Simple or Complex? Together for a More Accurate Just-In-Time Defect Predictor

Xin Zhou, DongGyun Han, and David Lo
School of Computing and Information Systems, Singapore Management University
Singapore
xinzhou.2020@phdcs.smu.edu.sg,{dhan,davidlo}@smu.edu.sg

ABSTRACT

Just-In-Time (JIT) defect prediction aims to automatically predict whether a commit is defective or not, and has been widely studied in recent years. In general, most studies can be classified into two categories: 1) simple models using traditional machine learning classifiers with hand-crafted features, and 2) complex models using deep learning techniques to automatically extract features. Hand-crafted features used by simple models are based on expert knowledge but may not fully represent the semantic meaning of the commits. On the other hand, deep learning-based features used by complex models represent the semantic meaning of commits but may not reflect useful expert knowledge. Simple models and complex models seem complementary to each other to some extent. To utilize the advantages of both simple and complex models, we propose a combined model namely SimCom by fusing the prediction scores of one simple and one complex model. The experimental results show that our approach can significantly outperform the state-of-the-art by 6.0–18.1%. In addition, our experimental results confirm that the simple model and complex model are complementary to each other.

ACM Reference Format:
Xin Zhou, DongGyun Han, and David Lo. 2022. Simple or Complex? Together for a More Accurate Just-In-Time Defect Predictor. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524610.3527910

1 INTRODUCTION

Modern software development tends to release software in a short period [53, 58], although developers often have limited testing resources [33, 53]. Such limited testing resources and tight development schedules often lead to introducing software defects [10]. To address the challenge, many defect prediction approaches have been proposed to help developers to narrow down the testing and debugging scope to software components that are highly likely to contain defects [15, 36]. The Just-in-Time (JIT) defect prediction approach is a specific type of defect prediction approach that can identify the possible defective code changes as soon as they are submitted, which can avoid the defects being merged to the codebase. Besides, JIT defect prediction approaches can give fine-grained predictions at the change level, while traditional defect prediction approaches only give coarse predictions at the file level or module level [15, 72]. Moreover, when a commit is submitted for a review, developers need to comprehend it to see whether to integrate it into the code base or not. JIT defect prediction approaches can help to flag cases that require developers' more scrutiny and thus guide the comprehension process to focus on the more risky cases.

In the past decades, JIT defect prediction has attracted many researchers and many approaches have been proposed [10, 12, 29, 30, 37, 44, 53, 75–77, 79, 81]. In general, most work can be classified into two categories: 1) models using traditional machine learning (ML) classifiers with hand-crafted features, and 2) models using deep learning (DL) techniques to automatically extract features. Hereafter, we refer to traditional machine learning as ML and deep learning as DL for simplicity. Compared to the models using traditional machine learning, deep learning-based models usually have a larger number of parameters to train. Thus, usually, DL-based approaches require much more training and evaluation time than the approaches using ML classifiers with hand-crafted features. In this paper, we regard deep learning-based models as **complex models** (many parameters to train, slower). Also, we consider the models using ML classifiers with hand-crafted features as **simple models** (small number of parameters to train; much faster than DL-based techniques). The majority of existing work are simple models, which rely heavily on hand-crafted change-level features [10, 12, 37, 44, 53, 75–77, 79, 81]. Simple models feed the features to a traditional machine learning classifier (e.g. Logistic Regression) to predict if code changes are defective. On the other hand, the complex models usually extract the features automatically from the content of commits (i.e. the commit logs and code changes) by using deep learning techniques.

There exists a saga of simple vs. complex in the JIT defect prediction task (and beyond) [19, 29, 46, 81]. Even though complex models are slower than simple models, complex models usually show their superiority over the simple models in predictive power. For instance, DeepJIT (a complex model) outperforms prior simple models by 7–9% in terms of Area Under the Receiver Operating Characteristic Curve (AUC-ROC) [29]. However, recently, Zeng et al. proposed a simple model namely LApredict that achieved 2.6% improvements over complex models in AUC-ROC [81].

Simple models use a set of hand-crafted features to represent a commit. These features are based on expert knowledge and describe some properties of commits such as code change size, code change diffusion, and history of code change. However, those hand-crafted features may not fully represent the semantic meaning of the actual code changes in commits. For instance, two commits with different

ICPC '22, May 16–17, 2022, Virtual Event, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9298-3/22/05…$15.00
https://doi.org/10.1145/3524610.3527910

purposes and defectiveness labels may still have the same hand-crafted features (e.g. code change size) [29].

On the other hand, complex models use features that are automatically extracted from the raw contents of commits via DL techniques. These features represent the semantic meaning of commits [29], which are proven useful in various software engineering tasks including JIT defect prediction [27, 29, 59, 74]. However, these automatically extracted features may not leverage useful expert knowledge. For instance, code files changed by many developers in the past may include defects [48]. However, the content in a single commit does not explicitly contain information about the number of developers who have changed the modified files of this commit in the past.

In terms of the features, the simple model and complex model seem complementary to each other, which motivates us to combine these two models to build a better JIT defect prediction model. A straightforward idea is to simply put all hand-crafted features as an extra input to a complex model. However, these two features have different properties: hand-crafted features are a list of numbers that each number has a clear meaning while DL features are vectors whose elements are not mutually exclusive and do not have clear interpretations [22]. Besides, the commonly used classifiers for hand-crafted features (e.g. Logistic Regression) and DL features (e.g. the fully connected layers) are also different. Thus, to better make use of two kinds of features, one potential approach is to assign the most suitable classifier to each of two features: separately train a simple model with hand-crafted features and a complex model with DL features.

In this paper, we propose an approach that combines a simple model and a complex model. Our approach leverages our own simple and complex models that are inspired by previous work [29, 76, 81]. These simple and complex models are combined to provide a prediction in the manner of late fusion [24, 68] (i.e. fusing the prediction scores from different models).

Although previous work [29, 30, 81] use Area Under the Receiver Operating Characteristic Curve (AUC-ROC) to evaluate the performance, ROC curves can present an overly optimistic view of performance in a largely skewed dataset [17]. Alternatively, Precision-Recall (PR) curves are more informative than ROC curves when evaluating binary classifiers on skewed datasets [65]. In JIT defect prediction, usually, the number of defective commits is smaller than the number of clean commits so that many JIT defect prediction datasets are skewed datasets. For instance, in Zeng et al.'s work [81], the QT dataset contains about 20,000 clean commits, while it only contains about 3,500 defective commits. To ensure fairness and validity, we use AUC-ROC, AUC-PR, and F1-scores, to evaluate the performance of each approach.

The experimental results show that our approach can significantly outperform the state-of-the-art by 6.0%, 12.4%, and 18.1% in terms of AUC-ROC, AUC-PR, and F1-score respectively. Besides, our approach (the combined model) shows better performance in a diverse set of projects than the simple model only or the complex model only.

Our main contributions can be summarized as follows:

- We propose an approach that combines a simple model and a complex model. To the best of our knowledge, it is the first approach to combine the simple and complex models to build an effective approach for JIT defect prediction.
- We conduct a study to evaluate our proposed approach on a large and diverse dataset with a total of 94,818 commits. The results show that our approach outperforms the baselines by a large margin. To better understand and analyze our approach, we also carry out further experiments including an ablation study.
- To ensure the validity of our experimental results, we choose to add a suitable metric (i.e. AUC-PR) to evaluate the performance in the JIT defect prediction task whose datasets usually have a (large) skew in the class distribution (i.e. clean or defective).

The rest of the paper is organized as follows. Section II provides background information on existing JIT defect prediction approaches and model fusion techniques. Section III introduces our proposed approach namely SimCom. Section IV describes our experimental settings and research questions. Section V reports our experimental results. Section VI discusses our findings. Section VII presents the related work. Section VIII concludes the paper with future work.

2 BACKGROUND

2.1 Just-in-time Defect Prediction

The Just-in-Time (JIT) defect prediction approach is to predict if a commit is defective or not. We briefly introduce the well-known simple and complex JIT defect prediction models.

Simple models. Simple models rely heavily on hand-crafted change-level features [10, 12, 37, 44, 53, 75–77, 79, 81]. The simple models feed the hand-crafted features to a machine learning classifier (e.g. Logistic Regression, Decision Tree, and Random Forest) to predict if commits are defective. LR-JIT is a classic simple model that is proposed by Kamei et al. [37], which integrates 14 commit-level hand-crafted features with the logistic regression model to predict if a commit is defective or not. These 14 features describe the properties of code changes, such as the size, diffusion, history, and author experience of code changes. LR-JIT has been widely adopted as an evaluation baseline for many previous JIT defect prediction studies [77, 78, 81]. Yang et al. proposed DBN-JIT [77] which uses Deep Belief Network (DBN) to extract high-level information from the commit-level defect prediction features mentioned above. Although DBN-JIT adopted Deep Learning techniques (i.e. Deep Belief Network), its goal is still to make better use of hand-crafted features and do not extract features automatically from the contents of commits. Thus, we categorize it as a simple model (i.e. relying heavily on hand-crafted features) in the context of this paper.

Complex models. Recently, two complex models are proposed, namely DeepJIT [29] and CC2Vec [30]. DeepJIT uses the convolutional neural network for text (textCNN) [40] to automatically extract features from the content of code changes and commit logs. CC2Vec learns the distributed representation of code changes guided by their log messages. Hoang et al. integrated the output of CC2Vec with DeepJIT and outperformed the original DeepJIT [30]. However, Zeng et al. argued that in their extended JIT defect prediction datasets, CC2Vec cannot outperform DeepJIT consistently [81].

State-of-the-art. LApredict is a simple model that only makes use of *the number of added lines* feature with the logistic regression classifier. In other words, LApredict gives defectiveness prediction only based on the number of code lines added in a commit: the more lines of code are added in a commit, the higher is its probability of being defective. LAPredict [81] has outperformed the other existing JIT defect prediction approaches regardless of its simpleness. Surprisingly, LAPredict even outperforms complex DL approaches by 2.6% in terms of the AUC-ROC score on average.

2.2 Model Fusion

Model fusion techniques are widely used in tasks that have multiple data modalities such as object detection and video analysis [18, 24, 45, 68, 71]. Generally, there are two typical fusion methods, namely early fusion and late fusion. Early fusion is carried out at the feature level: various features of different modalities are concatenated into one big vector for classification [24, 68]. Late fusion (also called decision level fusion) is to build several models independently for different modalities (or modality groups) and do a fusion at a decision-making stage (on the prediction scores) [24, 68]. Late fusion has different rules to determine how to finally combine each of the outputs of the independently trained models, e.g. product, sum, average, or weighted average of the outputs of all models [80]. Though late fusion seems similar to ensemble learning [64], they have differences between fused models. Late fusion usually aggregate heterogeneous models that deal with different inputs (modalities) while the ensemble learning models usually aggregate homogeneous models that are fed with the identical inputs.

JIT defect prediction task can also be regarded as a multi-modal task: hand-crafted features, commit logs, and code changes can be seen as three different aspects (modalities) of commits. These three modalities have quite different properties: hand-crafted features are numerical features and each feature has a clear meaning; commit logs are natural language texts, and the code changes are written in a programming language. In this case, it is natural to make use of all of these three modalities by using a fusion technique.

3 SIMCOM

In this section, we present the details of our proposed approach: the **Sim**ple and **Com**plex (SimCom) model. Figure 1 presents the overall framework of our proposed approach SimCom. The framework mainly contains three modules: a simple model module, a complex model module, and a model fusion module.

3.1 Simple Model: Sim

In this work, we call our simple model module as *Sim* for short. Following the well-known simple model, LR-JIT [37], we use the same 14 hand-crafted features (as presented in Table 1) in LR-JIT, which describe the properties of commits. We choose the features of LR-JIT because LR-JIT considers a comprehensive set of hand-made features based on many prior works [14, 23, 25, 26, 42, 48, 53, 54, 57, 61]. The effectiveness of the simple model will be affected by the set of hand-made features we choose. For the ML classifier, we choose Random Forest (RF) [7], which is one of the most popular ML classifiers. It is a fast and robust classifier that is capable of identifying non-linear patterns in both continuous and categorical features [13]. In this model, we directly feed 14 hand-crafted features of commits into the RF and the RF gives predictions on the labels (i.e. defective or clean) of commits. We use the implementation of RF in the Scikit-Learn package [8] with the default parameters of the package.

3.2 Complex Model: Com

In this work, we call our complex model module as *Com* for short. Following previous work, in the complex model module, we use textCNN [40] as the basic feature extractor to automatically extract features from the contents of commits. As a commit consists of a commit log and a set of code changes, we first separately extract features from the commit log and code changes respectively (i.e. a commit log vector and a code change vector). Then, we concatenate the two vectors to get the final feature for the whole commit (i.e. a commit vector). After getting the commit vector, we feed the commit vector into a fully connected layer (which acts as a classifier) to predict whether a commit is defective or not. In general, this complex model has the similar framework as the prior work DeepJIT. We follow the framework of DeepJIT since this framework can hierarchically encode structures in commits (i.e. line→hunk→file→commit), which is used in prior DL-based JIT defect predictors [29, 30] and is proven useful. Though the framework is similar, we make changes to reflect the code changes in a better way. We will explain the details in Section 3.2.3.

3.2.1 *textCNN*.

The textCNN, as the basic feature extractor, can aggregate a sequence of token embeddings into a single vector that contains the semantic meaning of the input sequence [40].

Given a sequence of tokens denoted as $M = [t_1, t_2, ..., t_{|M|}]$, each token t_i has its own embedding $w_i \in R^d$ where d is the dimension of token embeddings. $w_{i:j}$ stands for the concatenation of token embeddings from position i to j in the sequence. To fuse the embeddings of tokens in a sequence, a filter $f \in R^{k \times d}$ is utilized to a window of k files to compute a new feature:

$$x_i = \alpha(f \cdot w_{i:i+k-1} + b_i)$$

where $\cdot$ is a sum of element-wise product, $\alpha(.)$ is a non-linear activation function, and $b_i \in R$ is the bias value. The filter f is applied to every k-tokens in a sequence. The generated features are

Table 1: Widely used 14 hand-crafted features for code changes

Name	Description
NS	The number of modified subsystems [53]
ND	The number of modified directories [53]
NF	The number of modified files [57]
Entropy	Distribution of modified code across each file [14, 26]
LA	Lines of code added [54, 56]
LD	Lines of code deleted [54, 56]
LT	Lines of code in a file before the change [42]
FIX	Whether or not the change is a defect fix [25, 61]
NDEV	The number of developers that changed the modified files [48]
AGE	The average time interval between the last and current change [23]
NUC	The number of unique changes to the modified files [14, 26]
EXP	Developer experience [53]
REXP	Recent developer experience [53]
SEXP	Developer experience on a subsystem [53]

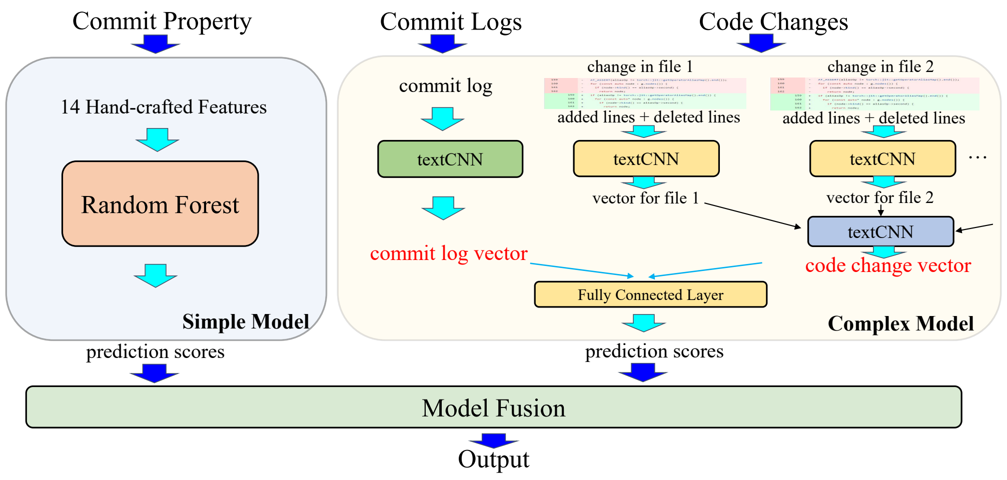

Figure 1: The framework of SimCom

then concatenated to form a vector X such that:

$$X = [x_1, x_2, ..., x_{|M|-k+1}]$$

Following prior work [29, 40], for each filter, we then use a max-pooling layer [22] to process the feature vector X to obtain the highest value:

$$c = \max_{1 \leq i \leq |M|-k+1} x_i$$

where c is the feature of the input corresponding to a particular filter. Therefore, each filter extracts a feature c from the input M. TextCNN uses multiple filters (with varying window sizes) to get multiple features [40]. Features from all filters are concatenated to form Z_M, a single vector that represents the whole sequence.

In brief, we can simply think of textCNN as a feature extractor that turns a sequence of tokens into a single vector:

$$Z_M = textCNN(M)$$

where M is the input sequence and Z_M is the output feature (i.e. a vector representing the input sequence).

3.2.2 ***Feature extraction on commit logs.*** Commit logs are mainly natural language (NL) sentences that summarize the purposes of those commits by authors. As a commit log m is a sequence of NL tokens, textCNN can be used without any pre-processing to extract a single vector from the commit log. We call this single vector the commit log vector (i.e. Z_m).

3.2.3 ***Feature extraction on code changes.*** Unlike a commit log, code changes in a commit are not a simple sequence but a list of sequences: code changes spread over different code files in a project because a commit may modify several files simultaneously.

For changes in each code file, we first concatenate all added and removed lines separately. Then we concatenate the added lines and the removed lines using special headers. The concatenated added and removed lines in each file are then processed to be one single sequence to represent changes in a file. Figure 2 presents a simple example of our processing of code changes. In this example, the red lines are removed code lines in a code file and the green lines are added code lines.

The motivation of this processing comes from the following observation. In the previous work [29], there is no difference in the ways of processing the added code lines and removed code lines (i.e. no special headers as indicators). We highlight the added (deleted) parts by adding two special headers indicating the property of the following code lines (i.e. added or deleted). Adding too many special tokens may hurt the model's performance. To avoid adding special headers to every changed line, we first concatenate the added (deleted) lines together and then only add two special headers.

As the added and removed lines in each file are processed to be one single sequence to represent changes in a file, we then feed the sequence into a textCNN to get the vector for changes in each file. One important step for effective JIT defect prediction is to find the most suspicious parts of commits. This process is very similar to extracting important keywords/phrases from the text to do classification [40, 82]. TextCNN is an efficient deep learning sequence feature extractor, which is good at capturing the key segments of an input text [82] (e.g. most defective code token sequences in our case). Thus, we use textCNN to extract features from code changes in a code file.

To fuse the vectors (features) from different code files, we use another textCNN to aggregate those vectors into a single vector Z_C representing the whole code changes (i.e. the code change vector).

3.2.4 ***Feature concatenation and prediction.*** After getting the vectors for the commit log and the code changes, we concatenate these two vectors to generate a final feature representation (i.e. Z) representing the commit:

$$Z = Z_m \oplus Z_C$$

where $\oplus$ is the concatenation operator.

Finally, the vector Z is fed into a widely used DL classifier, a fully-connected (FC) layer [29, 30], to get the prediction score.

$$y = \text{sigmoid}(\alpha(w_h \cdot Z + b_h))$$

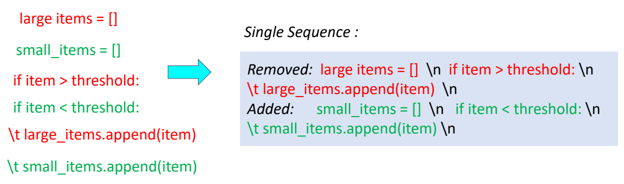
large items = []
small_items = []
if item > threshold:
if item < threshold:
\t large_items.append(item)
\t small_items.append(item)
Single Sequence :
Removed: large items = [] \n if item > threshold: \n \t large_items.append(item) \n
Added: small_items = [] \n if item < threshold: \n \t small_items.append(item) \n

Figure 2: An example about the processing of the code changes in a file into a single sequence. The red lines are removed code lines and the green lines are added code lines. "Removed:" and "Added:" are two special headers indicating a code line is removed or added. " $\backslash n$ " is a separator between different code lines.

where · is a dot product, w_h is a weight matrix of the FC layer, b_h is the bias term, $\alpha(.)$ is a non-linear activation function, and sigmoid(.) is a function to do normalization on the prediction scores.

3.3 Model Fusion

Our proposed approach, SimCom, is a combined model of one simple model (i.e. *Sim*) and one complex model (i.e. *Com*) by late fusion. Each model will give a prediction score for a commit and these scores are combined in the model fusion module to yield a final prediction score. Common late fusion methods include the average, maximum, sum, and weighted average of scores [80]. In our preliminary experiment, we found that taking the average of two models achieves the best performance. Thus, we simply calculate the average of two prediction scores from the simple and complex models as the final prediction score.

As JIT defect prediction is a binary classification task, the learning objective is to minimize the binary cross-entropy loss [55] between the final prediction scores and the ground truths (i.e. defectiveness labels).

4 EXPERIMENTAL SETTING

4.1 Datasets

In this paper, we select the datasets collected by Zeng et al. [81] for the following reasons. First, to make a fair comparison, we choose to use the same training and testing datasets with the state-of-the-art [81]. Second, we would like to carry out experiments on high-quality JIT defect prediction datasets. JIT defect prediction datasets are mainly collected by utilizing the SZZ algorithm [29, 37, 75]. When collecting the datasets, Zeng et al. paid attention to two known issues of the SZZ algorithm (i.e. the false label issue [39, 49] and authentication latency issue [10, 69]) and followed prior works [10, 39, 49] to reduce the impacts of the known issues. Finally, the dataset of Zeng et al. is a diverse set of real-world projects. They collect all code commits in six popular projects over the last 10 years.

Table 2 shows the statistics of six studied datasets including the project name and the number of commits in the training and test set respectively. For the columns, the "Defect" presents the number of commits that are defective in the dataset. The "Clean" column stands for the number of commits that are clean (not defective).

4.2 Evaluation metrics

We use three widely-used evaluation metrics to evaluate the performance of our approach and baselines.

AUC-ROC: Receiver Operator Characteristic (ROC) curves are used in the binary classification tasks, which show how the number of correctly classified positive examples varies with the number of wrongly classified negative examples [60]. The Area Under the ROC Curves (AUC-ROC) is commonly used to present the performance of approaches in binary classification tasks in software engineering [29, 30, 43, 50–52, 63, 81]. Following the previous work in JIT defect prediction [29, 30, 81], we adopt this widely-used AUC-ROC score to evaluate the performance of models. Besides, the AUC-ROC score does not need one to manually set a threshold [70].

AUC-PR: ROC curves can present an overly optimistic view of the performance of a model if there is a large skew in the class distribution [17]. Precision-Recall (PR) curves, often used in information retrieval [47], have been used as an alternative to ROC curves for tasks with a large skew in the class distribution[6, 9, 16, 17, 21]. Besides, Saito and Rehmsmeier found that PR curves are more informative than ROC curves when evaluating binary classifiers on imbalanced datasets [65]. As shown in Table 2, all of the six datasets are imbalanced datasets. For instance, in the test set of Gerrit, the number of clean commits is 13.88 times bigger than the number of defective commits. Thus, we adopt the Area Under the PR Curves (AUC-PR) as an evaluation metric in this study. Please note that the AUC-PR score also does not need to set a threshold manually.

F1-scores: AUC-ROC and AUC-PR consider prediction scores rather than predicted classes as inputs. Both metrics focus on ranking predictions, rather than the correctness of predicted classes. Specifically, both metrics compare the prediction scores assigned to positive instances (in our case, defective commits) versus those assigned to negative instances (in our case, clean commits). However, in practice, end-users may care more about the correctness of predicted classes.

In the case of JIT defect predictions, at any point in time, the tool needs to make a just-in-time prediction if a commit is defective or clean. At a particular point in time, there may be only one or a few commits being made to a version control system, and thus ranking them may not be helpful. Users of a JIT defect prediction tool may want to know how many correct just-in-time predictions such a tool makes.

To measure the ability of the tool to make correct just-in-time predictions while considering the imbalanced nature of the data, in this work, we also consider the F1-score. F1-score is a popular metric

Table 2: Datasets statistics

Statistics	Training set		Test set		#lines
	Clean	Defect	Clean	Defect	10%
QT	16,242	2,887	4,088	695	146
Openstack	13,153	5,052	3,577	875	209
JDT	1,342	1,281	335	321	200
Platform	5,477	3,350	1,501	706	121
Gerrit	10,348	1,593	2,798	188	120
GO	8,530	6,677	2,334	1,468	197

used in many prior software engineering studies [5, 20, 32, 62, 67], and is defined as the harmonic mean of precision and recall: F1-score = $\frac{2 \times Recall \times Precision}{Recall + Precision}$. Precision and recall, in turn, are defined based on the number of true positives (TP), false positives (FP), and false negatives (FN). If a defective commit is classified as defective, it is a true positive (TP), otherwise false negative (FN). If a clean commit is mistakenly classified as defective, it is a false positive (FP). Precision is the ratio of correctly predicted defective commits to all commits predicted as defective (i.e. Precision = $\frac{TP}{TP+FP}$). Recall is the ratio of the number of correctly predicted defective commits to the actual number of defective commits (i.e. Recall = $\frac{TP}{TP+FN}$).

4.3 Research Questions

RQ1: How effective is SimCom? In this RQ, we aim to explore the effectiveness of our proposed approach SimCom in JIT defect prediction. To answer this RQ, we carry out experiments by adopting typical baselines and different experimental settings to show the effectiveness of SimCom. To answer the RQ1, we set LR-JIT, DeepJIT, and LApredict (the state-of-the-art) as the baselines and compare the experimental results. We do experiments in two different setups (i.e. Setup I and Setup II).

Setup I: Entire Datasets. In Setup I, each approach is trained on the training sets and evaluated on the corresponding entire test sets. Studied JIT defect prediction approaches are trained to learn the relation between the inputs (i.e. hand-crafted features or contents in commits) and the defectiveness labels by applying data in the training sets. Then, the trained models are evaluated on hold-out test sets to show their predictive power.

Setup II: Datasets Excluding Large Commits. Wan et al. [73] surveyed practitioners to discuss the drawbacks of defect prediction tools. Some practitioners are unwilling to adopt defect prediction tools because they think the tools report nothing new. It is well-known that commits that have large numbers of deleted or added lines are more likely defective [34, 57]. Thus, it is necessary to explore the effectiveness of our approach in finding *surprising* defects (i.e. defects hidden in the small or medium size commits). Besides, many contribution guidelines of open-source software already advise developers to make small changes in a commit [1–3] because large code changes may take a very long time to review and test. There will be more and more small commits in the future.

Inspired by the observations above, we aim to investigate the effectiveness of SimCom on the small or medium size commits by using datasets that *exclude large commits*. To carry out experiments on Setup II, we drop the top 10% of large commits (i.e. top 10% commits that have the largest number of modified lines) in each dataset. As shown in Table 2, the last column presents the thresholds of the top 10% large commits for six datasets. For instance, we drop the commits with code change sizes larger than 209 code lines in the dataset of the OpenStack project. To better understand how the models' performance varies when dropping different ratio of large commits, we conduct experiments by dropping top 20%, 30%, 40%, and 50% of large commits. Please note that we drop the large commits in the entire dataset (i.e. the training, validation, and testing data.)

RQ2: How do the component models of SimCom perform? To answer this RQ, we first employ an ablation study to analyze the contribution of each component model (i.e. the simple and complex model). We use prediction scores from each component model and evaluate the predictive power via AUC-ROC, AUC-PR, and F1-scores.

RQ3: How do the different model fusion methods impact the performance? To answer this RQ, we explore the model performance when applying the early fusion method. Besides, we also do experiments on SimCom variants using different late fusion strategies.

4.4 Train-Test Pipelines

We follow the same data partitioning of the LAPredict work [81], i.e. the earlier 80% data are the training set and the later 20% data are the testing set. We use the same data split for all methods, including the three baseline methods. As our complex model is a DL-based model which has many hyper-parameters, we set aside 5% data from training data as the fixed validation set to tune the hyper-parameters of the complex model. Therefore, the data split ratio for SimCom is 75%, 5%, and 20% for training, validation, and testing respectively. We use the original hyper-parameters for the baselines. Since we do not need to tune the hyper-parameters for the baselines, we use the whole training set (80% data) to train the baseline approaches. Please note that all approaches are evaluated on the same hold-out test sets in the manner of fair comparison.

For the training of baselines, we use scripts in the replication packages and use the default hyper-parameters in the packages.

For SimCom, we separatedly train our simple model (i.e. *Sim*) and our complex model (i.e. *Com*). For training of the simple model (i.e. Sim), we use the default parameters of Random Forest (RF) in the Scikit-Learn package. As shown in Table 2, the experiment datasets are imbalanced: the number of defective commits is smaller than clean commits. The imbalance may cause performance degradation of the ML models if it is not handled properly [35, 37, 38]. To mitigate the class imbalance issue for our simple model, we follow the prior work [37] to use an undersampling approach for our training data. We randomly delete the majority class instances (i.e. clean commits in training data) until the majority class drops to the same level as the minority class (i.e. defective commits). Please note that we do not undersample the test data.

For training of the complex model (i.e. Com), we set the learning rate as $5e^{-5}$ and the batch size as 64 after tuning them in the validation. For other hyper-parameters, we set the size of the fully-connected layer as 512, the number of epoch as 30, and the dropout rate as 0.5. These hyper-parameter settings are commonly used in prior work [28, 29, 31, 66]. We use the Adam optimizer [41] to update model parameters. To address the class imbalance issue for our complex model, we use the same loss function as the previous work [29].

For validation, we tune the hyper-parameters of our complex model by observing the performance of the trained model on the validation set. Specifically, we tune the hyper-parameters using a grid search procedure with the following set of parameters and their possible values: the learning rate is in $\{1e^{-5}, 5e^{-5}, 1e^{-4}, 2e^{-4}\}$ and the batch size is in $\{16, 32, 64, 128\}$. We select the combination of hyper-parameters that lead to the best performance on the validation set. The best-performing model in the validation is used for

Table 3: The performance of SimCom and baselines in Setup I with respect to AUC-ROC, AUC-PR, and F1 scores

Projects	Evaluation Metrics	Traditional Machine Learning JIT Models Only			Deep Learning JIT Models Only		Ensemble on Simple	Traditional ML + DL JIT Models
		LR-JIT	DBN-JIT	LApredict	DeepJIT	CC2Vec	XGBoost	SimCom (Improve. %)
QT	AUC-ROC	0.694	0.666	0.744	0.694	0.694	0.725	**0.771** (+3.60%)
	AUC-PR	0.289	0.249	0.319	0.280	0.279	0.308	**0.370** (+16.0%)
	F1 Score	0.337	0.343	0.011	0.332	0.283	0.235	**0.402** (+17.2%)
OpenStack	AUC-ROC	0.742	0.729	0.749	0.713	0.723	0.730	**0.786** (+4.94%)
	AUC-PR	0.437	0.412	0.413	0.396	0.394	0.410	**0.474** (+8.47%)
	F1 Score	0.430	0.432	0.132	0.433	0.400	0.389	**0.491** (+13.4%)
JDT	AUC-ROC	0.682	0.587	0.677	0.670	0.665	0.663	**0.716** (+4.99%)
	AUC-PR	0.655	0.584	0.644	0.644	0.640	0.643	**0.694** (+5.95%)
	F1 Score	0.623	0.533	0.425	0.623	0.631	0.618	**0.645** (+2.22%)
Platform	AUC-ROC	0.646	0.697	0.747	0.771	0.761	0.784	**0.829** (+5.74%)
	AUC-PR	0.439	0.506	0.533	0.574	0.559	0.575	**0.657** (+14.3%)
	F1 Score	0.450	0.533	0.148	0.518	0.563	0.563	**0.646** (+14.7%)
Gerrit	AUC-ROC	0.682	0.671	0.750	0.703	0.699	0.745	**0.756** (+0.80%)
	AUC-PR	0.174	0.171	**0.184**	0.135	0.134	0.158	0.178 (-3.27%)
	F1 Score	0.158	0.197	0.093	0.155	0.167	0.090	**0.270** (+37.1%)
GO	AUC-ROC	0.675	0.674	0.683	0.689	0.692	0.728	**0.778** (+6.87%)
	AUC-PR	0.546	0.549	0.549	0.569	0.577	0.625	**0.681** (+8.96%)
	F1 Score	0.558	0.548	0.051	0.559	0.564	0.600	**0.640** (+6.67%)
Average	AUC-ROC	0.687	0.671	0.725	0.707	0.706	0.729	**0.773** (+6.04%)
	AUC-PR	0.423	0.412	0.440	0.433	0.431	0.453	**0.509** (+12.4%)
	F1 Score	0.426	0.433	0.143	0.437	0.435	0.416	**0.516** (+18.1%)

evaluation on the test sets. For our simple model, we directly use the default parameters of Random Forest and do not tune them.

For testing, the evaluation is carried out on a hold-out test set which is the test set of each dataset. When testing in Setup I (using entire test sets), we directly feed the whole test sets into approaches and get the prediction scores. Then, we calculate the evaluation metrics based on the prediction scores. When testing in Setup II (using datasets that exclude large commits), we will first investigate the code change size of each commit in the test sets and drop the largest 10% commits. Then, we feed the commits left into approaches to get the prediction scores and calculate the evaluation metrics.

5 EXPERIMENTAL RESULTS

5.1 RQ1: How effective is SimCom?

To answer this RQ, we carry out an experiment by adopting typical baselines to show the effectiveness of SimCom. We do experiments in two different setups.

5.1.1 ***Results on Entire Datasets.*** Table 3 shows the performance of all the compared approaches for Setup I (Entire Datasets Setup) in terms of AUC-ROC, AUC-PR, and F1-scores respectively. In Table 3, we categorize baselines into three groups: traditional machine learning models, deep learning-based models designed for the JIT defect prediction, and a popular ensemble learning model (i.e. Extreme Gradient Boosting (XGBoost)) [11]. We use the default parameters of the XGBoost Classifier in the XGBoost package [4]. XGBoost is trained with the commit-level hand-made features described in Table 1 and the defectiveness labels. In this paper, we use XGBoost as a baseline to show the superiority of combining both simple and complex models (e.g. SimCom) over simply ensembling multiple traditional ML models (e.g. XGBoost). In Table 3, the last column shows the performance of our proposed model SimCom, and the numbers in brackets are improvements over the best-performing baseline in each project and evaluation metric. Please note that the best-performing baseline varies based on the subjects and metrics. We use light gray color to show the best-performing baselines (excluding SimCom) and use numbers in bold and dark gray color to indicate the best model (including SimCom).

The experimental results demonstrate that our approach SimCom significantly and consistently outperforms all baselines for different evaluation metrics except for only one case (i.e. AUC-PR metrics on Gerrit). SimCom outperforms LApredict (the state-of-the-art approach) by 6.6% and 15.7% in terms of the AUC-ROC and AUC-PR on average. In addition, SimCom provides 2.6 times more accurate predictions than LApredict in terms of F1 scores on average. Particularly, SimCom leads to improvements of 13.9% and 24.0% in AUC-ROC and AUC-PR on the GO dataset. Compared to the ensemble model on multiple simple models (i.e. XGBoost), SimCom leads to improvements of 6.0%, 12.4%, and 24.0% in AUC-ROC, AUC-PR, and F1 on average. It indicates that combining both

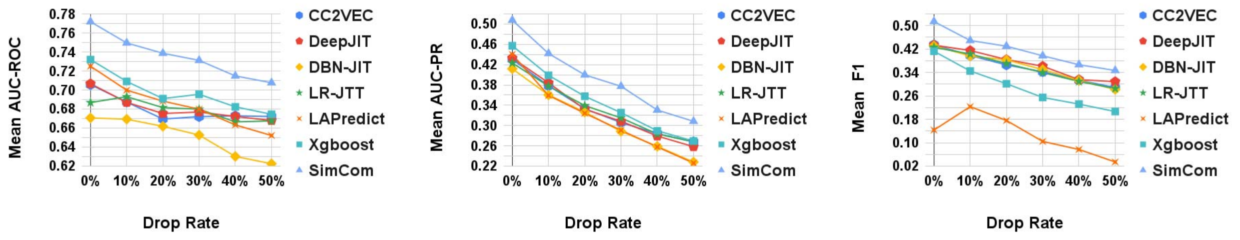

Figure 3: The average performance of SimCom and baselines in Setup II with different large commit drop rates.

simple and complex models can lead to better performance than only combining simple models.

In general, our approach SimCom outperforms all baselines consistently on different evaluation metrics and projects except for the Gerrit project when using AUC-PR as the evaluation metric. As shown in Table 2, however, Gerrit is highly imbalanced in both the training and the testing dataset so that the number of defective commits is small. Approaches including SimCom may not be able to learn effective features (patterns) because of the lack of defective commits in training data. In this case, we find that all approaches show poor performance on Gerrit datasets in terms of AUC-PR.

5.1.2 ***Results on Datasets Excluding Large Commits.*** Figure 3 shows the average performance of approaches in Setup II with respect to AUC-ROC, AUC-PR, and F1 scores, when dropping different ratios of largest commits. The y-axis in Figure 3 is the average performance over 6 datasets and the x-axis is the ratio of the largest commits we drop. For instance, the drop rate equals 20% indicating that we drop the top 20% of large commits in datasets. When the drop rate is 0%, the Setup II is degraded into Setup I (i.e. no commits dropped).

In general, comparing the performances in Setup I and Setup II, all approaches perform worse in Setup II. One possible reason is that distinguishing buggy commits from clean commits in datasets that *exclude large commits* is harder. Because over 40% of those dropped large commits are defective commits on average, the number of buggy commits in Setup II is smaller than that in Setup I. Thus, it is easier to make false-positive predictions in Setup II. The experimental results show that SimCom consistently and significantly outperforms the best-performing baseline by 5.5%, 13.0%, and 12.9% in terms of AUC-ROC, AUC-PR, and F1 scores on average across different drop rates. This indicates that the superiority of SimCom over baselines does not change when the experimental setup changes.

Answers to RQ1: In both Setup I and Setup II, SimCom outperforms all baselines consistently and significantly on diverse evaluation metrics and projects for most cases. In general, SimCom achieves improvements of 5.5–6.0%, 12.4–13.0%, and 12.9–18.1% in terms of AUC-ROC, AUC-PR, and F1 scores on average compared to best-performing baselines.

5.2 RQ2: How do the component models of SimCom perform?

5.2.1 ***Results of Setup I.*** To answer this RQ, we employ an ablation study to analyze the contribution of each component model (i.e. the simple and complex model). Table 4 shows the performance of each component model of SimCom in Setup I in terms of AUC-ROC, AUC-PR, and F1 scores respectively. We use light gray cell color to indicate the better model between simple and complex models and use boldface to indicate the best model (including SimCom).

In the ablation study, we could still see the saga of simple vs. complex: for some projects, the simple model performs better while the complex model is better in some other projects. In general, we find that the superiority of simple and complex models varies with different projects and different evaluation metrics. One advantage of combining the simple and complex models is that the combined

Table 4: Ablation study of each component model of SimCom in Setup I

	AUC-ROC			AUC-PR			F1-score		
	Simple	Complex	SimCom	Simple	Complex	SimCom	Simple	Complex	SimCom
QT	0.739	0.757	**0.771**	0.315	0.349	**0.370**	0.384	0.394	**0.402**
OpenStack	0.760	0.764	**0.786**	0.433	0.456	**0.474**	0.478	0.457	**0.491**
JDT	0.707	0.696	**0.716**	0.689	0.677	**0.694**	**0.656**	0.619	0.645
Platform	0.798	0.814	**0.829**	0.605	0.641	**0.657**	0.626	0.633	**0.646**
Gerrit	0.753	0.720	**0.756**	0.174	0.146	**0.178**	0.255	0.222	**0.270**
GO	0.740	0.764	**0.778**	0.635	0.652	**0.681**	0.606	0.634	**0.640**
Mean	0.750	0.753	**0.773**	0.475	0.487	**0.509**	0.501	0.493	**0.516**

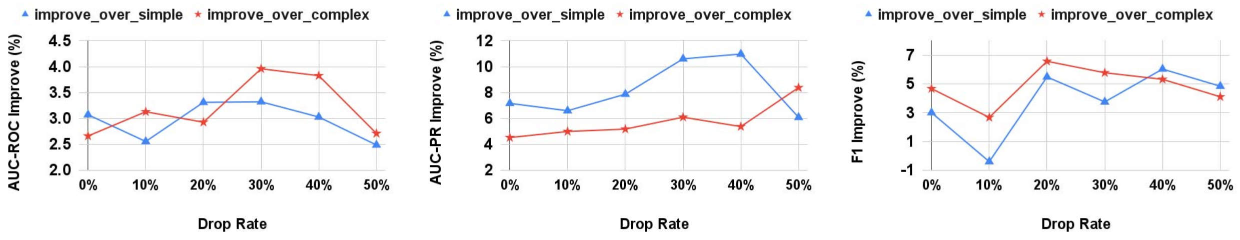

Figure 4: The average improvement percentage of SimCom over its components in Setup II with different large commit drop rates.

model (i.e. SimCom) leads to consistently better (3%–6% in average, up to 5.1%, 17.5%, 7.5% in AUC-ROC, AUC-PR, and F1-scores respectively) performance for most cases in this study.

5.2.2 ***Results of Setup II***. We can still see the superiority of SimCom over its component models in Setup II. Figure 4 depicts the improvement rates of SimCom over the simple model and complex model in terms of the average performance across six projects. As Figure 4 shows, when dropping different ratios of large commits, SimCom still consistently outperforms its component models by 3–6% on average for most cases.

Answers to RQ2: In the ablation study, the superiority of simple and complex component models varies with different projects and metrics. By combining simple and complex models, SimCom demonstrates consistently better performance (3–6% on average) for both Setups I and II.

5.3 RQ3: How do the different model fusion methods impact the performance?

5.3.1 ***Early Fusion v.s. Late Fusion***. In general, there are two ways to do model fusion: early fusion and late fusion. Early fusion is a way of fusing multiple data (usually multi-modal data) before feeding them into a classifier [18, 24, 68]. To carry out the early fusion, in this study, we concatenate the hand-made features with the commit representation Z in the complex model (described in 3.2.4) and feed the concatenated vector into a Fully Connected Layer (FC). The deep learning modules of the complex model and the last FC are updated together to minimize the binary cross-entropy.

5.3.2 ***Different Late Fusion Strategies***. Late fusion can have different strategies to determine how to finally combine the outputs of the independently trained models (e.g. product, average, or weighted average of the outputs of all models [80]). We conduct experiments to show how the model performance changes when late fusion strategies vary.

5.3.3 ***Results***. Table 5 presents the average performance of the SimCom variants using different model fusion methods and strategies. For rows of weighted average strategy, we set different weights from predictions from simple and complex models other than the

Table 5: The average performance of different model fusion methods

Fusion Methods	Opeartion		ROC	PR	F1
Early Fusion	**Concatenation**		0.745	0.487	0.486
Late Fusion	**Average**		**0.773**	**0.509**	**0.516**
	Product		0.769	0.507	0.514
	Weighted	**3:7**	0.767	0.507	0.507
	Average	**4:6**	0.770	0.508	0.516
	(Sim:	**6:4**	0.771	0.508	0.516
	Com)	**7:3**	0.769	0.504	0.516

equal contributions (the average strategy). The experimental results show that late fusion is 4%–6% better than early fusion on average across the projects. The result supports our design to adopt late fusion in SimCom. Among different late fusion strategies, averaging is simple and very effective. Besides, the performance differences among different late fusion strategies are very small (0.8%–1.8%), indicating that different strategies only slightly affect the performance of SimCom.

Answers to RQ3: The late fusion method performs consistently better (about 4%–6%) than the early fusion method for all projects. Different late fusion strategies only slightly affect the performance of SimCom and averaging is the best strategy in our experiments.

6 DISCUSSION

Time Cost. A slow speed will impair the practical usability of the approach. It is interesting to investigate the time efficiency of SimCom in both training and testing. To measure the speed, we experiment on a desktop computer equipped with Nvidia GeForce RTX 2080 Ti and Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz. To avoid the randomization bias, we repeat the experiment 5 times. As Table 6 shows, SimCom could be trained within 450 seconds (i.e. 7.5 minutes) and SimCom could give all predictions within 3.7 seconds, demonstrating its efficiency.

Simple & Complex: Better Together. Simple models and complex models can be partners rather than adversaries. In prior work of

Table 6: Time cost of SimCom

Time	QT	OpenStack	JDT	Platform	Gerrit	GO
Train	432s	450s	66s	192s	252s	402s
Test	3.7s	3.6s	2.7s	3.1s	3.4s	3.7s

JIT defect prediction, most works focused on either how to make better use of hand-made features by traditional ML classifiers, or how to effectively extract features from the commit content by DL techniques. As there is a big gap between commit-level hand-made features (a list of numbers) and commit contents (commit message tokens and code tokens), there is little work to leverage these two distinct input data. Our results indicate that combining both the simple model (dealing with hand-made features) and the complex model (dealing with commit contents) is promising in JIT defect prediction. Our work suggests that future work should explore further this idea on other software engineering tasks that have both meaningful hand-made features and effective DL models.

Threats to Internal Validity. For all baselines considered in this study, we directly use the source code in the replication packages of the studied techniques and use the default hyper-parameters. To reduce the threat, we carefully reviewed the experimental scripts to ensure the correctness. We also release our replication package for others to check.

Threats to External Validity. Threats to external validity are concerned with the generalizability of our findings. To reduce the threat, we use a diverse and large-scale JIT defect prediction dataset of real-world projects in different programming languages collected by Zeng et al. [81]. As this dataset contains a large number of labeled commits from six projects, our findings on this larger-scale dataset have the generalizability to other projects to some extend.

Threats to Construct Validity. One of the main threats to construct validity is the evaluation metrics we choose. To mitigate this threat, following the DeepJIT and LApredict studies, we adopt the widely-used AUC-ROC score to evaluate the performance. As ROC curves can lead to an overly optimistic view of a model's performance on skewed datasets [17], we also adopt an alternative metric for AUC-ROC (i.e. AUC-PR) that is more suitable for the skewed datasets. Besides, to evaluate the ability of the approaches in making correct predictions, we also consider the F1-score, which is computed based on the number of true/false positive/negatives.

7 RELATED WORK

Many approaches have been proposed for JIT defect prediction. Mockus et al. [53] built a model by using the historic information in commits and use the model to predict the risk of new commits. Kamei et al. [37] built an effort-aware prediction model using Logistic Regression with 14 change-level, hand-crafted metrics (features). Yang et al. [77] applied Deep Belief Network (DBN) to extract higher-level information from the change-level metrics. Later, Yang et al. [76] proposed a two-layer ensemble learning model that combines decision tree and ensemble learning. This two-layer model achieved better performance for JIT defect prediction. Young et al. [79] proposed a new ensemble approach by using arbitrary classifiers and optimizing the weights of the classifiers. Liu et al. [44] proposed an unsupervised learning approach based on code churn in effort-aware settings. Cabral et al. [10] proposed a new sampling method to address the issues of verification latency and class imbalance evolution in the online JIT defect prediction setting. Then, Yan et al. [75] proposed a two-phase framework that can handle the identification and localization tasks at the same time. Recently, Hoang et al. [29] proposed DeepJIT, which uses deep learning techniques to extract the features automatically from commit logs and code changes. To enhance it, Hoang et al. [30] proposed a new approach that learns the representation of code changes through the supervision of commit logs. Recently, Zeng et al. [81] conducted a study on the deep learning-based JIT defect prediction models (i.e. DeepJIT and CC2Vec) on an extended dataset and found that deep learning-based approaches cannot outperform a simple model namely LApredict.

Most of the existing work focused on either how to make better use of hand-made features by traditional ML classifiers, or how to extract better features from the commits by DL techniques. Different from the prior work, our model combines the simple model with the complex model by late fusion technique and outperforms the baselines significantly. Our results indicate that it is promising to combine the simple model (hand-crafted features with ML classifiers) and the complex model (commit contents with DL methods) to achieve better performance.

8 CONCLUSION AND FUTURE WORK

In this work, we propose an approach that combines one simple model and one complex model via late fusion. The experimental results show that our approach can significantly outperform the baselines and the state-of-the-art by 6.0%, 12.4%, and 18.1% in terms of AUC-ROC, AUC-PR, and F1-score on average. To further evaluate the effectiveness of our approach in different settings, we also explore the performance of JIT defect predictors in another setting that excludes large commits (i.e. top 10%–50% large commits): our approach can outperform the best-performing baselines by a large improvement (5.5%, 13.0%, and 12.9% in terms of AUC-ROC, AUC-PR, and F1-score) in this setting, indicating the effectiveness of our approach in different settings. Besides, we also carried out the ablation study and further analysis. We found that our approach (combining simple and complex models) performs consistently better (3–6% on average) than the simple model only or the complex model only in all different evaluation metrics and software projects.

We share the replication package[1] for further evaluation and extension of our study. In the future, we are interested in investigating the effectiveness of combining simple models and complex models in other SE tasks.

Acknowledgment. This research / project is supported by the National Research Foundation, Singapore, under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

[1]https://github.com/soarsmu/SimCom_JIT

REFERENCES

[1] [n.d.]. Code Contribution Guidelines for DSPACE. https://wiki.lyrasis.org/display/DSPACE/Code+Contribution+Guidelines.

[2] [n.d.]. Contribution Guidelines for Cake. https://cakebuild.net/community/contributing/contribution-guidelines.

[3] [n.d.]. Contribution guidelines for SKY UX. https://developer.blackbaud.com/skyux/contribute/contribution-process/guidelines.

[4] [n.d.]. XGBoost Documentation. https://xgboost.readthedocs.io/en/stable/index.html.

[5] Erik Arisholm, Lionel C. Briand, and Magnus Fuglerud. 2007. Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software. *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)* (2007), 215–224.

[6] Joseph Bockhorst and Mark W. Craven. 2004. Markov Networks for Detecting Overalpping Elements in Sequence Data. In *NIPS*.

[7] Leo Breiman. 2004. Random Forests. *Machine Learning* 45 (2004), 5–32.

[8] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[9] Razvan C. Bunescu, Ruifang Ge, Rohit J. Kate, Edward M. Marcotte, Raymond J. Mooney, Arun K. Ramani, and Yuk Wah Wong. 2005. Comparative experiments on learning information extractors for proteins and their interactions. *Artificial intelligence in medicine* 33 2 (2005), 139–55.

[10] George G. Cabral, Leandro L. Minku, Emad Shihab, and Suhaib Mujahid. 2019. Class Imbalance Evolution and Verification Latency in Just-in-Time Software Defect Prediction. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), 666–676.

[11] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[12] Xiang Chen, Yingquan Zhao, Qiuping Wang, and Zhidan Yuan. 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Inf. Softw. Technol.* 93 (2018), 1–13.

[13] Adele Cutler, D. Richard Cutler, and John R. Stevens. 2012. *Random Forests*. Springer US, Boston, MA, 157–175. https://doi.org/10.1007/978-1-4419-9326-7_5

[14] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)* (2010), 31–41.

[15] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2011. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering* 17 (2011), 531–577.

[16] Jesse Davis, Elizabeth S. Burnside, Inês de Castro Dutra, David Page, Raghu Ramakrishnan, Vítor Santos Costa, and Jude W. Shavlik. 2005. View Learning for Statistical Relational Learning: With an Application to Mammography. In *IJCAI*.

[17] Jesse Davis and Mark H. Goadrich. 2006. The relationship between Precision-Recall and ROC curves. *Proceedings of the 23rd international conference on Machine learning* (2006).

[18] Yuan Dong, Shan Gao, Kun Tao, Jiqing Liu, and Haila Wang. 2013. Performance evaluation of early and late fusion methods for generic semantics indexing. *Pattern Analysis and Applications* 17 (2013), 37–50.

[19] Wei Fu and Tim Menzies. 2017. Easy over hard: a case study on deep learning. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017).

[20] Harold Valdivia Garcia, Emad Shihab, and Meiyappan Nagappan. 2018. Characterizing and predicting blocking bugs in open source projects. *J. Syst. Softw.* 143 (2018), 44–58.

[21] Mark H. Goadrich, Louis Oliphant, and Jude W. Shavlik. 2004. Learning Ensembles of First-Order Clauses for Recall-Precision Curves: A Case Study in Biomedical Information Extraction. In *ILP*.

[22] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2015. Deep Learning. *Nature* 521 (2015), 436–444.

[23] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey P. Siy. 2000. Predicting Fault Incidence Using Software Change History. *IEEE Trans. Software Eng.* 26 (2000), 653–661.

[24] Hatice Gunes and Massimo Piccardi. 2005. Affect recognition from face and body: early fusion vs. late fusion. *2005 IEEE International Conference on Systems, Man and Cybernetics* 4 (2005), 3437–3443 Vol. 4.

[25] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. *2010 ACM/IEEE 32nd International Conference on Software Engineering* 1 (2010), 495–504.

[26] A. Hassan. 2009. Predicting faults using the complexity of code changes. *2009 IEEE 31st International Conference on Software Engineering* (2009), 78–88.

[27] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59 (2016), 122–131.

[28] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *ArXiv* abs/1207.0580 (2012).

[29] Thong Hoang, K. Dam, Yasutaka Kamei, D. Lo, and N. Ubayashi. 2019. DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019), 34–45.

[30] Thong Hoang, Hong Jin Kang, Julia L. Lawall, and David Lo. 2020. CC2Vec: Distributed Representations of Code Changes. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), 518–529.

[31] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In *IJCAI*.

[32] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013), 279–289.

[33] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and A. Hassan. 2015. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21 (2015), 2072–2106.

[34] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken ichi Matsumoto, Bram Adams, and A. Hassan. 2010. Revisiting common bug prediction findings using effort-aware models. *2010 IEEE International Conference on Software Maintenance* (2010), 1–10.

[35] Yasutaka Kamei, Akito Monden, Shinsuke Matsumoto, Takeshi Kakimoto, and Ken ichi Matsumoto. 2007. The Effects of Over and Under Sampling on Fault-prone Module Detection. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)* (2007), 196–204.

[36] Yasutaka Kamei and Emad Shihab. 2016. Defect Prediction: Accomplishments and Future Challenges. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 5 (2016), 33–45.

[37] Yasutaka Kamei, Emad Shihab, B. Adams, A. Hassan, A. Mockus, Anand Sinha, and N. Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39 (2013), 757–773.

[38] Taghi M. Khoshgoftaar, Xiaojin Yuan, and Edward B. Allen. 2004. Balancing Misclassification Rates in Classification-Tree Models of Software Quality. *Empirical Software Engineering* 5 (2004), 313–330.

[39] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)* (2006), 81–90.

[40] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *EMNLP*.

[41] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2015).

[42] Akif Günes Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. 2009. An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules. *IEEE Transactions on Software Engineering* 35 (2009), 293–304.

[43] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering* 34 (2008), 485–496.

[44] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code Churn: A Neglected Metric in Effort-Aware Just-in-Time Defect Prediction. *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2017), 11–19.

[45] Jiayi Ma, Yong Ma, and Chang Li. 2019. Infrared and visible image fusion methods and applications: A survey. *Inf. Fusion* 45 (2019), 153–178.

[46] Suvodeep Majumder, Nikhil Balaji, Katie Brey, Wei Fu, and Tim Menzies. 2018. 500+ Times Faster than Deep Learning: (A Case Study Exploring Faster Methods for Text Mining StackOverflow). *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)* (2018), 554–563.

[47] Christopher D. Manning and Hinrich Schütze. 2002. Foundations of statistical natural language processing. In *SGMD*.

[48] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken ichi Matsumoto, and Masahide Nakamura. 2010. An analysis of developer metrics for fault prediction. In *PROMISE '10*.

[49] Shane McIntosh and Yasutaka Kamei. 2018. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *IEEE Transactions on Software Engineering* 44 (2018), 412–428.

[50] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Trans. Software Eng.* 33 (2007), 2–13.

[51] Tim Menzies, Burak Turhan, Ayse Basar Bener, Gregory Gay, Bojan Cukic, and Yue Jiang. 2008. Implications of ceiling effects in defect predictors. In *PROMISE '08*.

[52] Ayse Tosun Misirli and Ayse Basar Bener. 2009. Reducing false alarms in software defect prediction by decision threshold optimization. *2009 3rd International Symposium on Empirical Software Engineering and Measurement* (2009), 477–480.

[53] Audris Mockus and David M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5 (2000), 169–180.
[54] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *2008 ACM/IEEE 30th International Conference on Software Engineering* (2008), 181–190.
[55] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective.* MIT press.
[56] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* (2005), 284–292.
[57] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. *Proceedings of the 28th international conference on Software engineering* (2006).
[58] Ning Nan and Donald E. Harter. 2009. Impact of Budget and Schedule Pressure on Software Development Cycle Time and Effort. *IEEE Transactions on Software Engineering* 35 (2009), 624–637.
[59] Anh Tuan Nguyen and Tien Nhut Nguyen. 2015. Graph-Based Statistical Language Model for Code. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* 1 (2015), 858–868.
[60] Foster J. Provost, Tom Fawcett, and Ron Kohavi. 1998. The Case against Accuracy Estimation for Comparing Induction Algorithms. In *ICML*.
[61] Ranjith Purushothaman and Dewayne E. Perry. 2005. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering* 31 (2005), 511–526.
[62] Foyzur Rahman, Daryl Posnett, and Premkumar T. Devanbu. 2012. Recalling the "imprecision" of cross-project defect prediction. In *SIGSOFT FSE*.
[63] Z. Rana, Mian M. Awais, and Shafay Shamail. 2009. An FIS for Early Detection of Defect Prone Modules. In *ICIC*.
[64] Omer Sagi and Lior Rokach. 2018. Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8, 4 (2018), e1249.
[65] Takaya Saito and Marc Rehmsmeier. 2015. The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets. *PLoS ONE* 10 (2015).
[66] Aliaksei Severyn and Alessandro Moschitti. 2015. Learning to Rank Short Text Pairs with Convolutional Deep Neural Networks. *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval* (2015).
[67] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, A. Hassan, and Ken ichi Matsumoto. 2012. Studying re-opened bugs in open source software. *Empirical Software Engineering* 18 (2012), 1005–1042.
[68] Cees G. M. Snoek, Marcel Worring, and Arnold W. M. Smeulders. 2005. Early versus late fusion in semantic video analysis. In *MULTIMEDIA '05*.
[69] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online Defect Prediction for Imbalanced Data. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* 2 (2015), 99–108.
[70] C. Tantithamthavorn, A. Hassan, and Ken ichi Matsumoto. 2020. The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models. *IEEE Transactions on Software Engineering* 46 (2020), 1200–1219.
[71] Sofia Tsekeridou and Ioannis Pitas. 2001. Content-based video parsing and indexing based on audio-visual interaction. *IEEE Trans. Circuits Syst. Video Technol.* 11 (2001), 522–535.
[72] Burak Turhan, Tim Menzies, Ayse Basar Bener, and Justin S. Di Stefano. 2008. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering* 14 (2008), 540–578.
[73] Zhiyuan Wan, Xin Xia, A. Hassan, D. Lo, Jianwei Yin, and Xiaohu Yang. 2020. Perceptions, Expectations, and Challenges in Defect Prediction. *IEEE Transactions on Software Engineering* 46 (2020), 1241–1266.
[74] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016), 297–308.
[75] Meng Yan, Xin Xia, Yuanrui Fan, A. Hassan, David Lo, and Shanping Li. 2020. Just-In-Time Defect Identification and Localization: A Two-Phase Framework. *IEEE Transactions on Software Engineering* (2020), 1–1.
[76] Xinli Yang, D. Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Inf. Softw. Technol.* 87 (2017), 206–220.
[77] Xinli Yang, D. Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. *2015 IEEE International Conference on Software Quality, Reliability and Security* (2015), 17–26.
[78] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton K. N. Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016).
[79] Steven Young, Tamer Abdou, and Ayse Basar Bener. 2018. A Replication Study: Just-in-Time Defect Prediction with Ensemble Learning. *2018 IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)* (2018), 42–47.
[80] Shi Yu, Tillmann Falck, Anneleen Daemen, Léon-Charles Tranchevent, Johan A. K. Suykens, Bart De Moor, and Yves Moreau. 2010. L2-norm multiple kernel learning and its application to biomedical data fusion. *BMC Bioinformatics* 11 (2010), 309 – 309.
[81] Zhen Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we? *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2021).
[82] Ye Zhang and Byron Wallace. 2015. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820* (2015).