}$ **do**
5: MG $\leftarrow$ mispredicted[$feature$]
6: CG $\leftarrow$ correctly-predicted[$feature$]
7: P-value $\leftarrow$ Mann-Whitney(MG, CG)
8: **if** P-value $< \alpha$ **then**
9: $\mathcal{BA}.insert$(feature)
10: **end if**
11: **end for**
return Biased feature list $\mathcal{BA}$

Table 3.2: Example of universe atomic predicates based on the dataset in Table 3.1

Atomic Predicates		
commit num > 3	*add file num > 5*	*developer num > 4*
commit num <= 3	*add file num <= 5*	*developer num <= 4*
commit num > 18	*add file num > 15*	*developer num > 9*
commit num <= 18	*add file num <= 15*	*developer num <= 9*
commit num > 28	*add file num > 30*	*developer num > 15*
commit num <= 28	*add file num <= 30*	*developer num <= 15*

$x_c = c_j$ and $x_c \neq c_j$, where $c_j \in \mathcal{BA}$. For numerical features, we use operators $\leq, >$ and generate constant c_j using equal frequency binning [102]. For instance, if we have a numerical feature containing values $V = \{v_1, v_2, ..., v_n\}$, we first partition the (sorted) set V into k bins where each bin has roughly equal size. The value of k is a hyper-parameter and is set to 4 by default. Then, we use the highest value in each bin as one of the constants in our predicates to generate atoms of the form "x_n *op* c". Table 3.2 shows the universe of atomic predicates that are generated based on the features illustrated in Table 3.1.

Rule Learning. During rule learning (line 8 in Algorithm 1), we want to learn rules that

are correlated with mispredictions. This problem is equivalent to maximizing the following objective function:

$$precision = \frac{|x \in \mathcal{X} | \Phi(x) \wedge I(x) = 1|}{|x \in \mathcal{X} | \Phi(x)|} \tag{3.5}$$

which tries to make identified instances by Φ contain a higher percentage of mispredicted instances, and it corresponds to the precision value of *ME* rule.

However, if our rule learning algorithm solely aims to maximize precision, the *BGMD* may lead to a small rule size that takes many iterations to converge. Moreover, it may produce an over-fitted rule to a specific mispredicted instance (100% precision). Generating many rules to meet the coverage threshold would also result in producing a large number of sub-rules in Φ. This ultimately compromises the interpretability. Hence, instead of optimizing only on *precision*, our rule learning algorithm also takes *rule size* and *recall* into account. The recall is shown below:

$$recall = \frac{|x \in \mathcal{X} | \Phi(x) \wedge I(x) = 1|}{|x \in \mathcal{X} | I(x) = 1|} \tag{3.6}$$

which corresponds to the ratio between the identified mispredicted instances by generated rules and all mispredicted instances by the given model.

Thus, our final rule learning optimization objective function is a linear combination of *precision*, *recall*, and *rule size*:

$$Obj = \lambda_1 \cdot precision + \lambda_2 \cdot recall + \lambda_3 \cdot \frac{1}{size(\phi)} e \tag{3.7}$$

where parameters λ_1, λ_2, and λ_3 are tunable hyper-parameters and they are depends on the context and set to 1 by default. Precision is the primary factor that identifies mispredictions instances density, i.e., reducing the number of correctly predicted instances in identified instances. And recall controls the coverage of all mispredicted instances, i.e., increasing the number of identified mispredicted instances. Furthermore, rule size is mainly used for accelerating convergence and improving the explainability of generated rules.

Main learning loop. After the initialization phase (line 1 to 6), the algorithm enters a loop (line 7 to 11) that iteratively adds previously generated atomic predicate into a decision list until the learned rules achieve the desired coverage δ. The learned decision lists Φ is a list of predicates. For example, the list $\Phi = [\phi_1, \phi_2]$ corresponds to the following explanation:

$$if\ (\phi_1)\ then\ 1\ else\ if\ (\phi_2)\ then\ 1\ else\ 0$$

At a high level, the learning loop synthesizes the target decision list using a standard *sequential covering* method [36]. In particular, it first learns a rule ϕ_1 for the whole data set, then filters out instances satisfying ϕ_1, then learns another rule ϕ_2 for the remaining instance, and so on, until the target coverage is reached. Intuitively, the predicate in the i'th branch is the best predictor for the mispredictions in the subset of the data not covered by the earlier predicates. The algorithm terminates only when misprediction coverage cvg exceeds target coverage δ, thus the output of the $BGMD$ procedure is guaranteed to satisfy the coverage constraint.

3.3.3 Implementation

We implemented *BGMD* as a Python library that can be installed using *pip* command. It takes a Pandas dataframe, a target coverage, and a set of optional parameters and returns a set of decision lists paired with precision, recall, F1 score, and coverage metrics. An implementation is available in accompany website [6]

3.4 MAPS: Mispredicted Area uPweight Sampling

In this section, we present Mispredicted Area uPweight Sampling (*MAPS*), which leverages the *ME* information generated by *BGMD* to improve the *ML* model's performance on instances that contains under-represented features.

3.4.1 Overview of the baseline algorithms

In this study, we use a standard *ML* model training method and two sampling algorithms as baselines.

Empirical Risk Minimization (**ERM**) is a standard approach to train a *ML* model by minimizing the average training loss. *ERM* is trying to minimize the following loss function:

$$L_{ERM}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(x_i, y_i; \theta) \tag{3.8}$$

where θ is the parameter of the trained model.

Synthetic Minority Oversampling TEchnique (**SMOTE**) [41] is one of the most popular over-

sampling methods to improve the model's robustness by synthesizing instances in minority groups. The intuition of *SMOTE* is that it tries to balance the number of instances between majority group and minority group by synthesizing artificial instances in the minority group. So that the trained model can pay more attention to the instances in the minority group.

SMOTE is parameterized with K neighbors (the number of nearest neighbors it will consider) and the number N of new instances that it wishes to create. The way *SMOTE* synthesizes an instance is : (1) Randomly selects an instance in the minority group. (2) Randomly selects any of its K nearest neighbors belonging to the same class and generates a temporary new instance X_{temp} using the average of selected K neighbors. (3) Randomly specifies a value lambda in the range [0, 1]. (4) Generates and places a new instance on the vector between the original and X_{temp}, located lambda percent of the way from the original instance. In this work, we consider the *BGMD* identified data groups as the minority group since these groups are under-represented during model training.

Just Train Twice (**JTT**) [117] is a upweight sampling technique that was proposed in PMLR'21 [17], whose goal is to improve model's robustness via fixing model's performance on the mispredicted instances. We selected JTT as one of the baseline because JTT has been proven to be the state-of-the-art technique which has been compared with several upweight and reweight methods such as CVaR DRO [56], and Group DRO [155]. *JTT* has two-stages. In the first stage, it trains a *ML* model $\hat{M}$ on training data and then constructs a misprediction indication vector I on the validation data using equation 3.1, such that:

$$\mathcal{I}(x) = 1 \Leftrightarrow (D(x) \neq \hat{M}(x))$$

where D is the ground truth label for validation data.

Next, *JTT* retrains a final model M with validation data by upweighting all instances in the

validation data that were mispredicted by the first trained model:

$$L_{JTT}(\theta, I) = \left(\lambda_{up} \sum_{I(x_i)=1} \ell(x_i, y_i; \theta) + \sum_{I(x_i)=0} \ell(x_i, y_i; \theta) \right) \tag{3.9}$$

where $\lambda_{up} \in R_+$ is a tunable hyperparameter. The intuition of JTT is that for instances that the first model mispredicted, the final model should pay more attention to them. However, increasing the model's weight only for mispredicted data instances can make the model overfit to them. This could also result in the previously correctly predicted instances to be mispredicted by the final model. This problem was also found in our experiments and details are in Section 3.5.

3.4.2 MAPS: Mispredicted Area uPweight Sampling

MAPS is a novel upweight sampling method proposed in this paper based on the empirical observation that *ML* models tend to perform poorly on subsets of data containing under-represented features [62]. Therefore, *MAPS* first utilizes the *ME* rules generated by *BGMD* to identify a subset of the dataset that is prone to misprediction due to feature under-representation and then uses up-weight sampling to make the new model more aware of data with under-represented features. Unlike *JTT*, the retrained model using *MAPS* avoid focusing too much on a small subset of data that can lead to overfitting, and instead focus more on balancing saliency and under-represented features. The *MAPS* details presents in algorithm 3

Stage 1: Mispredicted Area identification. *MAPS* first trains a normal *ML* model $\hat{M}$. Then it identifies groups of instances that tend to be mispredicted by using misprediction

Algorithm 3 MAPS training

Input: Training set $\mathcal{D}$ and hyperparameter λ_{up}.
Stage one: Mispredict area identification
1. Train $\hat{M}$ on $\mathcal{D}$ via ERM (equation 3.8).
2. Extract the misprediction explaining rules Φ (equation 3.10).
Stage two: Upweighting points meet rules
3. Construct upweighted dataset $\mathcal{D}_{up}$ containing the training instances that meet the misprediction explain rules Φ.
4. Set λ_{up} times in loss function for $\mathcal{D}_{up}$ training instances and one for other examples (equation 3.11).
5. Train final model M_{final} using L_{MAPS} as the loss function.

diagnosing techniques, such as *BGMD* (Section 3.3).

$$\Phi = BGMD(x_i, y_i,\ \hat{M}) \tag{3.10}$$

Stage 2: Upweighting. After identifying the groups of instances that first model tends to mispredict, *MAPS* retrains a final model M_{final} by upweighting the identified instances during model training, using below loss function:

$$L_{MAPS}(\theta, \Phi) = \left(\lambda_{up} \sum_{x_i \in \Phi}^{n} \ell(x_i, y_i; \theta) + \sum_{x_i \notin \Phi} \ell(x_i, y_i; \theta) \right) \tag{3.11}$$

Implementation. The *MAPS* training method is described in Algorithm 3. To implement the upweighted objective (equation 3.11), we multiply a upweight value λ_{up} on identified subset of data. However, it's challenging to determine a universal upweight value λ_{up} for all models, so we tried various upweight values and used the best performed retrained model. Similar upweight value λ_{up} selection method was also used for *JTT*. In addition, we also

Table 3.3: Generated misprediction explanation rule coverage metrics by BGMD and EXPLAIN. DT represents decision tree, RF represents random forest, SVM for Support Vector Machine.

Software Engineering		EXPLAIN			BGMD (ours)			Kaggle		EXPLAIN			BGMD (ours)		
Task	**Model**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Task**	**Model**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**
Merge Conflict	DT	33.08	62.46	43.25	58.40	72.99	64.89	Spam	SVM	49.24	82.60	61.70	49.24	82.60	61.70
Pred. (Ruby)	RF	93.02	94.83	93.91	92.06	95.84	93.91	Email	DT	46.38	61.62	52.93	46.20	61.35	53.40
Merge Conflict	DT	67.64	71.45	69.49	64.56	78.76	70.96	Hotel	SVM	33.13	98.58	49.60	33.13	98.58	49.60
Pred. (Python)	RF	98.74	95.58	97.13	98.74	95.58	97.13	Booking	DT	18.96	99.92	31.87	18.96	99.92	31.87
Merge Conflict	DT	54.47	67.14	60.15	61.06	64.33	62.65	Bank	SVM	45.90	38.48	41.86	45.90	38.48	41.86
Pred. (Java)	RF	89.26	94.32	91.72	92.62	90.83	91.72	Marketing	DT	33.33	35.92	34.58	33.33	35.92	47.83
Merge Conflict	DT	65.25	63.17	64.19	60.25	68.68	64.19	Change	SVM	42.19	89.54	57.36	42.19	89.54	59.51
Pred. (PHP)	RF	85.33	92.96	88.98	83.33	95.45	88.98	Job	DT	20.28	44.21	59.51	20.28	44.21	59.51
Bug Report	DT	30.60	19.53	23.84	31.89	31.00	31.44	Water	SVM	55.51	47.87	49.74	48.95	83.93	62.55
Close Time Pred.	RF	7.51	38.14	12.55	41.97	40.69	41.32	Quality	DT	19.50	50.43	28.13	22.75	87.83	36.14
Average		**62.49**	**69.96**	**64.52**	**68.49**	**73.42**	**70.72**	Average		**36.44**	**68.66**	**47.10**	**36.99**	**70.50**	**51.92**

analyzed the impact of different upweight value λ_{up} on *MAPS* in Section 3.5.3.

3.5 Evaluation

In this section, we present empirical evaluation results that aim to answer the following research questions:

- **RQ1:** How does *BGMD* perform compared to the state-of-the-art *ME* rule generation method? (Section 3.5.1)
- **RQ2:** Can *MAPS* help improve the performance of *ML* models? (Section 3.5.2)
- **RQ3:** How do different upweight values affect the performance of the model when using *MAPS*? (Section 3.5.3)

To answer first research question, we compared *BGMD* with the state-of-the-art *ME* rule generation method *EXPLAIN* [46] on two SE tasks and five Kaggle [14] classification tasks. And to answer the second question, we compared *MAPS* method with a popular oversampling method (*SMOTE*) and a state-of-the-art upweight-sampling method (*JTT*) [117].

3.5.1 *ME* rule generation technique comparison

ME techniques need to ensure (1) high model *ME* coverage (quality) by the generated rules, and (2) less rule generation time (efficiency).

In terms of *ME coverage* metric, the subset data covered by generated *ME* rules should ensure that: (i) the majority of the covered data is mispredicted by the given model (*precision*), and (ii) the covered data should account for as many mispredicted data by given model as possible (*recall*). Thus, both *precision* (equation 3.5) and *recall* (equation 3.6) are important for a good *ME coverage* metrci. Therefore, when comparing *MAPS* with the state-of-the-art *ME* rule generation technique *EXPLAIN* [46], we use F1 score as it is a harmonically balanced value of precision and recall.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.12}$$

In terms of *ME* rule generation efficiency, we use rule generation time as the evaluation metric, i.e., the less time spent in the rule generation process, the more efficient the technique is.

Evaluation Subjects. For evaluation, we selected the two models that performed best in the study that proposed *EXPLAIN* [46], i.e., Decision Tree (DT) and Random Forest (RF). To replicate the study conducted by Cito et al. [46] and to evaluate whether our approach can be extended to Non-SE models, we also evaluated on two publicly available models from Kaggle [14]. We select the Kaggle models that still have room for improvement. Thus, we select Support Vector Machine (SVM) and Decision Tree (DT). Since our goal is to compare the effectiveness of various *ME* rule generation approaches, we did not conduct hyper-parameter tuning so that the models have room for improvement and there are mis-

Table 3.4: Representative rule from each technique

	Feature #	Rule	Prec.	Rec.	F1.	Time(s)
BGMD	**8**	**If** line_removed > 332 & developer_num > 45 & commit_num>372 & parallel_changed_file_num > 12 **elseif** line_removed>332 & developer_num > 88 & commit_num>229 & parallel_changed_file_num>12	0.92	0.95	0.94	**38.53**
EXPLAIN	**29**	**If** line_removed>332 & developer_num>88 & parallel_changed_file_num>12 **elseif** developer_num>45.0 & commit_num > 229 & parallel_changed_file_num>12	0.93	0.94	0.93	**203.48**

predicted data that can be identified by the *ME* rules. So we used default hyper-parameters in all models. In total, there are 57 hyper-parameters for the three models used in this paper. Due to space constraints, we list them in the companion website [11]. To remove the model variance, we used five-fold cross-validation (train-80%, test-20%) and repeated it five times with random seeds, and finally reported the median value, which is a common approach used by other studies [37].

Three SE data were evaluated in *EXPLAIN*, which are private to the Meta company and inaccessible to us. So, we use two publicly accessible SE datasets: merge conflict prediction [137] and bug report close time prediction [72]. For non-SE tasks, *EXPLAIN* was evaluated on two publicly available datasets from Kaggle [14]. However, the datasets they evaluated are too small to highlight the efficiency or scalability of different *ME* rule generation techniques. So we decided to use larger datasets (Spam Email, Water Quality, Bank Marketing, Change Job, and Hotel Booking) to compare our approach with *EXPLAIN*. Details of these datasets are in the companion website [10].

Results: Table 3.3 shows the results of applying *EXPLAIN* and *BGMD* on five SE data sets and five non-SE data sets collected from Kaggle. In SE related models, *BGMD* outperformed *EXPLAIN* on all three metrics. From Table 3.3, we can observe that *BGMD* improved the explanation rule result's *precision* from 62.49 to 68.49, *recall* from 69.96 to 73.42, and *F1*

score from 64.52 to 70.72 on average. In addition, we also got similar results on non-SE models. On average, the precision of rules generated by *BGMD* is 36.99, recall is 70.5, and F1 score is 51.92. However, the precision of *EXPLAIN* generated rules is 36.44, recall is 68.66, and F1 score is 47.1.

For merge conflict prediction models, the *BGMD*'s F1 score improves on three out of four *DT* learners. For example, On Ruby data, *BGMD*'s F1 score is 64.89, but *EXPLAIN*'s F1 score is 43.25. However, for RF-based merge conflict prediction models, the performance of *BGMD* and *EXPLAIN* are similar since *EXPLAIN* already achieved a very high F1 score (close to 0.9), leaving small room for improvement. For bug report close time prediction models, *EXPLAIN* got 23.84 F1 score with *DT* learner and 12.55 with *RF* learner. However, *BGMD* received 31.44 and 41.32, respectively. In summary, *BGMD* generated rules from a smaller number of biased features and performed better than *EXPLAIN* in terms of rule coverage. Similar results can be observed for the Kaggle dataset in Table 3.3.

In addition, one thing to note is that both *BGMD* and *EXPLAIN* take an input parameter that denotes target recall (i.e., percentage of mispredicted instances covered by generated explanation). Thus, the generated rules prioritize improving the recall values, which hurts the precision score. This is visible in table 3.3 where the average recall is 68.66 for *EXPLAIN* for non-SE tasks, but its precision is only 36.44. The same is visible for SE tasks. This holds true also for *BGMD*.

Figure 3.3 and 3.4 present the rule generation time comparison between *BGMD* and *EXPLAIN*. The average *ME* rule generation time by *BGMD* on SE models was 34 seconds. In contrast, the average rule generation time by *EXPLAIN* was 238 seconds. The results are significantly different (Mann-Whitney test, p-value<5.02e-14) [126], and the effect size is large (Cohen's D = 9.28) [50]. In terms of non-SE models, *BGMD* spent four seconds to generate *ME* rules, but *EXPLAIN* needed 60 seconds in average. The results are statistically significantly different for *BGMD* (Mann-Whitney test, p-value<5.02e-14) [126], and

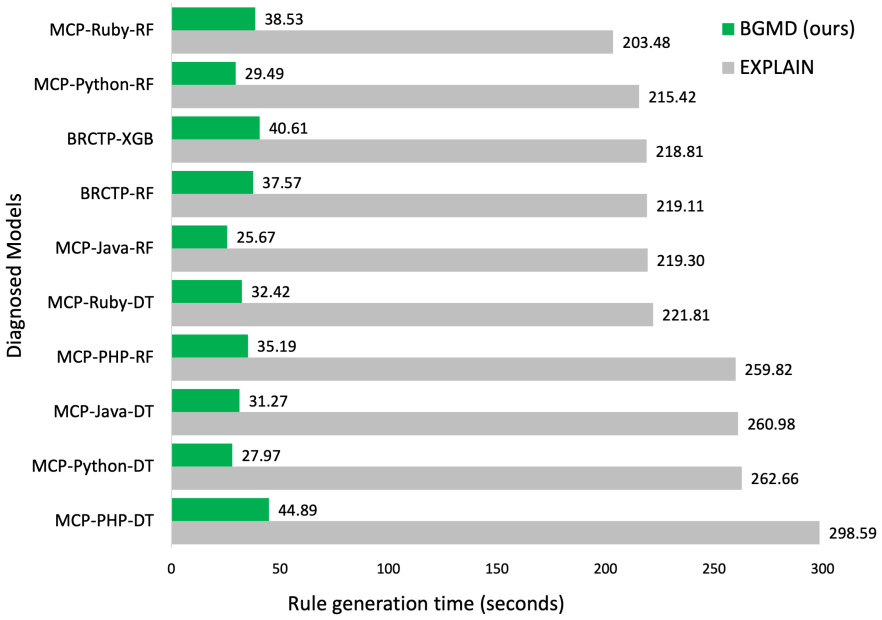


Figure 3.3: SE models ("MCP" represents Merge Conflict Prediction; "BRCTP" represents Bug Rreport Close Time Prediction).

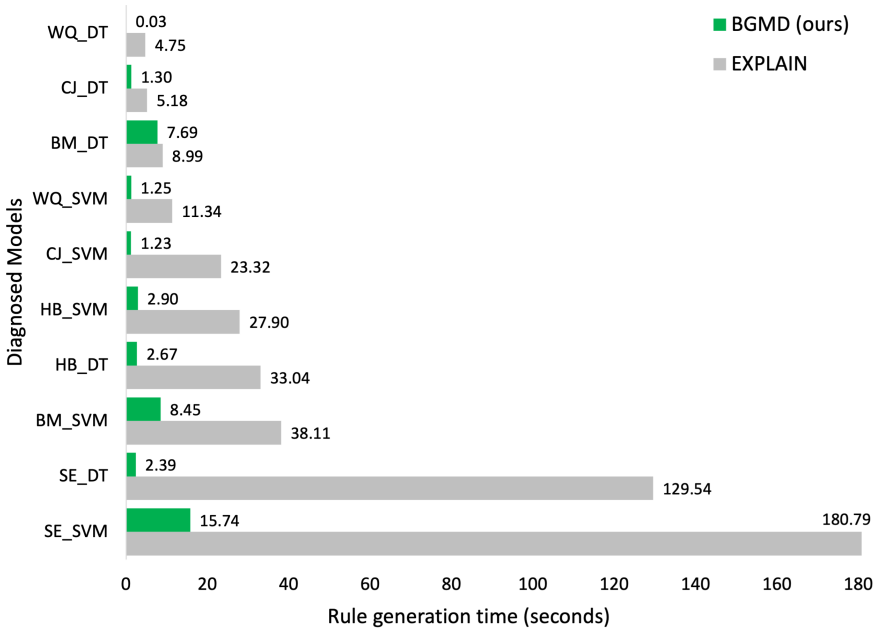


Figure 3.4: Non-SE models ("WQ" represents Warter Quality; "CJ" represents Change Job; "BM" represents Bank Market; "HB" represents Hotel Booking; and "SE" represents Spam Email).

the effect size is large (Cohen's D=9.28) [50].

Table 3.4 provides an illustrative overview of the representative *ME* rules for merge conflict prediction model produced by *BGMD* and *EXPLAIN*. *BGMD* spent 38.53 seconds to generate the rules, but *EXPLAIN* needed 203.48 seconds. This happened because *BGMD* induced rules from the identified eight biased features out of 29. In contrast, *EXPLAIN* tried to infer rules from all 29 features. Furthermore, after running a large number of models, we observed that the rules inferred by *EXPLAIN* contain the same features considered by *BGMD*.

3.5.2 Effectiveness of Mispredicted Area Upweight Sampling

In this section, we present the results to answer the research question: *Can MAPS fix the model's performance*? We present the evaluation results on five SE tasks that were used in RQ1 in table 3.5. Due to space constraints, we report the evaluation results for non-SE tasks in the companion website [6]. We used models trained with *ERM* as the baseline, which is named as "default" in table 3.5. In addition, we used oversampling method *SMOTE* and upweight sampling method *JTT* for comparison.

To compare *SMOTE*, *JTT*, and *MAPS* performance, it is important that these methods should not only improve the model's performance on mispredicted data, but also ensure model's performance on all data. Thus, in table 3.5, we present each method's performance on both *Mispredicted data* and *All data*. To remove the model variance, we used 5 fold cross-validation (train-80%, test-20%) and repeated 10 times with random seeds and finally reported the median, which is a common approach used by other studies [37].

Table 3.6 summarizes the win times of each method on various metrics in table 3.5. If more than one techniques get the highest value on a metric, we consider they all win for that metric.

Table 3.5: "Default" denotes off-the-shelf model; "SMOTE" is trained with SMOTE [41]; "JTT" is trained with JTT [117]; "MAPS" is trained with this paper proposed algorithm. The darker the color, the higher the value.

Task	Model	Algo	Mispredicted Data			All data		
			Pre.	Rec.	F1.	Pre.	Rec.	F1
Merge Conflict Predic. (Ruby)	DT	Default	0.54	0.57	0.55	0.63	0.68	0.64
		SMOTE	0.51	0.52	0.52	0.7	0.68	0.69
		JTT	0.53	0.55	0.55	0.72	0.7	0.71
		MAPS	0.56	0.58	0.57	0.78	0.79	0.79
	RF	Default	0.71	0.49	0.58	0.67	0.81	0.71
		SMOTE	0.62	0.69	0.65	0.69	0.9	0.78
		JTT	0.72	0.6	0.59	0.7	0.86	0.77
		MAPS	0.72	0.52	0.6	0.7	0.83	0.76
Merge Conflict Predic. (Java)	DT	Default	0.56	0.59	0.57	0.7	0.74	0.72
		SMOTE	0.56	0.61	0.58	0.7	0.75	0.73
		JTT	0.58	0.58	0.58	0.7	0.72	0.7
		MAPS	0.58	0.62	0.6	0.75	0.76	0.75
	RF	Default	0.75	0.57	0.65	0.82	0.81	0.81
		SMOTE	0.64	0.73	0.68	0.78	0.81	0.79
		JTT	0.75	0.56	0.64	0.82	0.84	0.83
		MAPS	0.76	0.58	0.66	0.83	0.86	0.84
Merge Conflict Predic. (Python)	DT	Default	0.44	0.46	0.44	0.6	0.59	0.59
		SMOTE	0.42	0.47	0.46	0.55	0.55	0.55
		JTT	0.45	0.46	0.45	0.57	0.61	0.61
		MAPS	0.46	0.48	0.48	0.64	0.62	0.63
	RF	Default	0.69	0.37	0.48	0.74	0.59	0.66
		SMOTE	0.54	0.55	0.55	0.63	0.73	0.69
		JTT	0.68	0.36	0.47	0.75	0.59	0.67
		MAPS	0.69	0.38	0.49	0.78	0.61	0.72
Merge Conflict Predic (PHP)	DT	Default	0.52	0.54	0.53	0.73	0.82	0.77
		SMOTE	0.5	0.56	0.53	0.72	0.76	0.74
		JTT	0.54	0.55	0.54	0.72	0.71	0.71
		MAPS	0.53	0.56	0.55	0.76	0.86	0.81
	RF	Default	0.7	0.5	0.58	0.7	0.88	0.75
		SMOTE	0.59	0.69	0.64	0.71	0.95	0.84
		JTT	0.71	0.51	0.59	0.65	0.86	0.74
		MAPS	0.72	0.52	0.61	0.73	0.91	0.82
Bug Report Close Time Predic.	RF	Default	0.69	0.66	0.68	0.71	0.67	0.69
		SMOTE	0.65	0.72	0.69	0.7	0.73	0.72
		JTT	0.69	0.71	0.7	0.74	0.73	0.73
		MAPS	0.71	0.7	0.71	0.75	0.71	0.73
	XGB	Default	0.8	0.63	0.69	0.8	0.76	0.78
		SMOTE	0.78	0.7	0.73	0.76	0.84	0.81
		JTT	0.82	0.64	0.72	0.81	0.76	0.78
		MAPS	0.84	0.66	0.74	0.81	0.78	0.8

Table 3.6: Summarized information of comparing MAPS with SMOTE [41], JTT [117] based on the result in table 3.5

	Mispredicted data			All data		
	Won on Prec.	Won on Rec.	Won on F1	Won on Prec.	Won on Rec.	Won on F1
SMOTE	0	6	4	0	6	3
JTT	2	0	0	2	1	1
MAPS	9	5	6	10	4	6

For example, MAPS received highest precision on all ten models, and JTT on two. However, in terms on Recall, SMOTE received best performance on six, but MAPS got the highest recall on four models. Since *SMOTE* balances the data via synthesizing instances in minority groups, the trained model is biased towards the data that has been "duplicated" many times, i.e., the previous minority data. Moreover, the new trained model performs worse on the majority group that performed well before the retraining. This results in *SMOTE* gaining in recall but lowering the precision. Similar affect has been observed in prior studies involving *SMOTE* [41, 116, 134]. While SMOTE outperforms MAPS in terms of recall, MAPS has better combined results in terms of the overall performance measured using F1. Not only does it improve the model's performance on mispredicted data, it doesn't corrupt data that was previously correctly predicted. Based on the evaluation results shown in table 3.6, *MAPS* won more times compared to *SMOTE* (Mann-Whitney test, p-value<3.8e-2) [126] and *JTT* (Mann-Whitney test, p-value < 2.5e-4) [126].

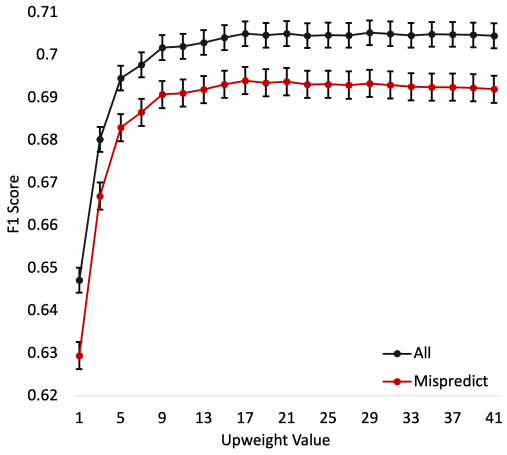
3.5.3 Impact of Upweight Value on *MAPS*

MAPS algorithm contains an important hyper-parameter: *upweight value* (λ_{up}) in equation 3.11, which is a number multiplied by the *ME* rule identified instances. The higher the *upweight value*, the retrained model pays more attention to the identified instances. However, the best *upweight value* has to be empirically determined. Thus, we investigated the impact of weight hyper-parameter on *MAPS* algorithm.

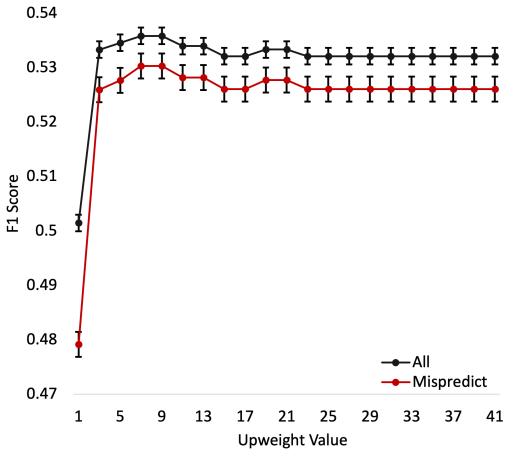
Figure 3.5 shows four representative F1-score change patterns when increasing the *upweight value* in *MAPS*. Note that when the *upweight value* is equal to one, all data have the same weight during model training. So for each figure in Figure 3.5, the left most pair of dots is the result for the default model without using *MAPS*. When the *upweight value* is equal to five, the instances identified by *ME* rule have fives times weight than others during model

training. The higher the *upweight value*, the trained *ML* model pays more attention to the instances that identified by *ME* rule generation tools. In each subfigure, we present the model's F1 score changes when using *MAPS* for *all data* and *mispredicted* data to show the various upweight value's impact when using *MAPS*.

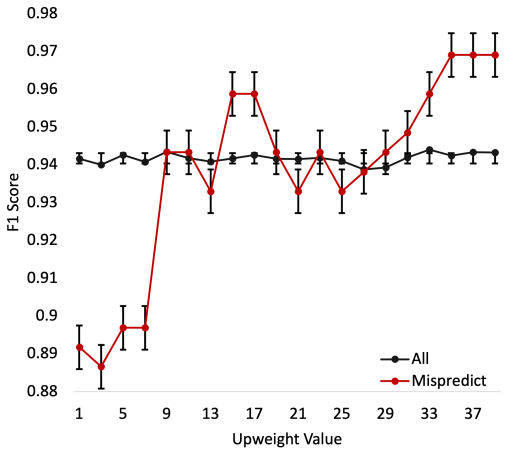
Figure 3.5(a) is the F1 score changes for all data and mispredicted data in the Merge Conflict Prediction (Ruby) dataset. The chart shows that F1 score grow gradually to a plateau. Figure 3.5(b) is the F1 score changes for the Hotel Booking dataset, and both F1 scores come to a plateau faster than Figure 3.5(a). These two patterns are the most common patterns when increasing *upweight value* in *MAPS* algorithm. In addition, we also observe other F1 score change patterns. Such as Figure 3.5(c), as the *upweight value* increases, the mispredicted data's F1 score increases, but all data's F1 score only changes a little. Figure 3.5(d) shows another interesting pattern. When the weight multiply is one, the F1 score of all data is higher than mispredicted data. But their performance drops drastically when the *upweight value* is equal to three, and all data's performance is even worse than mispredicted data. Then, as the weight increases, the two F1 scores begin to grow together and exceed the initial value until they enter a platform together. Figure 3.5 show the most common F1 score change patterns. Although their patterns are different, main trends are similar to the F1 scores increase and plateau after a particular point.



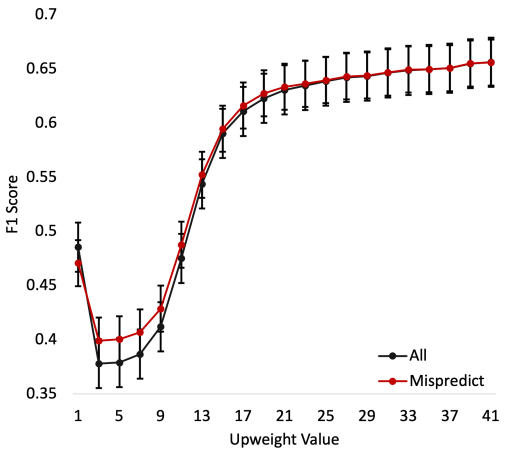
(a) MCP. (Ruby)



(b) Hotel Booking



(c) BRCTP



(d) Spam Email

Figure 3.5: F1 score change patterns when increasing weight times value in *MAPS*

3.6 Discussion

In this section, we first discuss why our proposed *BGMD* could perform better than the state-of-the-art model *ME* and the benefits of using *MAPS* to improve the model's robustness.

3.6.1 Why *BGMD* works better?

Focus only on useful features. *BGMD* deduces the *ME* rules only on biased features instead of "blindly" trying on all features. For example, in table 3.4, *BGMD* generated rules from eight biased features. In contrast, *EXPLAIN* generated rules from 29 features. Although *EXPLAIN* tried to explain misprediction using 29 features, their generated explanation rules were based on the same eight features that *BGMD* focused on. A similar situation happened on all evaluated models that *EXPLAIN* tried to explain mispredictions using all available features, but the constituent features for its generated rules were all considered by *BGMD*.

Make more attempts. Cito et al. [46] showed that with more granular predicates on features, the generated *ME* rules can get better results on misprediction coverage but need more computation time. Thus, given the same computation time, *BGMD* is able to deduce better *ME* rules. *BGMD* ignores many features that are not helpful to explain the mispredictions. For example, in Spam Email dataset, it contains 232 features, and *BGMD* only focuses on 23 features associated with the corresponding model's mispredictions. Thus, shown in Figure 3.4, *EXPLAIN* spent 180.79 seconds to generate *ME* rules. In contrast, *BGMD* only took 15.74 seconds.

3.6.2 Why *MAPS* is a good method to fix models?

Competitive performance. According to our empirical evaluation results in table 3.6, *MAPS* outperformed the popular oversampling method *SMOTE* [41] and state-of-the-art method *JTT* [117] on ten models in terms of precision and F1 score on mispredicted data. Thus, *MAPS* is a competitive method to improve the model's robustness.

Uncompromising performance for all data: Table 3.5 shows *MAPS* not only improved the model's performance on mispredicted data but also on all data. In contrast, although *JTT* used a similar upweight sampling approach as *MAPS*, it reduced the performance of four models on all data. We attribute our success in this regard to making retrained model pay more attention to the under-represented features instead of focusing more on particular mispredicted instances. Thus, *MAPS* can improve the model's misprediction performance without compromising all data prediction performance.

No extra computation: Table 3.6 shows that SMOTE also did well in helping the model fix performance on mispredicted data, especially on improving *recall*. Furthermore, if possible, adding more manually annotated data in minority groups might improve model performance even more than SMOTE. However, adding more data means more computation during model training. One benefit of *MAPS* is that it does not require extra annotated or synthesized data, which does not add computation overhead during model training. Note that *MAPS* is not an alternative to SMOTE, but a complement. Because table 3.6 shows that *SMOTE* performed best on improving the model's recall on both mispredicted and all data.

Model agnostic: *MAPS* entirely focuses on identified data groups that are prone to be mispredicted to fix the model's performance on them. There are works where optimization algorithms have been used to modify models [46]. These works are model specific and, most of the time, combined with internal model logic. Thus *MAPS* is much more general as it can be used for any kind of model.

3.7 Related Work

The study in the paper relates to several topics below.

Debugging ML Models. The goal of debugging a model is to identify the specific groups of data on which *ML* model is likely to fail and then fix the model's performance on identified data. Tongshuang et al. [188] presented an error analysis for NLP models called *Errudite*. However, *Errudite* requires users to tune the parameters in order to perform error analyses. On the contrary, our proposed method automatically identifies the groups of data that are prone to be mispredicted and uses a simple and effective method to fix the model's performance on them. Kim et al [96] is close to our work. However, their approach is tailored to Computer Vision, and adopting their approach for text/source code is not trival due to the difference between CV and text/source code. They can create permutations of features (i.e., weather, car model, etc.). If the object of interest remains intact, they can create new images without changing the meaning. In our case, the meaning is changed if the context is changed.

Select data for upweight sampling. Increasing part of data instance's weight during model training has been proved as an efficient approach to improve model's performance [35, 68, 51, 117]. For instance, Karan et al. [68] isolates features that differentiate subgroups within a class and then augment the minority groups. Jonathon et al. [35] tweaks L2 regularization to produce the correct weighting effect on minority groups. Fereshte et al. [94] improves fairness and robustness by halving the loss across all the groups. Another group of studies identifies the groups based on fairness [73, 186, 22, 94]. In addition, *JTT* identifies mispredicted instances from a validation set through a trained model and then retrains a model via upweighting only on mispredicted instances. In other words, *JTT* is trying to make the retrained model focus more on mispredicted instances in the first model. In contrast, our approach identifies the groups of data that tend to be mispredicted because

of some under-represented features during model training. Then, we increase these identified data weights during model training and make retrained models pay more attention to those under-represented features.

3.8 Threats to Validity

We have taken care to ensure that our results are unbiased and tried to eliminate the effects of random noise, but it's possible that mitigation strategies may not have been effective.

Bias due to dataset: Our findings may not generalize to all software projects since we evaluate using 10 datasets. However, all these datasets are publicly available and have been used in previous studies. Moreover, r considered projects are large and significantly different in size, programming languages, complexity. So we believe that the selected projects adequately address the concern.

Bias due to models: This work is based on binary classification and tabular data, which are very common in *ML* software. We select the models that have been used by the papers that introduced the dataset. In the future, we will test how our method performs in complex neural network models.

3.9 Conclusion

We propose an efficient model-agnostic technique for generating useful and interpretable misprediction explanations for machine learning models. We demonstrate through case studies that our proposed bias-guided misprediction explanation technique is significantly more efficient than the state-of-the-art technique and generates explanation rules that have higher misprediction explanation capability. In addition, we introduce a mispredicted area upweight

sampling algorithm to improve the model's robustness via fixing the model's performance on incorrectly predicted instances containing under-represented features. Our results show that our proposed method outperforms the state-of-the-art techniques. We plan to conduct studies on a broader range of tasks and datasets in the future.

Chapter 4

Attention Bias in Transformer-based Models for Software Engineering

4.1 Introduction

Pre-trained Language Models (PLMs) such as BERT [54], GPT [145], and T5 [146] have exhibited notable performance gains in various Natural Language Processing (NLP) tasks [49, 107, 192]. This trend has been further extended to software engineering applications, including but not limited to code summarization [25, 122, 24], code translation [57, 142, 179], and code search [59, 71, 81]. These models are built on the Transformer network architecture [168], featuring a self-attention mechanism that learns the weight and interdependence of attention among tokens within an input sequence.

The self-attention mechanism uses attention weight to capture inter-relationships and long-range dependencies among tokens in a sequence. Prior research has investigated how attention weights are distributed across different hidden layers of the PLM [170, 59, 71] and different code syntax [197, 177]. In the most recent work, Zhang et al. [197] identified that

CodeBERT [59] pays more attention to certain types of tokens and statements such as keywords and data-related statements.

To date, research on attention weight has concentrated on PLMs. In current practice, the prevalent approach is to utilize the pre-training and then fine-tuning paradigm, wherein PLMs are fine-tuned to attain optimal performance in downstream tasks. Despite this widespread usage, the alteration of attention weight distribution during fine-tuning and its distinction between correct and incorrect prediction groups remains poorly understood. Therefore, understanding this could lead to the exploration of innovative techniques to leverage this information and enhance the overall performance of fined-tuned PLMs.

This paper details our analysis of attention-weight assignments in the fine-tuned CodeBERT. Our research indicates that fine-tuned CodeBERT exhibits a discernible preference towards certain syntax tokens, including identifiers, modifiers, and Abstract Syntax Tree (AST) elements, such as method signatures when making correct predictions. Based on this observation, we propose an attention-guiding mechanism that promotes the allocation of more weight by attention heads towards these crucial syntax tokens and AST elements to improve the performance of the fine-tuned model.

A similar approach is presented by Deshpande et al. [53] for natural language text, where a limited number of generic pre-defined attention patterns are proposed to guide the self-attention heads towards paying more attention to global and local positions, such as the first, last, previous, and next tokens. However, due to the inherent differences between programming languages and natural languages, their proposed attention guiding patterns may not be suitable for software engineering tasks. Therefore, we introduce a set of specific syntax attention guiding patterns for programming language PLMs that encourage fine-tuned models to pay greater attention to critical source code syntax tokens and AST elements.

Figure 4.1 illustrates the attention-guiding mechanism we developed for fine-tuning source

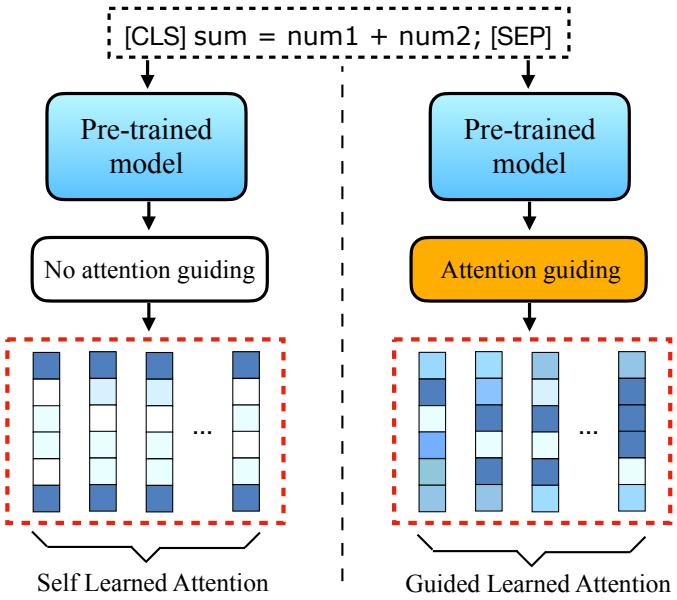


Figure 4.1: Illustration of attention guiding mechanism

code PLMs. Specifically, the left-hand side of Figure 4.1 shows the input source code being fed into a pre-trained model for fine-tuning without utilizing any attention-guiding techniques. The resulting attention weight vectors are entirely self-learned, and darker colors indicate higher attention weight assignments. Based on the findings of Sharma et al. [158], self-learned attention heads tend to assign a substantial proportion of attention weight to delimiters added by the tokenizer, such as [CLS] and [SEP], which represent the first and last positions in the learned attention weight vectors. Conversely, the right-hand side of Figure 4.1 shows the same source code input, but with the attention-guiding mechanism in place. This mechanism aims to encourage the self-attention heads to assign higher attention weights to pre-defined critical tokens.

In this study, we aim to investigate following research questions:

RQ1: How does the attention weight assignment of fine-tuned CodeBERT vary in relation to program syntax tokens between correct and incorrect predictions?

RQ2: How the attention weight assignment of fine-tuned CodeBERT vary in

relation to AST elements between correct and incorrect predictions?

RQ3: What is the efficacy of the proposed syntax pattern attention guiding mechanism on software engineering tasks?

RQ4: Among the syntax patterns considered in this study, which pattern yields the most notable impact on the downstream task performance?

Our study makes several significant contributions to the field of PLMs for software engineering:

- Firstly, we provide the first empirical evidence of attention weight bias towards source code syntax tokens and AST elements in fine-tuned language models.
- Secondly, we propose a novel attention-guiding technique, SyntaGuid, which enables PLMs to focus attention weight on critical source code syntax tokens and AST elements.
- Thirdly, we demonstrate the effectiveness of the proposed attention guiding mechanism across multiple software engineering datasets and tasks, establishing its potential as a generalizable solution for improving fine-tuned PLMs performance.

The remainder of the paper is structured as follows. Section 2 describes the necessary background. Section 3 presents details of the empirical analysis. Section 4 presents our proposed approach SyntaGuid. Section 5 places results in the broader context of work to date and Section 6 outlines the implications for practitioners and researchers. Section 7 presents the related works. Section 8 lists the threats to validate our results. Section 9 concludes with a summary of the key findings and an outlook on our future work.

4.2 Background

In this section, we explain the necessary backgrounds.

4.2.1 Pre-training Language Model

Given a corpus $\mathcal{C}$, each sentence (or code snippet) is first tokenized into a series of tokens. Prior to pre-training, the model takes the concatenation of two segments as the input, defined as $c_1 = \{t_1, t_2, ..., t_n\}$ and $c_2 = \{w_1, w_2, ..., w_m\}$, where n and m denote the lengths of the two segments, respectively. The two segments are concatenated with a special separator token [SEP]. Furthermore, the first and last tokens of concatenated sequence are padded with a special classification token [CLS] and an ending token [EOS], respectively. Formally, the input of each training sample can be represented as follows:

$$s = [CLS], t_1, t_2, ..., t_n, [SEP], w_1, w_2, ..., w_m[EOS].$$

The Transformer encoder is then used for pre-training with two self-supervised learning objectives: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). In MLM, a certain percentage of the tokens in an input sentence is randomly selected and replaced with the special token [MASK]. Specifically, BERT chooses 15% of the input tokens for possible replacement, and among them, 80% are replaced with [MASK], 10% remain unchanged, and the remaining 10% are randomly replaced with tokens from the vocabulary. The purpose of MLM is to train the model to predict the masked tokens based on the surrounding context. For NSP, it is modeled as a binary classification task to predict whether two segments are consecutive. Positive and negative training examples are generated based on the following rules: (1) if two segments are consecutive in a document, they are considered

positive examples; (2) otherwise, paired segments from different documents are considered negative examples.

4.2.2 CodeBERT

Recently, self-supervised learning techniques using MLM have gained popularity for natural language understanding and generation [136, 181, 118]. Similarly, in the field of software engineering, several pre-trained code models have been proposed for program comprehension, code generation and etc [9, 8, 2]. In this study, we have chosen to use the CodeBERT [59] pre-trained model since this is one of the state-of-the-art code models for code representation learning.

CodeBERT pre-trains the model on two tasks: MLM and Replaced Token Detection (RTD). In MLM, two random tokens from the input pair of code and natural language comments are masked, and the model aims to predict the original token from a large vocabulary. The RTD task involves two generators and a discriminator. The generators predict the original token for the masked token, while the discriminator predicts whether the tokens are original or not. After pre-training, CodeBERT can be fine-tuned on downstream tasks, making it a versatile model for a wide range of software engineering applications, such as defect detection [122], clone detection [7, 122], code generation [8, 9, 136] etc.

4.3 Empirical Analysis for Attention Weights

In this section, we present our research methodology, experimental setup, and attention bias analysis results of fine-tuned CodeBERT. The main objective of this empirical investigation is to ascertain the extent to which CodeBERT exhibits disparate attention weights on distinct positions in the source code syntax during both successful and unsuccessful predictions.

4.3.1 Study Design

Multi-head attention is a fundamental mechanism in Transformer-based models for language modeling, which enables the quantification of token importance in a given sentence. This attention distribution facilitates the learning and representation of a sentence by assigning higher weights to tokens that carry greater significance. Extracting useful information from PLMs requires an understanding of the important tokens within the code. As source code can be analyzed at different levels of granularity, including tokens, statements, and AST elements, this study focuses on the atomic unit of source code, i.e., syntax tokens and AST elements. Examining attention weights at this level of granularity can provide insights into CodeBERT's performance with respect to the assigned attention on larger code blocks. To this end, we seek to answer the following research questions through our analysis:

- **RQ1: How does the attention weight assignment of fine-tuned CodeBERT vary in relation to program syntax tokens between correct and incorrect predictions?**
- **RQ2: How the attention weight assignment of fine-tuned CodeBERT vary in relation to AST elements between correct and incorrect predictions?**

To address our research questions, we utilize the attention weights from the Transformer layers of CodeBERT after fine-tuning to measure the importance of each token. We subsequently explore whether the collected attention weights exhibit significant differences between correct and incorrect predictions.

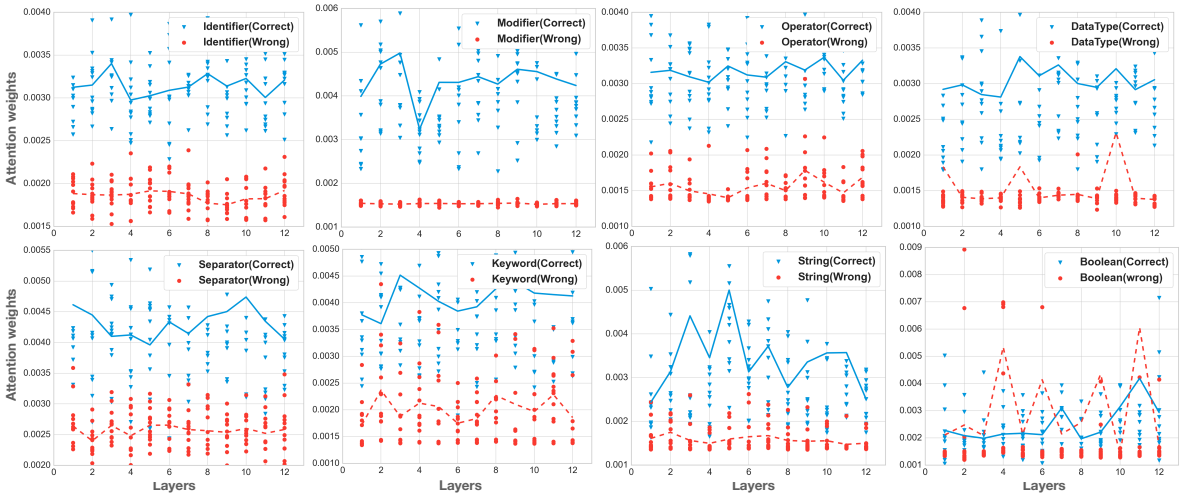


Figure 4.2: Empirical results for syntax token assigned attention weights comparison between correctly predicted and mis-predicted groups for cloze test.

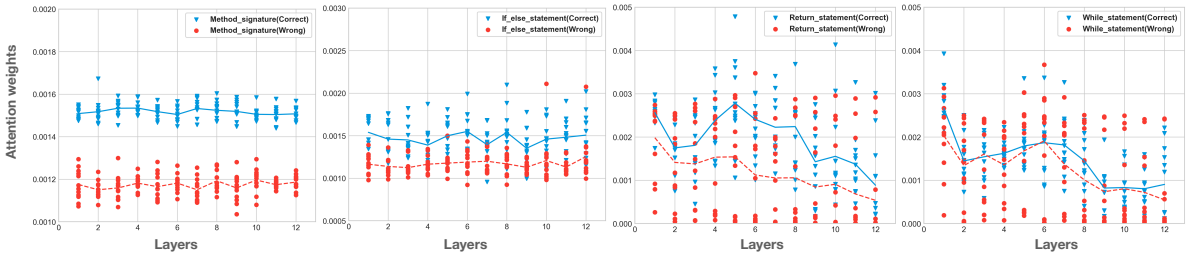


Figure 4.3: Empirical results for abstract syntax tree elements assigned attention weights comparison between correctly predicted and mis-predicted groups for cloze test.

4.3.2 Measuring attention weights

Our target PLM CodeBERT consists of 12 self-attention layers, each containing 12 heads that compute attention weights for the same token. To yield a comprehensive estimate of the attention weight for each token, we adopt an approach that aggregates the attention scores across all layers and heads. This approach is consistent with prior studies in the field [158, 170, 123, 92].

4.3.3 Experiment tasks

We chose to evaluate two code understanding tasks, namely code clone detection and cloze test, as well as one code generation task, named code translation. Below is the comprehensive description of each task and its associated dataset used in our study.

Task 1: Code clone detection. Clone detection is a crucial task in software engineering, aimed at identifying instances of duplicate code within a software system. Such code duplication can arise from ad-hoc code reuse practices, such as copy-pasting, and can result in numerous negative effects on the software development process [156]. To address this issue, binary classification algorithms are used to classify code pairs as either equivalent or not. For the purposes of our study, we employ the BigCloneBench dataset, which is a large-scale benchmarking dataset for clone detection tasks [161]. This dataset contains 6 million true clone pairs and 260 thousand false clone pairs, and covers ten different functionalities or "cases". The benchmark can be found at the GitHub repository [7].

Task 2: Cloze test. Cloze tests involve predicting the correct answer for a masked word or phrase given the context. This method has been extended to the source code domain in the CodeXGlue [122] dataset, with two cloze testing datasets, namely ClozeTest-maxmin and ClozeTest-all, each comprising instances of masked code functions, their corresponding docstrings, and the target words to be predicted. We focus our study on the more complex ClozeTest-all dataset, which includes a larger set of 930 target words as compared to only two in ClozeTest-maxmin. The CodeXGlue dataset has been introduced by Microsoft Research and encompasses six different programming languages. The dataset can be accessed in the CodeXGLUE repository [122].

Task3: Code translation. Code translation aims to migrate legacy software from one programming language and platform to another [83]. To this end, we rely on the CodeXGLUE dataset, which has been widely adopted in recent research efforts [122]. Specifically, we uti-

Table 4.1: Details of datasets of evaluate tasks

Task	**Dataset**	**Size**	**Language**
Cloze test	CodeXGlue	50k	Java
Code Clone Detection	BigCloneBench	901k	Java
Code Translation	CodeTrans	11.5k	Java-C#

lize the Code2Code translation dataset, which involves the translation of Java code to its equivalent in the C# programming language. This dataset has been curated from publicly available repositories such as Lucene [3], POI [4], and Antlr [18], with a focus on identifying parallel functions between the two languages. Furthermore, to ensure the quality of the dataset, duplicates and functions with empty bodies have been removed in Code2Code.

Table 4.1 summarizes the details of the datasets used in the evaluation for the three above mentioned tasks.

4.3.4 Selected syntax types and AST structures

Drawing upon the prior work by Aljehane et al. [26], which investigated the variance in attention behavior between expert and novice programmers when perusing source code, our study scrutinizes syntax tokens and AST that bear relevance to software development. Specifically, we assess the salience of various syntax tokens, namely *identifiers, modifiers, operators, data types, separators, keywords, strings, and Booleans*. To extract these syntax tokens from the input source code, we utilize Javalang [13], a well-known Java syntax collection library. Furthermore, we explore the elements of AST, such as *method signature, if-else elements, while elements, and return elements*, in determining the attention behavior of fine-tuned CodeBERT. To identify these AST elements, we employ a commonly used Java AST structure identification library, tree-sitter-java [12].

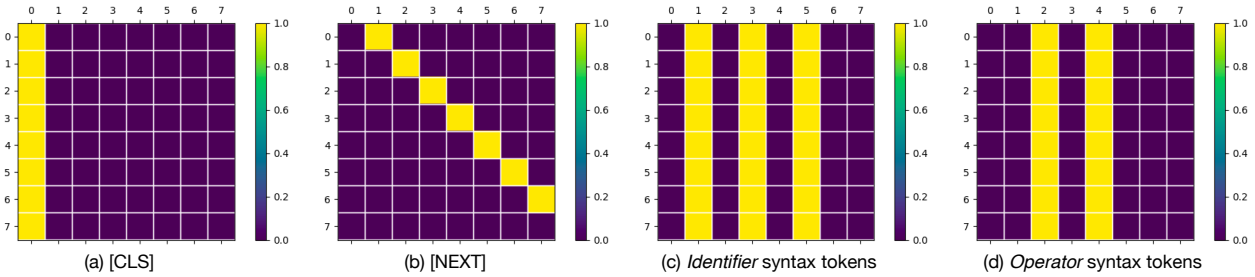


Figure 4.4: Example attention guiding patterns for the example code snippet "<s> sum = num1 + num2; <\s>", whose syntax type list is: [[CLS], identifier, operator, identifier, operator, identifier, separator, [SEP]]. Note that the first two patterns are proposed in [53] for natural language, and the last two syntax token patterns are proposed in this study for programming language.

4.3.5 Attention weight analysis

We start by fine-tuning three distinct downstream models for the aforementioned tasks with their respective original training datasets and hyper-parameters used in the CodeXGLUE [122] benchmark. All hyper-parameter details are in our study's companion website [5]. Subsequently, we partitioned the prediction data into two groups based on the model's performance: correct and incorrect predictions. We then analyze the attention assigned to each syntax token and AST element and compare the attention weights between the two aforementioned groups. Since we perform multiple tests, we account for multiple hypothesis corrections and adopt the Bonferroni correction [33], resulting in an adjusted p-value of 0.01. To determine whether significant differences exist between the correctly and incorrectly predicted groups for all syntax tokens and AST elements, we utilize the non-parametric Mann-Whitney test [126], considering that our population is not normally distributed.

4.3.6 Attention bias analysis results

Figure 4.2 and Figure 4.3 visualize the attention weights assigned by 12 attention heads in 12 layers to different syntax tokens and AST elements for cloze test. Due to space constraints,

we provide the visualization of the attention weights for the other tasks in the companion website [5]. The triangle ▼ symbol represents the average attention values assigned by self-attention heads to the syntax tokens of correctly predicted instances, and the solid line represents the average attention values of all 12 heads in 12 layers. On the other hand, the circle ● symbol represents the average attention values assigned to the syntax tokens of mispredicted instances, and the dashed line represents the average attention values of all 12 heads in 12 layers.

From Figure 4.2, we can infer that the self-attention heads exhibit a bias toward assigning higher attention weights to certain syntax tokens, such as Identifier, Modifier, Operators, Basic Datatype, Separator, Keywords, and String, when the fine-tuned model successfully makes predictions on the cloze test. However, we do not observe any significant difference in attention weights assigned to Boolean syntax tokens. To evaluate the statistical significance of these differences, we performed a paired t-test [126] on the attention weights assigned to each syntax token by self-attention heads between the correctly and incorrectly predicted groups. Our analysis revealed that self-attention heads assign attention weights that are significantly different on identifier tokens (p-value ¡ 2.13e-20), modifier tokens (p-value ¡ 1.12e-16), operator tokens (p-value ¡ 4.82e-21), basic data type tokens (p-value ¡ 4.16e-13), separator tokens (p-value ¡ 3.37e-17), keyword tokens (p-value ¡ 5.06e-16) and string tokens (p-value ¡ 1.73e-08), but not on boolean tokens (p-value ¡ 0.31). We see similar results for the other two tasks (Code Clone detection and Code translation).

In terms of AST elements, Figure 4.3 illustrates that self-attention heads exhibit a greater attention weight on specific AST elements, such as method signatures, if else elements, and return elements, when the CodeBERT model is able to successfully make predictions. Statistical analysis was conducted to investigate the difference in attention assigned by the self-attention heads to these AST elements between the correctly and incorrectly predicted instances. The results indicate that there is a significant difference in attention assigned

to method signatures (p-value ¡ 4.70e-26), if else elements (p-value ¡ 2.43e-12), and return elements (p-value ¡ 0.92e-3). However, no significant difference in attention assigned to while elements were observed (p-value ¡ 0.36). Similar findings were also obtained for the code clone detection and code translation tasks, and the detailed results can be found in the companion website [5].

CodeBERT's self-attention heads assign significantly greater attention weights to method signatures, if else elements, and return elements AST elements.

4.4 SyntaGuid: Syntax Pattern Attention Guiding

In this section, we present our novel approach for fine-tuning Transformer-based models on source code by utilizing attention guiding. Specifically, we begin by formally defining the MLM set up within the context of Transformers [168] and proceed to describe the attention guiding technique. Subsequently, we introduce our proposed Syntax Pattern Attention Guiding (SyntaGuid) technique, which leverages the syntactic structure of source code to guide the attention mechanism during fine-tuning.

4.4.1 Masked Language Modeling (MLM)

The application of Transformers in sequence-to-sequence prediction tasks involves training on a dataset $\mathcal{D}$ comprising pairs of sequences x and their corresponding labels y. In the case of MLM, the input sequence $x_1, x_2, ..., x_n$ of length n consists of individual tokens, and the output labels $y_1, y_2, ..., y_n$ are identical to the input sequence, i.e., $y_i = x_i$. A certain fraction k of the input tokens, randomly selected, are masked by replacing them with a special *¡MASK¿* token. These masked indices are grouped together in a set $\mathcal{C}$. The MLM objective is defined as a cross-entropy loss on the model's predictions $\hat{y}_i$ at the masked locations $j \in \mathcal{C}$

and is employed to optimize all the parameters θ of the model by minimizing the loss:

$$\mathcal{L}_{MLM}(x, y) = -\sum_{j \in \rfloor} logP(y_j|x; \theta) \tag{4.1}$$

The Transformer architecture used for MLM involves l layers, each containing $\langle$ self-attention heads. Let s_k be the input activations to layer k of this model, with $|s_k| = n$. The initial input activations s_1 are equivalent to the input sequence x, such that $s_1 = s = x$. For every position p in the output, each attention head in layer k induces a probability distribution over all positions in the input s_k. Specifically, the attention activations for a single head, denoted as a function of s and described by Equation 1 in Vaswani et al. [168], can be expressed as follows:

$$\mathbf{H}(s) = softmax(\frac{QK^T}{\sqrt{d_k}}) \in R^{n \times n} \tag{4.2}$$

the query and key matrices of dimension d_k are denoted by Q and K, respectively. For notational convenience, we drop the dependence on the input sequence s in the following sections. The attention paid by token p in the head's output layer to token q in the head's input layer is represented by the scalar $\mathbf{H}(s)[p, q]$.

4.4.2 Syntax Pattern Attention Guiding

The technique of attention guiding has been introduced to encourage self-attention heads to allocate more attention to predefined important positions of tokens [53]. This approach

can function as an auxiliary objective to regularize the fine-tuning process of downstream tasks [53, 174]. To guide an attention head, a mean squared error (MSE) loss is applied to $\mathbf{H}$ using a pre-defined pattern $\mathbf{P}(s) \equiv \mathbf{P} \in R^{n \times n}$, where $||\cdot||_F$ denotes the Frobenius norm:

$$\mathcal{L}_{ag} = ||\mathbf{H} - \mathbf{P}||_F \tag{4.3}$$

Figure 4.4 presents four examples of attention guiding patterns for a given code snippet. Specifically, Figure 4.4-(a) illustrates the attention guiding pattern that makes self-attention heads focus on the first [CLS] token. On the other hand, Figure 4.4-(b) depicts the pattern that guides self-attention heads to focus on the next tokens.

In Section 4.3, our analysis of attention weights reveals a bias in the self-attention heads of CodeBERT towards certain syntax tokens, such as identifiers and modifiers, as well as specific AST elements like method signatures. In order to capitalize on this insight and encourage the self-attention heads to focus more on critical programming language information, we introduce two sets of syntax attention guiding patterns: syntax token attention patterns and AST elements patterns.

- **Syntax token attention patterns** guide self-attention heads focusing on specific syntax type token positions, such as *idetifier*, *keywords*, *operator*, *data types* in a given source code sequence. As an example:

$$\mathbf{P}_{Syntax}[p, q] = \begin{cases} 1 & q_{Syntax} = Identifier \\ 0 & otherwise \end{cases} \tag{4.4}$$

  where q_{type} is the syntax type of source code token q. Two attention guiding patterns that focus on identifier and operator syntax tokens are presented in Figure 4.4-(c) and

(d), respectively.

- **Abstract syntax tree elements attention patterns** guide attention heads focusing on token positions belong to particular AST elements, such as *method signatures*, *if else elements*, and *return elements*. As an example:

$$\mathbf{P}_{AST}[p,q] = \begin{cases} 1 & q_{AST} = Return \\ 0 & otherwise \end{cases} \tag{4.5}$$

where q_{ast} belongs to the return elements in AST for input source code sequence.

SyntaGuid Loss Function. We apply the attention loss in Equation 4.3 to each head in each layer to obtain the overall source code syntax attention guidance (SAG) loss:

$$\mathcal{L}_{SAG}(x) = \sum_{k=1}^{\ell} \sum_{j=1}^{h} \mathcal{L}_{ag} \times I(k,j) \tag{4.6}$$

where $I(k,j)$ denotes an indicator function which is 1 only if the j_th head in layer k is being guided.

In principle, this loss permits any choice of patterns for each $\mathbf{P}kj$. However, for the sake of simplicity in our experiments, we guide a specific head number to the same pattern across all layers. That is, $\mathbf{P}j$ is constant for all layers. We use the gradients from this loss to update all the model's parameters, including the feedforward and input embedding layers. It is important to note that this loss depends solely on the input x and not on the labels y.

Finally, the overall optimization objective is obtained by combining the attention guidance (AG) loss with the MLM loss:

$$\mathcal{L}(\theta) = E_{(x,y)\mathcal{D}}[\mathcal{L}_{MLM} + \alpha \cdot \mathcal{L}_{SAG}] \tag{4.7}$$

where α is a hyper-parameter that controls the scale that we apply on selected heads for attention guiding. According to Deshpande et al. [53], the $\mathcal{L}_{SAG}$ converges faster than $\mathcal{L}_{MLM}$. We linearly decay α from an initial value $\alpha_0 = 1$ to 0 as the fine-tuning progresses.

4.4.3 Syntax attention patterns

Based on the attention bias results presented in Section 4.3, we have observed that the self-attention heads of fine-tuned CodeBERT assign significantly higher attention weights to certain syntax tokens, including identifiers, modifiers, operators, basic data types, separators, keywords, and string tokens. Additionally, the self-attention heads of CodeBERT also assign greater attention weights to specific code structures, such as method signatures, if-else elements, and return elements. Therefore, we propose the following attention guiding patterns for syntax token attention guiding during pre-trained model fine-tuning:

1. `[Modifier]` attends to the modifier syntax tokens.
2. `[Separator]` attends to the separator syntax tokens.
3. `[Key]` attends to the keyword syntax tokens.
4. `[Identifier]` attends to the identifier syntax tokens .
5. `[DataType]` attends to the basic data type syntax tokens.
6. `[Operator]` attends to the operator syntax tokens.
7. `[String]` attends to the string syntax tokens.

And, following abstract syntax tree attention guiding patterns:

1. `[MethodSignature]` attends to tokens belonging to the method signature AST elements.

2. `[IfElseElement]` attends to tokens belonging to the if else AST elements.

3. `[ReturnElement]` attends to tokens belonging to the return AST elements.

Furthermore, to enable a comprehensive evaluation of our proposed attention guiding patterns, we compare their performance with the local and global attention patterns proposed by Deshpande et al. [53]. The global attention patterns focus self-attention heads on global position, such as `[First]`, `[CLS]`, and `[SEP]`. And the local attention patterns either focus on the next or previous tokens, such as [NEXT] and [PREV]. This enables us to conduct a fair and thorough analysis of the effectiveness of our proposed attention guiding patterns compared to other existing patterns.

4.5 Evaluation

In this section, we first outline the experimental setup details to empirically evaluate the efficacy of our proposed syntax pattern Attention Guiding (AG) mechanism, and then present the experimental results for the following research questions:

- **RQ3: What is the efficacy of the proposed syntax pattern attention guiding mechanism on software engineering tasks?**
- **RQ4: Among the syntax patterns considered in this study, which pattern yields the most notable impact on the downstream task performance?**

4.5.1 Experimental setup

In this study, we employ the software engineering tasks of code clone detection, cloze test, and code translation, which were previously described in Section 4.3, along with their respective datasets. It is noteworthy that the AG patterns proposed by Deshpande et al. [53] are intended for natural languages, whereas our guiding patterns are specifically designed for source code. As a result, we replicated their AG patterns to compare their effectiveness with our code-specific patterns. Furthermore, we conducted an ablation study to determine the significance of each syntax token and AST AG patterns.

Implementation details

To ensure comparability across different experimental settings, we select CodeBERT [59] as the foundational pre-trained model for all our evaluations. CodeBERT is a prominent pre-trained model specifically designed for code, and has served as the foundation for RoBERTa [120], a widely-used language model in other programming language modeling studies [71, 91, 190, 172, 170, 177].

Basic model fine-tune. We tune the learning rate is 5e-5 for two epochs. The batch size for training is 16 and for testing is 32.

AG model. For attention guiding models, we guide a fraction of $\lambda \in \{\frac{1}{4}, \frac{2}{4}, \frac{3}{4}, 1\}$ of heads in each layer. We choose α for equation 4.7 from the set $\{1, 10, 100\}$ such that scales of the MLM loss and auxiliary loss are comparable at the beginning of the fine-tuning. To achieve fair comparison and reduce deep learning model's variance impact [141], we used five-fold cross validation for each basic and AG model.

Evaluation metrics

Consistent with prior works on code clone detection [82, 184, 176, 59], we evaluate the performance of our models using precision, recall, and F1 score [69]. Precision measures the accuracy of the predicted clone pairs. Whereas recall represents the proportion of actual clone pairs correctly predicted by the model. The F1 score is the harmonic mean of precision and recall, providing a balanced assessment of the model's performance.

The objective of the Cloze test is to predict the appropriate code token for a blank position in the context of the surrounding code. Consequently, we evaluate the prediction accuracy, which is calculated using the same formula as precision (i.e., the number of correct predictions divided by the total number of predictions).

Regarding the code translation task, we adopt the evaluation metrics proposed in CodeXGLUE. Specifically, we report three metrics: BLEU [140] score, CodeBLEU score [122], and accuracy (ACC). BLEU score is a commonly used metric for machine translation tasks, which measures the similarity between the generated code and the target code based on n-gram precision. CodeBLEU is a variant of BLEU proposed by CodeXGLUE, which takes into account not only surface-level matching but also grammatical and logical correctness, utilizing the AST and data-flow structure. In addition, we also evaluate the accuracy which calculates the exact match between generated code and the target code.

4.5.2 Evaluation results for syntax pattern attention guiding

Table 4.2 presents the empirical results of the three software engineering tasks. The baseline model, CodeBERT, is fine-tuned on all three tasks without the use of any attention guiding techniques. The global and local AG patterns are derived from Deshpande et al. [53]. Additionally, we propose syntax token and AST elements attention patterns that are tailored

Table 4.2: Evaluation results on software engineering tasks. AG represents attention guiding patterns. $\mathbf{AG_{global}}$ and $\mathbf{AG_{local}}$ attention patterns are proposed in [53]. $\mathbf{AG_{syntax}}$ and $\mathbf{AG_{AST}}$ are proposed in this study. The number with * means statistically significant (paired t-test) with corresponding default CodeBERT value.

Task name	**Cloze test**		**Code clone detetcion**				**Code translation**			
Evaluation metrics	**Acc.**	**Acc. Delta (%)**	**Pre.**	**Rec.**	**F1**	**F1 Delta (%)**	**BLEU**	**CodeBLEU**	**Acc.**	**Acc. delta (%)**
CodeBERT	64.57	-	0.947	0.935	0.941	-	71.99	85.10	59.00	-
CodeBERT + AG_{global}	64.71	0.14	0.951	0.933	0.942	0.10	74.83	85.35	59.31	0.3
CodeBERT + AG_{local}	64.86	0.29	0.935	0.940	0.938	-0.34	72.05	86.79	59.73	0.7
CodeBERT + AG_{global} + AG_{local}	64.95	0.38	0.948	0.947	0.948	0.70	73.95	86.73	60.21	1.21
CodeBERT + AG_{syntax}	65.88*	1.31	0.959	0.934	0.946	0.54	74.36	87.82	60.55	1.55
CodeBERT + AG_{AST}	66.27*	1.70	0.954	0.931	0.942	0.13	72.91	86.55	60.82	1.8
CodeBERT + AG_{syntax} + AG_{AST}	67.82*	3.25	0.962*	0.938	0.950*	0.88	76.88*	88.23*	61.93*	2.9

Table 4.3: Attention guiding performance on fixing wrong predictions by default CodeBERT

Task Name	**Cloze test**				**Code clone detection**				**Code Translation**			
	Correct prediction	Wrong prediction	Fixed prediction	Fix %	Correct prediction	Wrong prediction	Fixed prediction	Fix %	Correct prediction	Wrong prediction	Fixed prediction	Fix %
CodeBERT	2,412	1,323	-		393,399	22,017	-	-	590	410	-	
CodeBERT + AG_{global}	2,417	1,318	5	0.40%	395,061	20,355	1,662	7.55%	593	407	-3	0.76%
CodeBERT + AG_{local}	2,423	1,312	11	0.82%	388,414	27,002	-4,985	-22.64%	597	403	-7	1.78%
CodeBERT + AG_{global} + AG_{local}	2,426	1,309	14	1.07%	392,568	22,848	-831	-3.77%	602	398	-12	2.95%
CodeBERT + AG_{syntax}	2,461	1,274	49	3.70%	398,259	17,157	4,860	22.08%	606	395	-16	3.78%
CodeBERT + AG_{AST}	2,475	1,260	63	4.80%	396,431	18,985	3,033	13.77%	608	392	-18	4.44%
CodeBERT + AG_{syntax} + AG_{AST}	2,533	1,202	121	9.17%	399,630	15,786	6,231	28.30%	619	381	-29	7.15%

specifically for software engineering tasks.

In the context of cloze test, our experimental results reveal that AG patterns enhance CodeBERT's predictive accuracy. Specifically, the application of global AG patterns improves CodeBERT's prediction accuracy from 64.57 to 64.71, while local attention patterns lead to an accuracy improvement to 64.86. Notably, when both local and global AG patterns are utilized concurrently, the resulting accuracy is further enhanced to 64.95.

In contrast, the incorporation of syntax token AG patterns results in a significant improvement in CodeBERT's prediction accuracy, achieving a score of 65.88 (p-value ¡ 5.24e-07). Similarly, the utilization of AST AG patterns leads to an accuracy improvement of 66.27 (p-value ¡ 1.94e-08). Moreover, the simultaneous integration of syntax token and AST AG patterns results in an accuracy improvement to 67.82 (p-value ¡ 3.19e-10).

In our investigation of code clone detection, we found that the use of global AG patterns

improved the F1 score of CodeBERT from 0.941 to 0.942, while the application of local attention patterns resulted in a decrease in F1 score to 0.938. However, when both global and local attention patterns were applied simultaneously, the F1 score increased to 0.948. Interestingly, we observed that the use of our proposed syntax token and AST elements attention patterns resulted in a higher F1 score of 0.946 and 0.942, respectively. When both sets of attention patterns were applied simultaneously, the F1 score further increased to 0.950 (p-value ¡ 6.53e-05). It is worth mentioning that the default CodeBERT already achieved a very high F1 score of 0.941 in code clone detection, and the addition of AG patterns only resulted in a marginal improvement.

In the task of code translation, our empirical results show that the application of global AG patterns enhances the BLEU score of the default CodeBERT model from 71.99 to 74.83, CodeBLEU score from 85.10 to 85.35 and accuracy from 59.00 to 59.31. Similarly, local AG patterns improve the BLEU score to 72.05, CodeBLEU score to 86.79, and accuracy to 59.73. By using both patterns simultaneously, we achieve a further improvement in BLEU score (73.95), CodeBLEU score (86.73) and accuracy (60.21).

In contrast, our proposed syntax token AG pattern improves the default CodeBERT's code to code translation BLEU score to 74.36, CodeBLEU to 87.82, and accuracy to 60.55. Furthermore, the AST AG pattern enhances the BLEU score to 72.91, CodeBLEU to 86.55, and accuracy to 60.82. Finally, when both patterns are applied to the CodeBERT model, the BLEU score reaches 76.88, CodeBLEU score reaches 88.23 and accuracy improves to 61.93.

Our proposed attention guiding mechanism aims to improve CodeBERT prediction performance by fixing incorrect predictions by the model. In addition to evaluating the performance of the fine-tuned models using commonly used evaluation metrics for each software engineering task, we also investigated the effectiveness of the AG patterns in rectifying mispredicted samples. Table 4.3 presents an overview of the incorrectly predicted instances by the default CodeBERT model and the fine-tuned models incorporating various AG patterns. Notably,

for the cloze test, our proposed syntax token AG and AST AG patterns were successful in rectifying 9.17% of the mispredicted instances, outperforming the 1.07% instances corrected by the global and local AG patterns. For code clone detection, our proposed syntax attention patterns were effective in fixing 28.3% of the mispredicted instances. Interestingly, the global and local attention patterns resulted in an increased 3.77% of the mispredicted instances. Finally, for code translation, our proposed syntax AG patterns led to the correction of 7.15% of the mispredicted instances, whereas the global and local attention patterns only corrected 3.78% of the mispredicted instances.

Syntax token and AST attention guiding patterns have demonstrated significant performance improvements over the default CodeBERT model. Notably, these patterns have exhibited better performance on software engineering tasks than previously proposed global and local attention guiding patterns.

One interesting observation from Table 4.2 was that the efficacy of the proposed syntax token and AST elements AG patterns is more prominent for the Cloze test as compared to code translation. We posit that this is due to the relatively larger data size of Cloze test (50k) as compared to code translation (around 11.5k). So we sought to investigate the impact of training data size on the proposed AG patterns' effectiveness.

For this purpose, we randomly selected 25%, 50%, 75%, and 100% of the training set for Cloze test fine-tuning and conducted experiments. Our findings indicate that both syntax token AG patterns and AST AG patterns improve the fine-tuned CodeBERT model's performance on the Cloze test at different training data sizes. Combining both AG patterns always resulted in the best performance. When only 25% of training data was used, syntax token AG patterns performed better than AST AG patterns. However, after using 50% of training data, the AST AG patterns consistently outperformed the syntax token AG patterns. For a detailed breakdown of our findings, we refer the reader to Figure 4.5.

Syntax token and AST element AG patterns can improve fine-tuned CodeBERT model's performance with different training data sizes.

4.5.3 Ablation study results

To perform a comprehensive analysis of the effectiveness of the AG patterns on the CodeBERT model, we conducted an ablation study on each task using the best model, which consists of CodeBERT, syntax token AG patterns (AG_{syntax}), and AST AG patterns (AG_{AST}). Due to space limitations, we present the ablation study results for only one task, and the results for other tasks are available on our companion website [5]. The results for cloze are presented in table 4.4.

Our study revealed that the fine-tuning process of CodeBERT benefits significantly from both syntax token and AST AG patterns, as the removal of either of them led to a decrease in accuracy. Specifically, for syntax token AG patterns, the most significant drop in accuracy was observed when [Identifier], [Operator], and [Modifier] patterns were removed. Meanwhile, for AST AG patterns, method signatures were found to be the most impactful, followed by return elements and then if-else elements.

Furthermore, for code clone detection, our analysis showed that [Identifier], [Operator], and [Data type] were the most important syntax token AG patterns, while method signatures were the most important AST AG patterns. Similarly, for code translation, [Identifier], [Operator], and [Modifier] were the most important syntax token AG patterns, and method signatures were the most important AST AG patterns. Because of the limited space, their detailed experiment results are in our companion website [5].

Table 4.4: Ablation study results for cloze test. The number with * means statistically significant (P-value < 0.05)

		Accuracy	**Accuracy drop**
CodeBERT + AG_{syntax} + AG_{AST}		67.8213	-
Syntax token attention patterns (AG_{syntax})	w/o Identifier	64.7859	-3.04
	w/o Operator	64.8960*	-2.93
	w/o Modifier	64.9447*	-2.87
	w/o Data type	66.6900*	-1.13
	w/o Keyword	66.9548	-0.87
	w/o Separator	67.5993	-0.22
	w/o String	67.6523	-0.17
Abstract syntax element attention patterns (AG_{AST})	w/o Method signature	64.6881*	-3.13
	w/o If else element	64.9564*	-2.86
	w/o Return element	64.9806*	-2.84

4.6 Implications

Based on our findings and analyses, we provide the following implications for researchers and practitioners.

4.6.1 Implications for researchers

We demonstrate that fine-tuned CodeBERT assigns significantly greater weights to specific types of syntax tokens and AST elements when making correct predictions in Section 4.3. This provides a new perspective for interpreting attention-based models and analyzing the attention weight distribution in Transformer-based models. One intriguing future research entails investigating the applicability of these syntax tokens and AST elements and their associated weights for building defect prediction models. Another interesting future research direction would entail utilizing this information to build a tool for explaining the model's decisions to a developer. Also, we believe a study on a more extensive group of software engineering tasks and language models can uncover more syntax tokens and AST elements

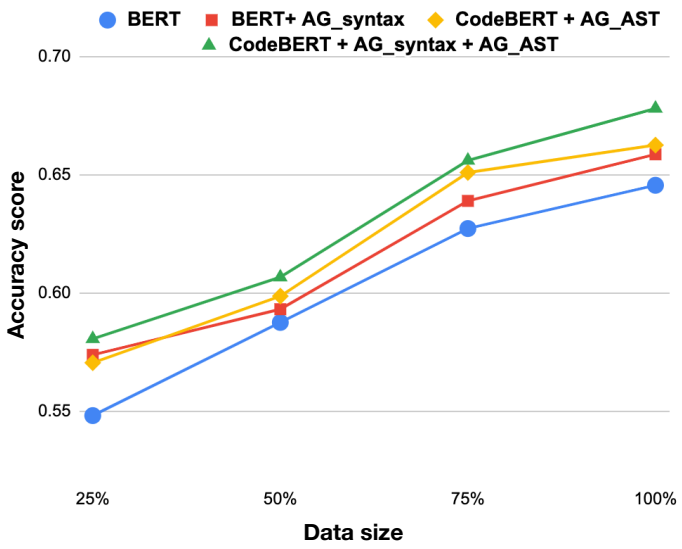


Figure 4.5: Results of CodeBERT and CodeBERT with various syntax AG on different amounts of training data

and opportunities to further improve the performances of fine-tuned models.

We propose a method to guide self-attention heads to pay more attention to critical token positions, though our approach only guides a fraction of the self-attention heads to focus on input source code sequence positions. However, there is potential for more fine-grained attention guidance, such as directing identifier tokens to pay attention to operator tokens or if-else elements. Such granular token-to-token attention analysis may prove valuable for guiding attention. Additionally, researchers have proposed utilizing contrastive learning methods or pruning unimportant source code to retrain the pre-trained language models. Combining syntax-based attention guidance with other methods may lead to further improvements in Transformer-based models for programming languages.

4.6.2 Implications for users

Based on the results presented in Table 4.3, it is evident that incorporating syntax attention guiding patterns can effectively rectify erroneous predictions without adding extra

data. Oversampling has been demonstrated to be an effective approach for enhancing the performance of machine learning models. However, utilizing Transformer-based models to automatically learn representation features for programming languages presents a challenge in achieving oversampling, as the learned representation features are challenging to employ for this purpose, in contrast to tabular data. Hence, our proposed attention guiding mechanism, which does not require any extra data, represents a promising option for improving Transformer-based models for software engineering tasks. Moreover, as illustrated in Figure 4.5, our attention guiding approach exhibits a robust performance across different data sizes, further highlighting its efficacy.

4.7 Related Works

4.7.1 Analyzing self-attention weight

Recent studies [24, 127, 138, 92] have investigated the attention assignment patterns of Transformer-based language models trained for software engineering tasks. For instance, Karmakar etal [92] applied four probing tasks on pre-trained code models to investigate whether pre-trained models can learn different aspects of source code such as syntactic, structural, surface-level, and semantic information. Wan et al. [170] showed that CodeBERT's attention aligns strongly with syntax structure of the code and it preserves the syntax structure of code in the intermediate representations of each Transformer layer. In addition, Zhang et al [197] and Sharma et al [158] reveal that CodeBERT, in general, pays more attention to certain types of tokens and statements. However, none of the prior studies investigated the alteration of attention weight distribution between correct and incorrect prediction groups and our study sets itself apart by showing that CodeBERT demonstrates a noteworthy bias toward assigning greater attention weights to particular syntax tokens and

statements when making correct predictions.

4.7.2 Guiding self-attention weight

Several studies have explored methods for guiding self-attention heads on important syntax tokens and statements in source code. For instance, Zhang et al.[197] propose a new pre-trained model for source code that guides attention to important syntax tokens by excluding unimportant or high-frequency tokens in the input source code sequence during pre-training. Similarly, Wang et al. [177] proposed another pre-trained model that guides self-attention heads' attention on symbolic and syntactic properties of source code using contrastive learning [95]. In contrast to focusing on the pre-training phase, our approach targets the fine-tuning stage for attention guiding and can improve the performance of the fine-tuned model more efficiently since fine-tuning requires significantly less time, computational resources, and data compared to pre-training.

4.8 Threats to Validity

We have taken care to ensure that our results are unbiased and have tried to eliminate the effects of random noise, but it's possible that our mitigation strategies may not have been effective.

Dataset Bias: It is important to note that our findings may not necessarily apply to all software engineering datasets and tasks, as we have only evaluated our approach on the publicly available BigCloneBench [176], CodeXGLUE code translation, and cloze test datasets [122]. However, these datasets have been used in previous studies [108, 193, 172, 42, 197]. Thus their quality and reliability are well-established. Moreover, the software engineering datasets we have considered are diverse in size, programming language, and

complexity, which mitigates concerns of bias due to dataset selection. Therefore, we believe our selection is appropriate to address the research questions.

Bias Due to Syntax Extraction: One potential bias can arise from the syntax extraction process. Specifically, we utilized javalang [13] to extract the source code token syntax types and tree-sitter-java [12] to extract the AST elements, and the selected sets of syntax types and structures were obtained from Aljehane et al. [26] study about the attention difference between expert and novice programmers' when debugging. Even though different syntax extraction libraries may yield different results, the libraries we used have been widely adopted in previous research [158, 109, 162].

Bias Due to Pre-trained Language Model: Our study focuses on examining the attention-weight assignment differences in fine-tuned language models. Even though there any many PLMs, we opted to use one of the state-of-the-art PLMs named CodeBERT which may have introduced unwarranted bias to our study.

Bias Due to Implementation: To address the potential bias, we took several measures to minimize errors in our study. First, we relied on existing implementations in CodeXGLUE for fine-tuning CodeBERT without attention guiding and applying global and local attention guiding patterns, as well as our proposed syntax attention guiding patterns. Additionally, we thoroughly tested our code and data to identify and correct any potential errors. However, we cannot completely rule out the possibility of implementation bias.

4.9 Conclusion

In this study, we conducted an investigation to ascertain the extent to which self-attention heads in Transformer-based models allocate attention weights to source code syntax tokens and AST elements during correct and incorrect predictions. Our empirical results revealed

that self-attention heads tend to assign significantly greater attention weights to particular syntax tokens and AST elements. Drawing on this knowledge, we proposed SyntaGuid, which facilitates an improved performance of CodeBERT on downstream tasks by guiding self-attention heads toward critical elements of source code. Our experimental results demonstrate that SyntaGuid substantially enhances the performance of fine-tuned CodeBERT on various software engineering tasks, which improves overall performance up to 3.25% and fixes up to 28.30% of wrong predictions.

Chapter 5

Conclusion

In the rapidly evolving domain of machine learning, particularly as it intersects with software engineering, the nuances and intricacies of feature bias have emerged as pivotal areas of inquiry. This dissertation represents a comprehensive exploration into the realm of feature bias, its manifestations, and its implications for machine learning models tailored specifically for software engineering tasks.

Chapter 2 embarked on a meticulous dissection of feature bias, elucidating its multifaceted nature and the potential ramifications it holds for the integrity and reliability of machine learning models. The challenges posed by feature bias are not merely theoretical but have tangible repercussions, especially when models are confronted with under-represented features in datasets. The introduction of "SifterJIT" in this chapter stands as a testament to the innovative approaches that can be harnessed to counteract these challenges, offering a beacon of hope for more balanced and unbiased machine learning applications.

Building upon the foundational insights gleaned from Chapter 2, Chapter 3 transitioned our focus towards the nuanced methodologies of "Bias Guided Misprediction Diagnoser" and the "Mispredicted Area Upweight Sampling" technique. These tools, borne out of rigorous

research and empirical testing, were presented as groundbreaking solutions to address areas of misprediction. By leveraging the knowledge of feature bias, these methodologies underscore the dissertation's commitment to not only identifying challenges but also proactively seeking solutions to enhance the robustness and reliability of machine learning models.

In Chapter 4, our exploration delved deeper into the sophisticated world of Transformer-based models. These state-of-the-art architectures, while formidable in their capabilities, are not immune to biases. The chapter's exploration of potential attention biases and the subsequent introduction of the "Program Syntax-based Attention Guiding Mechanism" showcased a pioneering approach. This mechanism, designed to optimize attention allocation, stands as a testament to the continuous evolution and refinement of machine learning methodologies.

As we reflect upon the journey undertaken in this dissertation, it becomes evident that the landscape of machine learning, especially in the context of software engineering, is rife with challenges and opportunities. The insights, methodologies, and solutions presented herein are not merely academic exercises but are foundational pillars that will undoubtedly shape the trajectory of future research in this domain. As technology continues its relentless march forward, it is our fervent hope that the findings of this dissertation will serve as a guiding light, ensuring that machine learning models in software engineering remain robust, reliable, and free from the shackles of bias.

Bibliography

[1] Ai fairness 360. https://aif360.mybluemix.net/. Accessed: 2021-05-1.

[2] Amazon codewhisperer: build applications faster with the ml-powered coding companion. https://aws.amazon.com/codewhisperer/. Accessed: 2023-03-29.

[3] The apache lucene™ project develops open-source search software. https://lucene.apache.org/. Accessed: 2023-03-29.

[4] Apache poi - the java api for microsoft documents. https://poi.apache.org. Accessed: 2023-03-29.

[5] Attention bias analysis and attentiong guiding experiment results companion website. https://github.com/syntaxGuiding/SytaGuid. Accessed: 2023-03-29.

[6] Bias guided misprediction explanation companion code and experimental results website. https://github.com/Jirigesi/BGMD_MAPS. Accessed: 2022-08-27.

[7] Bigclonebench github webpage. github.com/clonebench/BigCloneBench. Accessed: 2023-03-29.

[8] Competitive programming with alphacode. https://www.deepmind.com/blog/competitive-programming-with-alphacode. Accessed: 2023-03-29.

[9] Copilot: Your ai pair programmer. https://github.com/features/copilot. Accessed: 2023-03-29.

[10] Dataset details for experiments in this study. https://github.com/Jirigesi/BGMD_MAPS/blob/main/README.md#:~:text=0.0%2C%20max_samples%3DNone-,Data,5%2C940,-Footer. Accessed: 2022-08-27.

[11] Default parameters of support vector machines, decision trees and random forest learners used in this study. https://github.com/Jirigesi/BGMD_MAPS/blob/main/README.md#:~:text=SVM,0.0%2C%20max_samples%3DNone. Accessed: 2022-08-27.

[12] Java program for tree-sitter. https://github.com/tree-sitter/tree-sitter-java. Accessed: 2023-03-29.

[13] javalang python library for working with java source code. https://github.com/c2nes/javalang. Accessed: 2023-03-29.

[14] Kaggle machine learning competitions. https://www.kaggle.com/. Accessed: 2022-08-27.

[15] Open stack data set. https://docs.openstack.org/wallaby/?_ga=2.205840979.1124305833.1619313296-1069099767.1617679651. Accessed: 2020-07-1.

[16] Openstack. https://www.openstack.org/. Accessed: 2021-05-1.

[17] Proceedings of machine learning research. https://proceedings.mlr.press/. Accessed: 2022-08-27.

[18] The project organization for the antlr parser generator. https://github.com/antlr/. Accessed: 2023-03-29.

[19] Qt. https://www.qt.io/. Accessed: 2021-05-1.

[20] University of california at irvine machine learning dataset repository. https://archive.ics.uci.edu/ml/datasets.php?format=&task=cla&att=&area=&numAtt=&numIns=&type=&sort=attDown&view=table. Accessed: 2022-08-27.

[21] What-if tool. https://pair-code.github.io/what-if-tool/. Accessed: 2021-05-1.

[22] A. Agarwal, A. Beygelzimer, M. Dudík, J. Langford, and H. Wallach. A reductions approach to fair classification. In *International Conference on Machine Learning*, pages 60–69. PMLR, 2018.

[23] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha. Black box fairness testing of machine learning models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 625–635, 2019.

[24] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[25] T. Ahmed and P. Devanbu. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1443–1455, 2022.

[26] S. Aljehane, B. Sharif, and J. Maletic. Determining differences in reading behavior between experts and novices by investigating eye movement on source code constructs during a bug fixing task. In *ACM Symposium on Eye Tracking Research and Applications*, pages 1–6, 2021.

[27] H. Altae-Tran, B. Ramsundar, A. S. Pappu, and V. Pande. Low data drug discovery with one-shot learning. *ACS central science*, 3(4):283–293, 2017.

[28] S. Barocas, M. Hardt, and A. Narayanan. Fairness and machine learning: Limitations and opportunities, 2018.

[29] K. E. Bennin, K. Toda, Y. Kamei, J. Keung, A. Monden, and N. Ubayashi. Empirical evaluation of cross-release effort-aware defect prediction models. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 214–221. IEEE, 2016.

[30] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–18. ACM, 2007.

[31] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced? bias in bug-fix datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 121–130, 2009.

[32] S. Biswas and H. Rajan. Do the machine learning models on a crowd sourced platform exhibit bias? an empirical study on model fairness. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 642–653, 2020.

[33] J. M. Bland and D. G. Altman. Multiple significance tests: the bonferroni method. *Bmj*, 310(6973):170, 1995.

[34] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah. Signature verification using a" siamese" time delay neural network. *Advances in neural information processing systems*, 6:737–744, 1993.

[35] J. Byrd and Z. Lipton. What is the effect of importance weighting in deep learning? In *International Conference on Machine Learning*, pages 872–881. PMLR, 2019.

[36] J. Cendrowska. Prism: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27(4):349–370, 1987.

[37] J. Chakraborty, S. Majumder, and T. Menzies. Bias in machine learning software: why? how? what to do? In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 429–440, 2021.

[38] J. Chakraborty, S. Majumder, Z. Yu, and T. Menzies. Fairway: a way to build fair ml software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 654–665, 2020.

[39] J. Chakraborty, K. Peng, and T. Menzies. Making fair ml software using trustworthy explanation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1229–1233. IEEE, 2020.

[40] J. Chakraborty, T. Xia, F. M. Fahid, and T. Menzies. Software engineering for fairness: A case study with hyperparameter optimization. *arXiv preprint arXiv:1905.05786*, 2019.

[41] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[42] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. L. Goues. Varclr: Variable semantic representation pre-training via contrastive learning. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2327–2339, 2022.

[43] S. Chen, S. Bateni, S. Grandhi, X. Li, C. Liu, and W. Yang. Denas: automated rule generation by knowledge extraction from neural networks. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 813–825, 2020.

[44] N. Chirkova and S. Troshin. Empirical study of transformers for source code. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 703–715, 2021.

[45] Y. Chung, T. Kraska, N. Polyzotis, K. H. Tae, and S. E. Whang. Automated data slicing for model validation: A big data-ai integration approach. *IEEE Transactions on Knowledge and Data Engineering*, 32(12):2284–2296, 2019.

[46] J. Cito, I. Dillig, S. Kim, V. Murali, and S. Chandra. Explaining mispredictions of machine learning models using rule induction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 716–727, 2021.

[47] J. Cito, I. Dillig, V. Murali, and S. Chandra. Counterfactual explanations for models of code. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 125–134. IEEE, 2022.

[48] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning. What does bert look at? an analysis of bert's attention. *arXiv preprint arXiv:1906.04341*, 2019.

[49] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[50] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

[51] E. Creager, J.-H. Jacobsen, and R. Zemel. Environment inference for invariant learning. In *International Conference on Machine Learning*, pages 2189–2200. PMLR, 2021.

[52] H. K. Dam, T. Tran, and A. Ghose. Explainable software analytics. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 53–56, 2018.

[53] A. Deshpande and K. Narasimhan. Guiding attention for self-supervised learning with transformers. *arXiv preprint arXiv:2010.02399*, 2020.

[54] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[55] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2018.

[56] J. Duchi, P. Glynn, and H. Namkoong. Statistics of robust optimization: A generalized empirical likelihood approach. *arXiv preprint arXiv:1610.03425*, 2016.

[57] A. Elnaggar, W. Ding, L. Jones, T. Gibbs, T. Feher, C. Angerer, S. Severini, F. Matthes, and B. Rost. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*, 2021.

[58] Y. Fan, D. A. da Costa, D. Lo, A. Hassan, and L. Shanping. The impact of mislabeled changes by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 2020.

[59] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[60] M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, and F. Herrera. A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(4):463–484, 2011.

[61] A. Galassi, M. Lippi, and P. Torroni. Attention in natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32(10):4291–4308, 2020.

[62] J. Gesi, J. Li, and I. Ahmed. An empirical examination of the impact of bias on just-in-time defect prediction. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2021.

[63] J. Gesi, X. Shen, Y. Geng, Q. Chen, and I. Ahmed. Leveraging feature bias for scalable misprediction explanation of machine learning models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023.

[64] A. Ghazikhani, H. S. Yazdi, and R. Monsefi. Class imbalance handling using wrapper-based random oversampling. In *20th Iranian Conference on Electrical Engineering (ICEE2012)*, pages 611–616. IEEE, 2012.

[65] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 789–800. IEEE, 2015.

[66] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 171–180. ACM, 2012.

[67] E. Giger, M. Pinzger, and H. C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 83–92. ACM, 2011.

[68] K. Goel, A. Gu, Y. Li, and C. Ré. Model patching: Closing the subgroup performance gap with data augmentation. *arXiv preprint arXiv:2008.06775*, 2020.

[69] C. Goutte and E. Gaussier. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *Advances in Information Retrieval: 27th European Conference on IR Research, ECIR 2005, Santiago de Compostela, Spain, March 21-23, 2005. Proceedings 27*, pages 345–359. Springer, 2005.

[70] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. Reflections on the nasa mdp data sets. *IET software*, 6(6):549–558, 2012.

[71] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[72] M. Habayeb, S. S. Murtaza, A. Miranskyy, and A. B. Bener. On the use of hidden markov model to predict the time to fix bugs. *IEEE Transactions on Software Engineering*, 44(12):1224–1244, 2017.

[73] M. Hardt, E. Price, and N. Srebro. Equality of opportunity in supervised learning. *Advances in neural information processing systems*, 29, 2016.

[74] F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, and M. Kim. Is neuron coverage a meaningful measure for testing deep neural networks? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 851–862, 2020.

[75] A. E. Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st international conference on software engineering*, pages 78–88. IEEE, 2009.

[76] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

[77] K. Herzig and N. Nagappan. Empirically detecting false test alarms using association rules. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 39–48. IEEE, 2015.

[78] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45. IEEE, 2019.

[79] T. Hoang, H. J. Kang, D. Lo, and J. Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, 2020.

[80] X. Huo, M. Li, Z.-H. Zhou, et al. Learning unified features from natural and programming languages for locating buggy source code. In *IJCAI*, volume 16, pages 1606–1612, 2016.

[81] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*, 2020.

[82] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[83] N. Jiang, T. Lutellier, and L. Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[84] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–289. Ieee, 2013.

[85] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu. Dictionary learning based software defect prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 414–423, 2014.

[86] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[87] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE international conference on software maintenance*, pages 1–10. IEEE, 2010.

[88] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto. The effects of over and under sampling on fault-prone module detection. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 196–204. IEEE, 2007.

[89] Y. Kamei and E. Shihab. Defect prediction: Accomplishments and future challenges. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 5, pages 33–45. IEEE, 2016.

[90] 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. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

[91] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pages 5110–5121. PMLR, 2020.

[92] A. Karmakar and R. Robbes. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1332–1336. IEEE, 2021.

[93] C. Khanan, W. Luewichana, K. Pruktharathikoon, J. Jiarpakdee, C. Tantithamthavorn, M. Choetkiertikul, C. Ragkhitwetsagul, and T. Sunetnanta. Jitbot: An explainable just-in-time defect prediction bot. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1336–1339. IEEE, 2020.

[94] F. Khani, A. Raghunathan, and P. Liang. Maximum weighted loss discrepancy. *arXiv preprint arXiv:1906.03518*, 2019.

[95] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan. Supervised contrastive learning. *Advances in neural information processing systems*, 33:18661–18673, 2020.

[96] E. Kim, D. Gopinath, C. Pasareanu, and S. A. Seshia. A programmatic and semantic approach to explaining and debugging neural network based object detectors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11128–11137, 2020.

[97] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[98] S. Kim and E. J. Whitehead Jr. How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 173–174. ACM, 2006.

[99] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 481–490. IEEE, 2011.

[100] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *29th International Conference on Software Engineering (ICSE'07)*, pages 489–498. IEEE, 2007.

[101] G. Koch, R. Zemel, and R. Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2. Lille, 2015.

[102] S. Kotsiantis and D. Kanellopoulos. Discretization techniques: A recent survey. *GESTS International Transactions on Computer Science and Engineering*, 32(1):47–58, 2006.

[103] S. Kotsiantis, D. Kanellopoulos, P. Pintelas, et al. Handling imbalanced datasets: A review. *GESTS International Transactions on Computer Science and Engineering*, 30(1):25–36, 2006.

[104] P. Kourouklidis, D. Kolovos, N. Matragkas, and J. Noppen. Towards a low-code solution for monitoring machine learning model performance. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–8, 2020.

[105] N. Kumar, A. Berg, P. N. Belhumeur, and S. Nayar. Describable visual attributes for face verification and image search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(10):1962–1977, 2011.

[106] H. Lakkaraju, E. Kamar, R. Caruana, and J. Leskovec. Interpretable & explorable approximations of black box models. *arXiv preprint arXiv:1707.01154*, 2017.

[107] G. Lample and A. Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

[108] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

[109] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel. Hyperast: Enabling efficient analysis of software histories at scale. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[110] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[111] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[112] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 311–321, 2011.

[113] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering*, 25(2):279–313, 2018.

[114] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead. Does bug prediction support human developers? findings from a google case study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 372–381. IEEE, 2013.

[115] J. Li, P. He, J. Zhu, and M. R. Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE, 2017.

[116] K. Li, W. Zhang, Q. Lu, and X. Fang. An improved smote imbalanced data classification method based on support degree. In *2014 international conference on identification, information and knowledge in the internet of things*, pages 34–38. IEEE, 2014.

[117] E. Z. Liu, B. Haghgoo, A. S. Chen, A. Raghunathan, P. W. Koh, S. Sagawa, P. Liang, and C. Finn. Just train twice: Improving group robustness without training group information. In *International Conference on Machine Learning*, pages 6781–6792. PMLR, 2021.

[118] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

[119] S. Liu, X. Chen, W. Liu, J. Chen, Q. Gu, and D. Chen. Fecar: A feature selection framework for software defect prediction. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 426–435. IEEE, 2014.

[120] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[121] V. López, A. Fernández, S. García, V. Palade, and F. Herrera. An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics. *Information sciences*, 250:113–141, 2013.

[122] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[123] W. Ma, M. Zhao, X. Xie, Q. Hu, S. Liu, J. Zhang, W. Wang, and Y. Liu. Is self-attention powerful to learn code syntax and semantics? *arXiv preprint arXiv:2212.10017*, 2022.

[124] N. Madaan, I. Padhi, N. Panwar, and D. Saha. Generate your counterfactuals: Towards controlled counterfactual generation for text. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 13516–13524, 2021.

[125] R. Malhotra and S. Kamal. An empirical study to investigate oversampling methods for improving software defect prediction using imbalanced data. *Neurocomputing*, 343:120–140, 2019.

[126] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[127] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347. IEEE, 2021.

[128] S. McIntosh and Y. Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 44(5):412–428, 2017.

[129] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan. A survey on bias and fairness in machine learning. *arXiv preprint arXiv:1908.09635*, 2019.

[130] I. Melekhov, J. Kannala, and E. Rahtu. Siamese network features for image matching. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 378–383. IEEE, 2016.

[131] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[132] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering*, 23(5):2901–2947, 2018.

[133] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318. IEEE, 2010.

[134] M. Naseriparsa and M. M. R. Kashani. Combination of pca with smote resampling to boost the prediction rate in lung cancer dataset. *arXiv preprint arXiv:1403.1949*, 2014.

[135] P. Neculoiu, M. Versteegh, and M. Rotaru. Learning text similarity with siamese recurrent networks. In *Proceedings of the 1st Workshop on Representation Learning for NLP*, pages 148–157, 2016.

[136] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2017.

[137] M. Owhadi-Kareshk, S. Nadi, and J. Rubin. Predicting merge conflicts in collaborative software development. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[138] M. Paltenghi and M. Pradel. Thinking like a developer? comparing the attention of humans with neural models of code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 867–879. IEEE, 2021.

[139] R. Pan, V. Le, N. Nagappan, S. Gulwani, S. Lahiri, and M. Kaufman. Can program synthesis be used to learn merge conflict resolutions? an empirical analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 785–796. IEEE, 2021.

[140] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[141] H. V. Pham, M. Kim, L. Tan, Y. Yu, and N. Nagappan. Deviate: A deep learning variance testing framework. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1286–1290. IEEE, 2021.

[142] L. Phan, H. Tran, D. Le, H. Nguyen, J. Anibal, A. Peltekian, and Y. Ye. Cotext: Multi-task learning with code-text transformer. *arXiv preprint arXiv:2105.08645*, 2021.

[143] C. Pornprasit and C. Tantithamthavorn. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. *arXiv preprint arXiv:2103.07068*, 2021.

[144] L. Qiao and Y. Wang. Effort-aware and just-in-time defect prediction with neural network. *PloS one*, 14(2):e0211359, 2019.

[145] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[146] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[147] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM, 2011.

[148] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu. Sample size vs. bias in defect prediction. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 147–157, 2013.

[149] A. Rajkomar, M. Hardt, M. D. Howell, G. Corrado, and M. H. Chin. Ensuring fairness in machine learning to advance health equity. *Annals of internal medicine*, 169(12):866–872, 2018.

[150] Z. A. Rana, M. M. Awais, and S. Shamail. Impact of using information gain in software defect prediction models. In *International Conference on Intelligent Computing*, pages 637–648. Springer, 2014.

[151] M. T. Ribeiro, S. Singh, and C. Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[152] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, volume 177, 2006.

[153] A. Ross, A. Marasović, and M. E. Peters. Explaining nlp models via minimal contrastive editing (mice). *arXiv preprint arXiv:2012.13985*, 2020.

[154] D. Ryu, J.-I. Jang, and J. Baik. A hybrid instance selection using nearest-neighbor for cross-project defect prediction. *Journal of Computer Science and Technology*, 30(5):969–980, 2015.

[155] S. Sagawa, P. W. Koh, T. B. Hashimoto, and P. Liang. Distributionally robust neural networks for group shifts: On the importance of regularization for worst-case generalization. *arXiv preprint arXiv:1911.08731*, 2019.

[156] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[157] N. Seliya and T. M. Khoshgoftaar. The use of decision trees for cost-sensitive classification: an empirical study in software quality prediction. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(5):448–459, 2011.

[158] R. Sharma, F. Chen, F. Fard, and D. Lo. An exploratory study on code attention in bert. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 437–448, 2022.

[159] D. Song, W. Lee, and H. Oh. Context-aware and data-driven feedback generation for programming assignments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 328–340, 2021.

[160] M. Sundararajan, A. Taly, and Q. Yan. Axiomatic attribution for deep networks. In *International conference on machine learning*, pages 3319–3328. PMLR, 2017.

[161] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE, 2014.

[162] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[163] S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song. An investigation of cross-project learning in online just-in-time software defect prediction. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 554–565. IEEE, 2020.

[164] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE, 2015.

[165] Y. Tian, Z. Zhong, V. Ordonez, G. Kaiser, and B. Ray. Testing dnn image classifiers for confusion & bias errors. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1122–1134, 2020.

[166] F. Tramer, V. Atlidakis, R. Geambasu, D. Hsu, J.-P. Hubaux, M. Humbert, A. Juels, and H. Lin. Fairtest: Discovering unwarranted associations in data-driven applications. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 401–416. IEEE, 2017.

[167] I. van der Linden, H. Haned, and E. Kanoulas. Global aggregations of local explanations for black box models. *arXiv preprint arXiv:1907.03039*, 2019.

[168] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[169] M. Veale and R. Binns. Fairer machine learning in the real world: Mitigating discrimination without collecting sensitive data. *Big Data & Society*, 4(2):2053951717743530, 2017.

[170] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2377–2388, 2022.

[171] A. Wang, A. Narayanan, and O. Russakovsky. Revise: A tool for measuring and mitigating bias in visual datasets. In *European Conference on Computer Vision*, pages 733–751. Springer, 2020.

[172] D. Wang, Z. Jia, S. Li, Y. Yu, Y. Xiong, W. Dong, and X. Liao. Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th International Conference on Software Engineering*, pages 287–298, 2022.

[173] Q. Wang, J. Gao, and Y. Yuan. Embedding structured contour and location prior in siamesed fully convolutional networks for road detection. *IEEE Transactions on Intelligent Transportation Systems*, 19(1):230–241, 2017.

[174] S. Wang, Z. Chen, Z. Ren, H. Liang, Q. Yan, and P. Ren. Paying more attention to self-attention: Improving pre-trained language models via attention guiding. *arXiv preprint arXiv:2204.02922*, 2022.

[175] S. Wang and X. Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013.

[176] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[177] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*, 2021.

[178] Y. Wang and H. Li. Code completion by modeling flattened abstract syntax trees as graphs. *Proceedings of AAAIConference on Artificial Intellegence*, 2021.

[179] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[180] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys (CSUR)*, 53(3):1–34, 2020.

[181] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

[182] K. Q. Weinberger, J. Blitzer, and L. K. Saul. Distance metric learning for large margin nearest neighbor classification. In *Advances in neural information processing systems*, pages 1473–1480, 2006.

[183] M. Wen, R. Wu, and S.-C. Cheung. How well do change sequences predict defects? sequence learning from software changes. *IEEE Transactions on Software Engineering*, 46(11):1155–1175, 2018.

[184] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 87–98, 2016.

[185] C.-P. Wong, P. Santiesteban, C. Kästner, and C. Le Goues. Varfix: balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–366, 2021.

[186] B. Woodworth, S. Gunasekar, M. I. Ohannessian, and N. Srebro. Learning non-discriminatory predictors. In *Conference on Learning Theory*, pages 1920–1953. PMLR, 2017.

[187] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25, 2011.

[188] T. Wu, M. T. Ribeiro, J. Heer, and D. Weld. Errudite: Scalable, reproducible, and testable error analysis. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[189] Y. Xiong and B. Wang. L2s: A framework for synthesizing the most probable program under a specification. *TOSEM: ACM Transactions on Software Engineering and Methodology*, 2021.

[190] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[191] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE, 2015.

[192] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

[193] Z. Yang, J. Shi, J. He, and D. Lo. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1482–1493, 2022.

[194] Y. Yao, M. Xu, Y. Wang, D. J. Crandall, and E. M. Atkins. Unsupervised traffic accident detection in first-person videos. *arXiv preprint arXiv:1903.00618*, 2019.

[195] B. Yu, H. Qi, Q. Guo, F. Juefei-Xu, X. Xie, L. Ma, and J. Zhao. Deeprepair: Style-guided repairing for deep neural networks in the real-world operational environment. *IEEE Transactions on Reliability*, 2021.

[196] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 182–191, 2014.

[197] Z. Zhang, H. Zhang, B. Shen, and X. Gu. Diet code is healthy: Simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1073–1084, 2022.

[198] L. Zhao, Z. Shang, L. Zhao, A. Qin, and Y. Y. Tang. Siamese dense neural network for software defect prediction with small data. *IEEE Access*, 7:7663–7677, 2018.

[199] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100, 2009.

[200] T. Zimmermann, N. Nagappan, and A. Zeller. Predicting bugs from history. In *Software evolution*, pages 69–88. Springer, 2008.