An Empirical Examination of the Impact of Bias on Just-in-time Defect Prediction

Jiri Gesi
University of California, Irvine
Irvine, CA, USA
fjiriges@uci.edu

Jiawei Li
University of California, Irvine
Irvine, CA, USA
jiawl28@uci.edu

Iftekhar Ahmed
University of California, Irvine
Irvine, CA, USA
iftekha@uci.edu

ABSTRACT

Background: Just-In-Time (JIT) defect prediction models predict if a commit will introduce defects in the future. DeepJIT and CC2Vec are two state-of-the-art JIT Deep Learning (DL) techniques. Usually, defect prediction techniques are evaluated, treating all training data equally. However, data is usually imbalanced not only in terms of the overall class label (e.g., defect and non-defect) but also in terms of characteristics such as *File Count, Edit Count, Multiline Comments, Inward Dependency Sum* etc. Prior research has investigated the impact of class imbalance on prediction technique's performance but not the impact of imbalance of other characteristics. **Aims**: We aim to explore the impact of different commit related characteristic's imbalance on DL defect prediction. **Method**: We investigated different characteristic's impact on the overall performance of DeepJIT and CC2Vec. We also propose a **Si**amese network based **f**ew-sho**t** l**e**arning f**r**amework for JIT defect prediction (**SifterJIT**) combining Siamese network and DeepJIT. **Results**: Our results show that DeepJIT and CC2Vec lose out on the performance by around 20% when trained and tested on imbalanced data. However, SifterJIT can outperform state-of-the-art DL techniques with an average of 8.65% AUC score, 11% precision, and 6% F1-score improvement. **Conclusions**: Our results highlight that dataset imbalanced in terms of commit characteristics can significantly impact prediction performance, and few-shot learning based techniques can help alleviate the situation.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**.

KEYWORDS

Deep learning, defect prediction, few-shot learning, software engineering

ACM Reference Format:
Jiri Gesi, Jiawei Li, and Iftekhar Ahmed. 2021. An Empirical Examination of the Impact of Bias on Just-in-time Defect Prediction. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '21), October 11–15, 2021, Bari, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3475716.3475791

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 [49]. Moreover, bug fixing is effort-intensive; Kim et al. [43] 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 [89], historical data [28, 61], and developers' interaction information [51, 65]. 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 [37] 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 [31, 32, 52, 84] and is the state-of-art in terms of performance. For example, Yang et al. [84] utilized Deep Belief Network, Hoang et al. [31] 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 [32].

All defect prediction techniques, including JIT, relies on the quality of data [8, 15]. 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 [80]. 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 [29], 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 [48]. For vehicle recognition, it is more difficult to detect crashed vehicles than regular vehicles since training data rarely contain crashed vehicles [85].

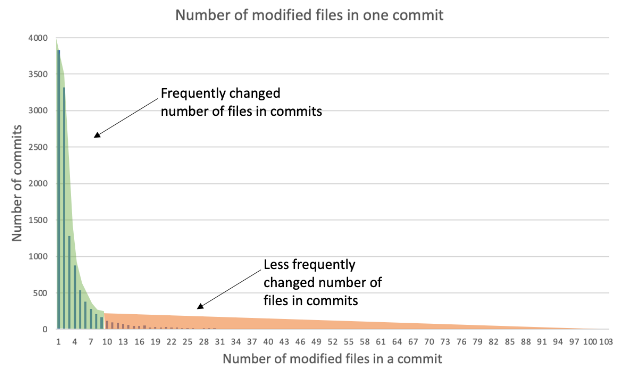


Figure 1: Long-tail distribution of number of changed files in a commit.

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 [79]. 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 [55, 73].

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 shows the frequency along with the number of modified files in each commit from OPENSTACK [2] 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 [53, 68]) 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 describes the related work. Section 3 presents details of our methodology. Section 4 reports the findings. Section 5 places our results in the broader context of work to date and outlines the implications for practitioners and researchers. Section 6 is the threats to validate our results. Section 7 concludes with a summary of the key findings and an outlook on our future work.

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.

2.1 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 maintenance cost [28, 35]. 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 [28, 45, 88]. Researchers have investigated on how to manually design new features or combinations of features to represent defects [56]. Prior research also looked into using DL algorithms to learn features or new representations automatically [31, 32]. Besides these approaches, researchers also explored transfer learning [36], Personalized [34] for defect predictions.

To further reduce the costs of software development by identifying defects as soon as they are introduced, research studies [40, 42, 84] 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 [39] 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 [31]. As a result, JIT defect prediction research has gained much attention in recent years [20, 41, 72].

ML techniques such as Support Vector Machine [23], Random Forest [9] and Nearest-Neighbor [70] 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 [40, 59, 63]. DL techniques have also been adopted in JIT defect prediction [31, 32, 64, 82, 84]. Yang et al. [84] integrated DL in JIT defect prediction by constructing a Deep Belief Network-based approach. Qiao et al. [64] 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 [31, 32] 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 [31] and CC2Vec [32] with respect to performance. DeepJIT [31] is a DL-based JIT defect prediction technique. It trains on the information of both commit message and code change [31]. DeepJIT uses two separate Convolutional Neural Networks (CNN) for feature extraction and concatenation [50]. Using the resulting vector, the output layer computes the probability of a commit being defective. CC2Vec [32] 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.

2.2 Study on Data Bias in Software Engineering

In recent years, with the advent of ML, numerous studies have analyzed the biases within ML related software [16, 17, 74]. This type of biases mainly results from biased ML model training/evaluation processes and data distribution imbalance in the training or testing datasets [6, 12, 16, 17, 27, 74]. In addition, the data labeling, model training, and model evaluation may contribute to building a biased model [8, 57, 75, 77]. Data-driven decisions have the potential to negatively impact already disadvantaged populations [8, 15, 57, 67, 76] as they are relatively less represented in the training data.

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 [11]. 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 [56, 66]. 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 [83]. 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. [66] 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 [89], 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 [21, 38, 47, 54, 71, 79]. To the best of our knowledge, researchers have used the methods mentioned above to mitigate the class imbalance issue [63, 84]. 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.

2.3 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 [7], image classification [46]), 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 [80].

As a representative method of few-shot learning, the concept of Siamese networks was proposed by Bromley et al. [14]. Koch [46] and Neculoiu et al. [62] 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 [58]. Moreover, they proved that Siamese networks are good at learning on a dataset where a small amount of data is available [46, 62, 78].

Siamese network has also been used in defect prediction field, Zhao et al. [87] 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 [26]. 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.

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.

3.1 Prediction Technique Selection

Among many available DL-based JIT defect prediction techniques, we picked DeepJIT [31] and CC2Vec [32] since they are state-of-the-art. These two studies use the same training/testing datasets originally curated by McIntosh et al. [56] and have been widely used in JIT defect prediction [31, 32, 63] literature. In the dataset, McIntosh et al. manually filtered and analyzed commits from two well-known software projects QT [4] and OPENSTACK [3]. The dataset contains 25,150 commits from the QT project and 12,374 commits from the OPENSTACK project. Table 1 presents summary statistics of the dataset.

Table 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%)

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 [63]. We focused on extracting characteristics relevant to a commit since JIT prediction is performed after every new commit. We relied on the study conducted by Motwani et al. [60] 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. [60] 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.

To extract *Edit Count* and *File Count*, we first collected the information (e.g., added code lines, deleted code lines, the names of

Table 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.

Table 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}"

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 [60], 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 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 3.

Prior research identified comments when modified with source can act as a significant feature for defect prediction [42]. 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.

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 [31] and CC2Vec [32] with

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

their original training and testing datasets so that we can compare the characteristics' impact on these two techniques.

DeepJIT [31] 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) [50] 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. [32] 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 [13], 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" [69].

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:

$$Threshold = median[\sum_i \sum_j mean(CVC_{m^i}^{d^j}) + \sum_i \sum_j mean(CVW_{m^i}^{d^j})] \quad (1)$$

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), 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 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 4, and for QT is in Table 5.

Table 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%

Table 5: Training and testing data of QT dataset based on thresholds

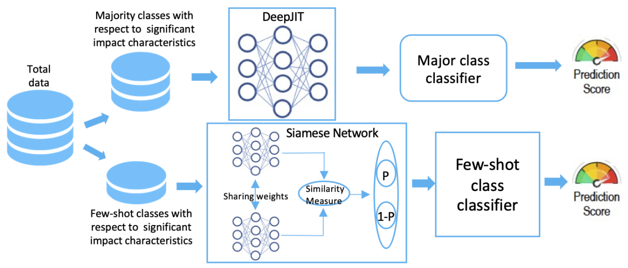
Characteristics	Divide Threshold	Smaller than threshold (train/test)	Bigger than threshold (train/test)
Edit Count	247.35	92.25%/93.38%	7.75%/6.62%
File Count	13.22	94.39%/94.32%	5.61%/5.68%
Multiline Comments	58.13	93.37%/93.81%	6.63%/6.19%
Inward Depen- -dency Sum	71.71	88.23%/86.84%	11.77%/13.16%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

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. First, we divide the training dataset into majority and few-shot classes based on their characteristics (Section 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. [46] and Neculoiu et al. [62] 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.



Total data
Majority classes with respect to significant impact characteristics
DeepJIT
Major class classifier
Prediction Score
Siamese Network
Sharing weights
Similarity Measure
P
1-P
Few-shot class classifier
Prediction Score
Few-shot classes with respect to significant impact characteristics

Figure 2: Overview of SifterJIT

3.5.1 *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 3 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.

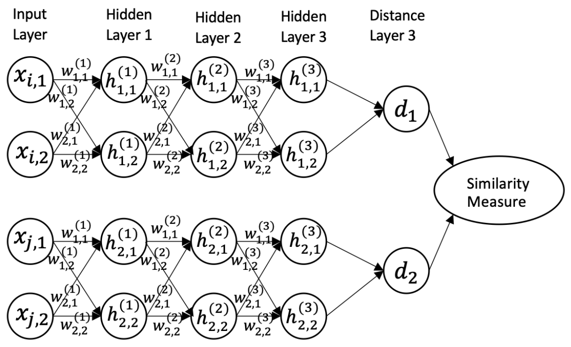


Figure 3: Siamese network with three hidden layers

3.5.2 *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:

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

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

3.5.3 *Loss function.*

: For the Siamese network loss function, we use the most popular Contrast loss function [81]. The loss function is defined as:

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

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.

3.5.4 *SifterJIT Model Training.*

The SifterJIT's base network is similar to DeepJIT [31], 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. [31] 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 [31, 33].

3.5.5 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 [22, 55] 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 [18] 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 [18]. 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 implemented a random oversampling scheme [22]. The random over-sampling technique randomly duplicates examples from the few-shot class and adds them to the training dataset [22]. 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.

3.5.6 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 [10]. 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 [19, 24, 25, 31, 32, 86].

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.*

$$precision = \frac{t_p}{t_p + f_p} \tag{3}$$

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

$$recall = \frac{t_p}{t_p + f_n} \tag{4}$$

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

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{5}$$

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

$$FPR = \frac{f_p}{f_p + t_n} \tag{6}$$

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).

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 6 and in table 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 6 and 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 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 7).

Table 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 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***

Observation: DL defect prediction technique's prediction performance is affected by characteristics such as *File Count, Edit Count, Multiline Comments, Inward Dependency Sum* and *Outward Dependency Sum.*

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 8. Then, we divided training and testing data into two groups using the threshold shown in table 4 for OPENSTACK and table 5 for QT. We call the group with values less than the corresponding threshold as *Majority class* since they 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.

Table 8: The AUC results on all testing data

	OPENSTACK	QT
DeepJIT	75.1	76.8
CC2vec + DeepJIT	80.9	82.2

From Table 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 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 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 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 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, $p<0.119$).

Observation: AUC score for few-shot classes are significantly lower than corresponding majority classes, up to 25.9 for DeepJIT and 19.4 for CC2vec.

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 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.

Observation: All characteristics negatively affect the AUC, however, *Edit Count*, and *Outward Dependency Sum* are the most affecting characteristics for both datasets used in the experiment.

Table 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 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

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 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 [30]. 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 4. From Figure 4 we can observe that SifterJIT outperforms original and oversampling results for most characteristics.

Table 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

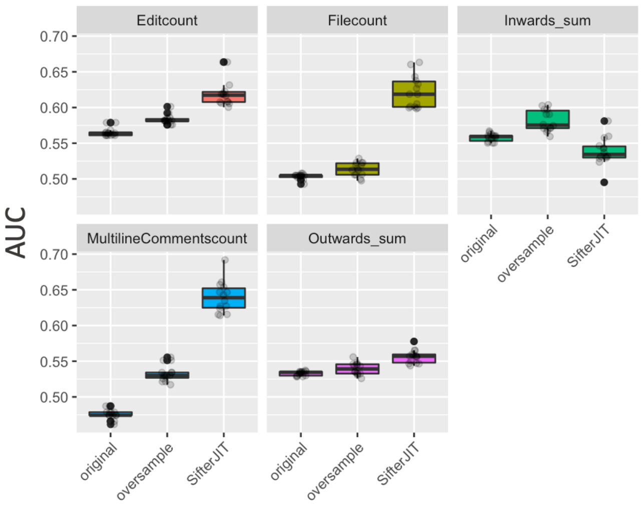


Figure 4: AUC distribution for different characteristics using OPENSTACK

Table 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 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 5 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 11 and Table 12.

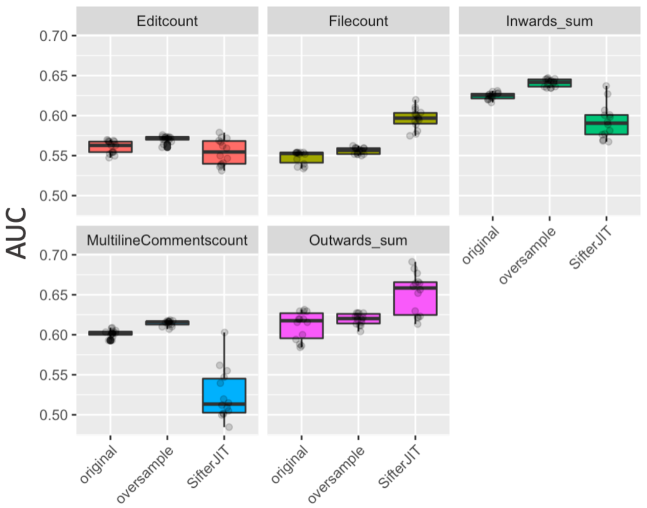


Figure 5: AUC distribution for different characteristics using QT

Observation: SifterJIT significantly improves the original defect prediction AUC score by 11.31% and 5.98% for OPENSTACK few-shot classes and QT few-shot classes, respectively.

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 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 [40, 56, 59], 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 Table10. 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 Table10. 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. Table9 and 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 [44].

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 [1, 5]. 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.

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 [31, 32, 56, 63]. 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 [60]. 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.

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.

REFERENCES

[1] "Ai fairness 360," https://aif360.mybluemix.net/, accessed: 2021-05-1.
[2] "Open stack data set," https://docs.openstack.org/wallaby/?_ga=2.205840979.1124305833.1619313296-1069099767.1617679651, accessed: 2020-07-1.
[3] "Openstack," https://www.openstack.org/, accessed: 2021-05-1.
[4] "Qt," https://www.qt.io/, accessed: 2021-05-1.
[5] "What-if tool," https://pair-code.github.io/what-if-tool/, accessed: 2021-05-1.
[6] 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, 2019, pp. 625–635.

[7] H. Altae-Tran, B. Ramsundar, A. S. Pappu, and V. Pande, "Low data drug discovery with one-shot learning," *ACS central science*, vol. 3, no. 4, pp. 283–293, 2017.

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

[9] 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)*. IEEE, 2016, pp. 214–221.

[10] 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*. ACM, 2007, pp. 11–18.

[11] 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*, 2009, pp. 121–130.

[12] 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*, 2020, pp. 642–653.

[13] J. M. Bland and D. G. Altman, "Multiple significance tests: the bonferroni method," *Bmj*, vol. 310, no. 6973, p. 170, 1995.

[14] 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*, vol. 6, pp. 737–744, 1993.

[15] 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*, 2020, pp. 654–665.

[16] 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)*. IEEE, 2020, pp. 1229–1233.

[17] 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.

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

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

[20] 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.

[21] 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)*, vol. 42, no. 4, pp. 463–484, 2011.

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

[23] 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*, vol. 1. IEEE, 2015, pp. 789–800.

[24] 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*. ACM, 2012, pp. 171–180.

[25] 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*. ACM, 2011, pp. 83–92.

[26] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Reflections on the nasa mdp data sets," *IET software*, vol. 6, no. 6, pp. 549–558, 2012.

[27] 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*, 2020, pp. 851–862.

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

[29] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on knowledge and data engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.

[30] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 39–48.

[31] 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)*. IEEE, 2019, pp. 34–45.

[32] 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*, 2020, pp. 518–529.

[33] X. Huo, M. Li, Z.-H. Zhou *et al.*, "Learning unified features from natural and programming languages for locating buggy source code." in *IJCAI*, vol. 16, 2016, pp. 1606–1612.

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

[35] 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*, 2014, pp. 414–423.

[36] 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*, vol. 21, no. 5, pp. 2072–2106, 2016.

[37] 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*. IEEE, 2010, pp. 1–10.

[38] 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)*. IEEE, 2007, pp. 196–204.

[39] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 5. IEEE, 2016, pp. 33–45.

[40] 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*, vol. 39, no. 6, pp. 757–773, 2012.

[41] 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)*. IEEE, 2020, pp. 1336–1339.

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

[43] 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*. ACM, 2006, pp. 173–174.

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

[45] 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)*. IEEE, 2007, pp. 489–498.

[46] G. Koch, R. Zemel, and R. Salakhutdinov, "Siamese neural networks for one-shot image recognition," in *ICML deep learning workshop*, vol. 2. Lille, 2015.

[47] S. Kotsiantis, D. Kanellopoulos, P. Pintelas *et al.*, "Handling imbalanced datasets: A review," *GESTS International Transactions on Computer Science and Engineering*, vol. 30, no. 1, pp. 25–36, 2006.

[48] 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*, vol. 33, no. 10, pp. 1962–1977, 2011.

[49] 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*. IEEE, 2012, pp. 3–13.

[50] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[51] 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*, 2011, pp. 311–321.

[52] 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)*. IEEE, 2017, pp. 318–328.

[53] 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*. IEEE, 2014, pp. 426–435.

[54] 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*, vol. 250, pp. 113–141, 2013.

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

[56] 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*, vol. 44, no. 5, pp. 412–428, 2017.

[57] 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.

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

[59] A. Mockus and D. M. Weiss, “Predicting risk of software changes,” *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[60] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, “Do automated program repair techniques repair hard and important bugs?” *Empirical Software Engineering*, vol. 23, no. 5, pp. 2901–2947, 2018.

[61] 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*. IEEE, 2010, pp. 309–318.

[62] 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*, 2016, pp. 148–157.

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

[64] L. Qiao and Y. Wang, “Effort-aware and just-in-time defect prediction with neural network,” *PloS one*, vol. 14, no. 2, p. e0211359, 2019.

[65] 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*. ACM, 2011, pp. 491–500.

[66] 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*, 2013, pp. 147–157.

[67] 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*, vol. 169, no. 12, pp. 866–872, 2018.

[68] 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*. Springer, 2014, pp. 637–648.

[69] 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*, vol. 177, 2006.

[70] 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*, vol. 30, no. 5, pp. 969–980, 2015.

[71] 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*, vol. 1, no. 5, pp. 448–459, 2011.

[72] 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)*. IEEE, 2020, pp. 554–565.

[73] 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*, vol. 2. IEEE, 2015, pp. 99–108.

[74] 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*, 2020, pp. 1122–1134.

[75] 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)*. IEEE, 2017, pp. 401–416.

[76] M. Veale and R. Binns, “Fairer machine learning in the real world: Mitigating discrimination without collecting sensitive data,” *Big Data & Society*, vol. 4, no. 2, p. 2053951717743530, 2017.

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

[78] 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*, vol. 19, no. 1, pp. 230–241, 2017.

[79] S. Wang and X. Yao, “Using class imbalance learning for software defect prediction,” *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[80] 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)*, vol. 53, no. 3, pp. 1–34, 2020.

[81] 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*, 2006, pp. 1473–1480.

[82] 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*, vol. 46, no. 11, pp. 1155–1175, 2018.

[83] 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*, 2011, pp. 15–25.

[84] 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*. IEEE, 2015, pp. 17–26.

[85] 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.

[86] 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*, 2014, pp. 182–191.

[87] 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*, vol. 7, pp. 7663–7677, 2018.

[88] 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*, 2009, pp. 91–100.

[89] T. Zimmermann, N. Nagappan, and A. Zeller, “Predicting bugs from history,” in *Software evolution*. Springer, 2008, pp. 69–88.