A study on the impact of pre-trained model on Just-In-Time defect prediction

Yuxiang Guo[1], Xiaopeng Gao[2], Zhenyu Zhang[3], W.K.Chan[4], Bo Jiang[5,*]
[1,2,5]School of Computer Science and Engineering, Beihang University, Beijing, China
[3]State Key Laboratory of Computer Science Institute of Software, Chinese Academy of Sciences, Beijing, China
[4]Department of Computer Science, City University of Hong Kong, Hong Kong
IrisG@buaa.edu.cn, gxp@buaa.edu.cn, zhangzy@ios.ac.cn, wkchan@cityu.edu.hk, jiangbo@buaa.edu.cn

Abstract—**Previous researchers conducting Just-In-Time (JIT) defect prediction tasks have primarily focused on the performance of individual pre-trained models, without exploring the relationship between different pre-trained models as backbones. In this study, we build six models: RoBERTaJIT, CodeBERTJIT, BARTJIT, PLBARTJIT, GPT2JIT, and CodeGPTJIT, each with a distinct pre-trained model as its backbone. We systematically explore the differences and connections between these models. Specifically, we investigate the performance of the models when using Commit code and Commit message as inputs, as well as the relationship between training efficiency and model distribution among these six models. Additionally, we conduct an ablation experiment to explore the sensitivity of each model to inputs. Furthermore, we investigate how the models perform in zero-shot and few-shot scenarios. Our findings indicate that each model based on different backbones shows improvements, and when the backbone's pre-training model is similar, the training resources that need to be consumed are much more closer. We also observe that Commit code plays a significant role in defect detection, and different pre-trained models demonstrate better defect detection ability with a balanced dataset under few-shot scenarios. These results provide new insights for optimizing JIT defect prediction tasks using pre-trained models and highlight the factors that require more attention when constructing such models. Additionally, CodeGPTJIT and GPT2JIT achieved better performance than DeepJIT and CC2Vec on the two datasets respectively under 2000 training samples. These findings emphasize the effectiveness of transformer-based pre-trained models in JIT defect prediction tasks, especially in scenarios with limited training data.**

Keywords—Just-In-time defect prediction; pre-trained model; few-shot scenario; model sensitivity

I. INTRODUCTION

The existence of software defects may lead to significant economic loss to the end users. Defect prediction can help developers identify defects before deploying buggy software.

Just-In-Time(JIT for short) defect prediction predicts the presence or absence of defects based on software change characteristics[1]. JIT defect prediction can remind software developers to review their submitted code for defects as early as possible, which significantly reduces the cost of software maintenance. Most of the traditional JIT defect prediction techniques rely on manually extracted features and use different machine learning methods such as logistic regression, SVM, etc., to train a suitable classifier. Mockus et al. were the first to propose the selection of manually extracted features, which they divided into different categories, such as changes in code lines and developer expertise[2]. These features have been extensively studied in previous works[3–6]. Some other studies have extended the traditional features by adding characteristics like code complexity, number of changed files, and word frequency of code modules to improve code change characterization[1][7–12].

With the recent development of deep learning, various network structures have been proposed, providing new ideas for feature extraction in code change[13–19]. Some researchers have introduced deep learning technologies into feature extraction for JIT defect prediction[20, 21]. For example, Hong et al.[22] propose to extract features directly from the data submitted by the developer using deep learning techniques.

Nowadays, feature extraction using deep learning technology has become a common method for deep JIT defect prediction models like DeepJIT[22] and CC2Vec[23]. DeepJIT is an end-to-end JIT defect prediction model that predicts defects with the Commit code and Commit message in a submission[22]. CC2Vec improves DeepJIT by a pre-trained distributed change-code vector in advance, resulting in performance improvements on both the QT and Openstack[23].

Hong believes that the extra pre-trained code vector is an important reason for the improved performance of CC2Vec model in JIT defect prediction[23]. For the traditional two-step deep learning models based on train-test such as DeepJIT, they only learn domain knowledge from scratch without any external knowledge. Different from DeepJIT, CC2Vec utilizes pre-trained code vector as a part of the input in JIT defect prediction model.

However, CC2Vec introduces extra information before training a model, that information is still intra-domain which means all the information learned by DeepJIT and CC2Vec are from the same dataset. The appearance of large language models provides deep learning methods with a new way of pre-training and fine-tuning to resolve the previous tasks. Some transformer encoder-based pre-trained models like RoBERTa[24], CodeBERT[25], decoder-based pre-trained models like GPT2[43], CodeGPT, encoder-decoder based model like BART[44], PLBART[45], T5[47]. have shown huge improvement in previous natural language and program language tasks like text classification, code search and vulnerability detection[35, 46]. The process of fine-tuning can make a pre-trained model specific to a certain task.

Few previous works have explored how transformer-based pre-trained models perform in JIT defect prediction tasks. Therefore, it is crucial to systematically explore how different models behave and what they are sensitive to.

In this study, we build six deep JIT prediction models: RoBERTaJIT, CodeBERTJIT, GPT2JIT, CodeGPTJIT, BARTJIT, and PLBARTJIT, using corresponding pre-trained models and CNN network. To comprehensively evaluate these pre-trained JIT defect prediction models, we selected three transformer-based models and their natural language and programming language versions as backbones. Our primary motivations were to investigate the performance of different transformer structures in JIT defect prediction tasks and how they behave in different situations. We used Commit code and Commit message as model inputs and evaluated the models based on metrics such as AUC score, accuracy, precision, recall, and F1 score to compare their performance. Additionally, we compared the training efficiency and distribution similarity of prediction results among the six JIT prediction models. We also conducted ablation experiments on different input parts of the four models to analyze the influence of input data on their performance. Lastly, we explored the models' performance under zero-shot and few-shot scenarios. All experiments were conducted using data from QT and Openstack projects.

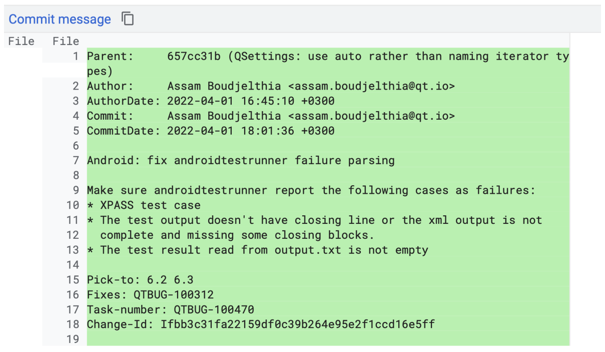
The contribution of this work is fourfold. First, rather than concentrating solely on individual model performance, our research delves deeply into the comparative analysis of these models based on different backbones. Second, we analyze the correlation between training efficiency and model distribution across the six models and we find that when the backbone's pre-training model is similar, the training resources that need to be consumed are much closer. Third, we conduct an extensive ablation experiment to gain insights into the sensitivity of each model to various inputs and find that Commit code plays a significant role in defect detection between Commit code and Commit message, which provides the idea of optimization: researchers can focus on the processing of Commit code. Finally, we explore the performance of different models under zero-shot and few-shot learning scenarios. We find that the pre-trained models demonstrate superior defect detection abilities with a balanced dataset, and we obtain two models which can outperform DeepJIT and CC2Vec under a 2000-shot scenario, which provides the possibility to optimize a model under the few-shot scenario.

II. Background on JIT defect prediction and pre-trained model

In this section, we will revisit background information on JIT defect prediction and pre-trained models.

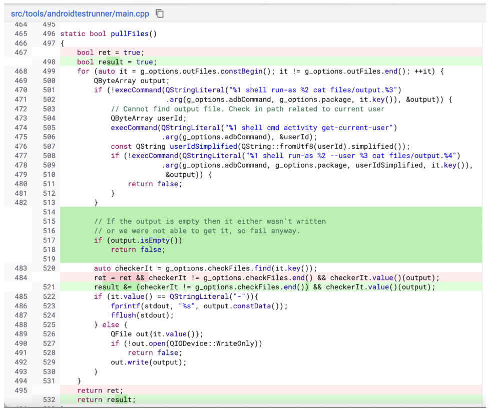
A. *Definition of submission in JIT defect prediction*

Developers often modify more than one file when submitting code to a project. Figure 1 presents an example of a Commit message while Figure 2 presents the example of a Commit code within one submission. The Commit message shows the developer who submitted the code (line 4), the


Commit message

```
File  File
 1 Parent:     657cc31b (QSettings: use auto rather than naming iterator ty
   pes)
 2 Author:     Assam Boudjelthia <assam.boudjelthia@qt.io>
 3 AuthorDate: 2022-04-01 16:45:10 +0300
 4 Commit:     Assam Boudjelthia <assam.boudjelthia@qt.io>
 5 CommitDate: 2022-04-01 18:01:36 +0300
 6
 7 Android: fix androidtestrunner failure parsing
 8
 9 Make sure androidtestrunner report the following cases as failures:
10 * XPASS test case
11 * The test output doesn't have closing line or the xml output is not
12   complete and missing some closing blocks.
13 * The test result read from output.txt is not empty
14
15 Pick-to: 6.2 6.3
16 Fixes: QTBUG-100312
17 Task-number: QTBUG-100470
18 Change-Id: Ifbb3c31fa22159df0c39b264e95e2f1ccd16e5ff
19
```

Figure 1. An example Commit message


src/tools/androidtestrunner/main.cpp

```
static bool pullFiles()
{
    bool ret = true;
    bool result = true;
    for (auto it = g_options.outFiles.constBegin(); it != g_options.outFiles.end(); ++it) {
        QByteArray output;
        if (!execCommand(QStringLiteral("%1 shell run-as %2 cat files/output.%3")
                         .arg(g_options.adbCommand, g_options.package, it.key()), &output)) {
            // Cannot find output file. Check in path related to current user
            QByteArray userId;
            execCommand(QStringLiteral("%1 shell cmd activity get-current-user")
                        .arg(g_options.adbCommand), &userId);
            const QString userIdSimplified(QString::fromUtf8(userId).simplified());
            if (!execCommand(QStringLiteral("%1 shell run-as %2 --user %3 cat files/output.%4")
                        .arg(g_options.adbCommand, g_options.package, userIdSimplified, it.key()),
                         &output)) {
                return false;
            }
        }

        // If the output is empty then it either wasn't written
        // or we were not able to get it, so fail anyway.
        if (output.isEmpty())
            return false;

        auto checkerIt = g_options.checkFiles.find(it.key());
        ret = ret && checkerIt != g_options.checkFiles.end() && checkerIt.value()(output);
        result &= (checkerIt != g_options.checkFiles.end()) && checkerIt.value()(output);
        if (it.value() == QStringLiteral("-")){
            fprintf(stdout, "%s", output.constData());
            fflush(stdout);
        } else {
            QFile out{it.value()};
            if (!out.open(QIODevice::WriteOnly))
                return false;
            out.write(output);
        }
    }
    return ret;
    return result;
```

Figure 2. An example Commit code

commit time (line 5), the task number (line 17), and the change ID (line 18). The most important information in the Commit message is the developer's annotation (lines 7-13). The developer's annotation is made up of natural language text that reflects the developer's intention for submission. Annotation is a useful feature often used in deep JIT defect prediction tasks, which is one of the input data used in our model. For convenience, we call the annotated data "Commit message".

Figure 2 shows the part of the changed code in a submission whose changed ID is "Ifbb3c31fa22159df0c39b264e95e2f1ccd16e5ff". The green and red code identifies changed code in the file whose path within the project is "src/tools/androidtestrunner/main", the green part represents the added code and the red part represents the deleted code. In Figure 2, we can see that the major bug of this submission is replacing the variable "ret" with "result" and changing the symbol "&&" to "&". From Figure 2, we know that there are two types of changed code in a submission, added code and deleted code. Changed code in a submission is important information, which is another part of the input data used in our model. For convenience, we also call the changed code "Commit code".

B. DeepJIT and CC2Vec

Hong et al. introduced a novel deep Just-In-Time (JIT) defect prediction model, utilizing Commit code and Commit message as the model input. In contrast to previous approaches that relied on traditional handcrafted features, DeepJIT represents an end-to-end JIT defect prediction model, eliminating the need for manual feature extraction[22]. The model utilizes a Convolutional Neural Network (CNN) to extract meaningful features to achieve classification tasks.

To find a suitable vector representation with generalization, Hong et al. propose a neural network model that can learn a representation of code changes by pre-training the Commit code into a distributed vector CC2Vec in advance[23]. CC2Vec uses HAN[34] network architecture to better extract semantic information from code by training vectors using information from Commit messages submitted by developers at a time. Hong et al. achieved performance improvements in the JIT defect prediction task by concatenating the CC2Vec vector with other input data from DeepJIT.

C. Pre-trained model

Word embedding or distributed vector is a technique for converting natural language words into vectors or matrices such that computers can process and manipulate. Only by using reasonable word representation can we carry out subsequent machine learning more easily. The development of a pre-trained model is closely related to the research of word embedding. The first generation of pre-trained models is pre-trained word embedding. Some models, such as one-hot, Word2Vec[26], N-Gram[27] and Glove[28], can capture certain semantic information. But they are context-free. The first generation of pre-trained models has obvious shortcomings. For example, the embedding is static and is unable to consider the context, semantic role, and syntactic characteristics of the text, which makes it hard to distinguish polysemous words effectively. Therefore, another stage of the development of word embedding is to study context-related word embedding.

The second generation of the pre-trained model depends on the proposed transformer structure[33]. The main pre-trained models fall into three categories: encoder based such as ELMo[30], BERT[24], and RoBERTa[32], decoder based such as GPT[31], GPT2[43], encoder-decoder based such as BART[44]. These pre-trained models are usually trained on massive corpora and support many training tasks. Large-scale pre-training can learn more general language representation, provide a better initialization model for downstream tasks, improve the performance of target tasks, and accelerate the convergence of models. By continuously improving the existing model structure and adding code-related corpora, researchers have derived code-related versions of the corresponding pre-trained models like CodeBERT, CodeGPT, PLBART, etc. Which models obtain great performance in clone detection, vulnerability detection[35, 46]. But for JIT defect prediction task, the role of these models, and the scenarios in which they work, remains unexplored. Figure 3 shows the process of using a pre-trained model that feeds a set of tokens into the pre-trained model and outputs a generalized language representation vector. Finding suitable word embedding has been the focus of natural language research for a long time. Code is a semi-structured language similar to natural language, so when we use deep learning to solve problems related to program language, it is also necessary to find an appropriate embedding accordingly.

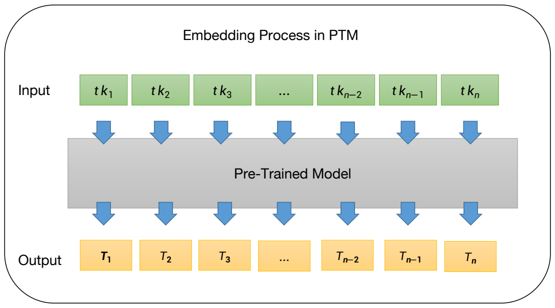


Figure 3. The process of using Pre-trained models

III. Model Structure

In our research, we developed six distinct JIT defect prediction models by leveraging pre-trained models as backbones and employing CNN networks for feature extraction. Specifically, we incorporated three transformer-based models: RoBERTa (an encoder-based model), GPT2 (a decoder-based model), and BART (an encoder-decoder-based model) to create RoBERTaJIT, GPT2JIT, and BARTJIT, respectively. Additionally, we extended the scope by selecting three code-related versions from the RoBERTa, GPT2, and BART models to establish CodeBERTJIT, CodeGPTJIT, and PLBARTGPT models. These models were designed to explore the effectiveness of different transformer structures in the JIT defect prediction task, aiming to provide comprehensive insights into their performances and capabilities.

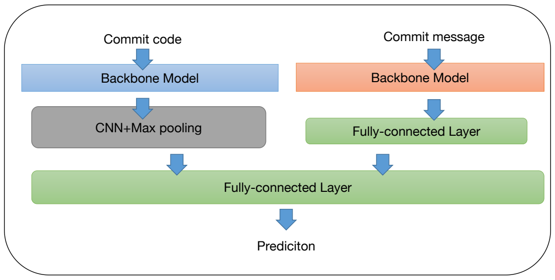


Figure 4. The framework of JIT defect prediction models with different pre-trained model as backbones

Figure 4 illustrates the framework of our six JIT defect prediction models, all of which utilize Commit code and Commit message as the model input. In the process of handling the Commit code, we adopt a similar approach as the DeepJIT model[22], where we select the most recent four patches from each submission. The Commit code, denoted as $C = [P_1, P_2, ..., P_{|C|}]$, comprises $|C|$ patches in a submission, where $|C| = 4$ represents the number of patches. In cases where the number of patches in a submission is less than 4, we employ "empty patches" to pad the remaining slots on the left. Each individual patch P_i is then treated as a sequence within the Commit code, which serves as input to our backbone pre-trained model for obtaining the embedding representation $\bar{P_i}$ using Equation (1).

$$\bar{P_i} = BackboneModel(P_i) \tag{1}$$

When Commit code in each submission is processed by backbone pre-trained model, we obtain the vector representation of Commit code $\bar{C}$ using $\oplus$ operation following Equation (2).

$$\bar{C} = \bar{P_1} \oplus \bar{P_2} \oplus ... \oplus \bar{P_{|C|}} \tag{2}$$

For the process of Commit message, we treat the whole Commit message as sequence $M = [w_1, w_2, w_3, ...w_{|m|}]$ in a submission whose sequences content contains 13-17 lines in Figure 1, M is a sequence composed of a set of words, and $|m|$ is the length of the sequence. Then we input Commit message M into the backbone model to get embedding representation $\bar{M}$ following Equation (3).

$$\bar{M} = BackboneModel(M) \tag{3}$$

Then we conduct feature extraction for $\bar{C}$ and $\bar{M}$ respectively. We use CNN and max-pooling to extract the features of code embedding $\bar{C}$. We define a filter $f \in R^{k \times d}$ and use K windows to extract new features of the Commit code. For the code patch $\bar{P_i}$ in $\bar{C}$, we obtain new vector x_i following Equation (4).

$$x_i = ReLU(f * \bar{P_{i:i+k-1}} + b_i) \tag{4}$$

In this paper, symbol $*$ is the sum of the product of the element, $ReLU$ is the nonlinear activation function and $b_i \in R$ is the bias value. New features are generated by applying filter f to every k file of the Commit code. The new feature x_i is strung together into vector X, as shown in Equation (5).

$$X = [x_1, x_2, ..., x_{|C|-k+1}] \tag{5}$$

Then we use the max-pooling method to process vector X to obtain the most significant feature Z_c, as shown in Equation (6).

$$Z_c = \max_{1 \leq i \leq |C|-k+1} (x_i) \tag{6}$$

Because the sequence length of the Commit message is shorter than that of the Commit code, only one layer of a fully connected network is used for feature extraction in message embedding $\bar{M}$. We can obtain feature vector Z_m following Equation (7).

$$Z_m = w_m \cdot \bar{M} + b_m \tag{7}$$

After obtaining the feature vector Z_c for the Commit code and Z_m for the Commit message, we fuse the Z_c and Z_m to get the feature vector Z for one submission, as shown in Equation (8).

$$Z = Z_c \oplus Z_m \tag{8}$$

Then we input Z into a layer of a fully connected network as a classifier. Finally, the high-dimensional vector is mapped to the one-dimensional vector by the $sigmod$ function to complete the prediction task of binary classification, as shown in Equation (9).

$$y = sigmod(ReLU(w_h \cdot Z + b_h)) \tag{9}$$

IV. Experimental research

A. Research Questions

RQ1: What is the performance of six pre-trained based JIT defect prediction models with different backbone structures?

RQ2: What is the relationship between the training effectiveness of different models and the distribution of predicted results?

RQ3: How sensitive are the different models to the input data?

RQ4: How do different models behave in few-shot scenarios?

B. Datasets

All of our experiments are carried out on QT and Openstack datasets. Table **??** shows the statistical description of QT and Openstack dataset. There are 25704 submissions in the QT dataset, including 1825 submissions that are labeled as defective. There are 1627 submissions in the Openstack dataset, including 1627 submissions that are labeled as defective. In this paper, 80% of the data is used as the training set, and the left 20% of the data is used as the test set for evaluation.

TABLE I
STATISTICS OF THE DATASET

Project	Commit	Defect	%Defect
QT	25704	1825	7.1
Openstack	13304	1627	12.2

C. Experiment Setup

We use a desktop PC as our experiment environment. The PC is running Ubuntu 18.04 and is equipped with RTX3090 with 24GB of memory. All experiments are implemented by Python3 and we use the official Python release version 3.7. We select the set of parameters that can minimize the loss of the validation set, the corresponding model is saved as the final model in our training process.

D. Criteria for evaluation

When a binary model is used for prediction, four different results are produced. A commit that introduces a defect is predicted to be defective namely True Positive(TP), a commit that does not introduce a defect was predicted to be defect-free namely True Negative(TN), a commit that does not introduce a defect was predicted to be defective namely False Positive(FP) and a commit which introduces defect was predicted to be defect-free namely False Negative(FN). **AUC** Area under the Curve of ROC(AUC) is often used to evaluate the training effect of a binary model. A high AUC score shows off the stability of a model.

Accuracy In particular, Accuracy (ACC) is the ratio of the number of correct predictions over the total number of predictions.

$$ACC = \frac{TN + TP}{TN + TP + FN + FP}$$

Precision Precision rate refers to the proportion of correctly predicted defects among all predicted defects.

$$Precision = \frac{TP}{TP + FP}$$

Recall Recall refers to the proportion of defects correctly predicted to all defects submitted.

$$Recall = \frac{TP}{TP + FN}$$

F1 score F1 score is a metric used to evaluate the performance of a binary classification model. It is the harmonic mean of precision and recall, and it takes both false positives and false negatives into account. The F1 score is a value between 0 and 1, where 1 represents perfect precision and recall, and 0 represents the worst possible performance.

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

E. Results and Discussions

Results and analysis for RQ1. For this research question, we try to explore the overall performance of different models in JIT defect prediction task under full fine-tuning scenario. To achieve this, we compare six models, each utilizing different backbone pre-trained models: RoBERTaJIT, CodeBERTJIT, GPT2JIT, CodeGPTJIT, BARTJIT, and PLBARTJIT. Additionally, we include two state-of-the-art models, DeepJIT and CC2Vec, for comparison. Most of the previous work only select auc score as evaluation index for JIT defect prediction model, we acknowledge that relying on a single metric may be limiting. To comprehensively explore the performance of these models, we additionally considered other evaluation metrics in this section and performed results analysis. Table II and Table III shows the experimental results of these six models on QT and Openstack projects. Results 1: All JIT defect prediction models constructed with three different transformer structures exhibit improvements in both datasets. As shown in Table II and Table III, all models with various pre-trained models yield significant enhancements across the evaluation metrics, with the exception of recall. It is clear that decoder-based models and encoder-decoder-based models are typically well-suited for generation tasks. However, in the JIT defect prediction task, which involves understanding, we observe that pre-trained models deliver excellent performance regardless of their different architectures. This suggests that when the model size is sufficiently large, it demonstrates adaptability to tasks that may not be inherently suited to its architecture.

TABLE II
EXPERIMENTAL RESULTS OF THE DIFFERENT MODELS ON QT PROJECT

Model	auc score	accuracy	precision	recall	F1 score
DeepJIT	0.79	0.73	0.17	0.70	0.27
CC2Vec	0.81	0.56	0.13	0.87	0.23
RoBERTaJIT	0.81	0.87	0.27	0.48	0.35
CodeBERTJIT	0.81	0.87	0.26	0.42	0.32
GPT2JIT	0.81	0.81	0.21	0.58	0.31
CodeGPTJIT	0.81	0.89	0.29	0.39	0.33
BARTJIT	0.82	0.82	0.23	0.62	0.34
PLBARTJIT	0.82	0.90	0.32	0.34	0.33

Results 2: Complicated Model is sensitive to the balance of the dataset. Figure 5 illustrates the distribution of prediction results for the six models on both the QT and Openstack datasets. It is evident that the models tend to predict most submissions as non-defective commits in the QT dataset, whereas this behavior differs in the Openstack dataset. This observation is also reflected in the results presented in Table II and Table III, where the recall value for the six models on the Openstack dataset significantly outperforms that on the QT dataset. While the difference in recall performance in the previous two state-of-art models, DeepJIt and CC2vec, is very small. To further investigate this phenomenon, we compared the existing results with the results obtained after extending the model's training time. Figure 6 depicts the differences between the base results and those achieved with longer training times. It is apparent that with increased training time, the accuracy of all six models improves, while the recall value deteriorates. This observation can be attributed to the strong learning ability of JIT defect prediction models based on pre-trained models, owing to their deep and complex network structures. Consequently, during the training process, when faced with an imbalanced dataset, the models' parameters tend to favor the class with more data. This provides an explanation for why these large pre-trained models exhibit lower recall rates in the JIT defect prediction task.

Results and analysis for RQ2. We investigate the overall training efficiency of different JIT defect prediction models with pre-trained models as backbones, as well as whether there is any relationship in the distribution of prediction results among the models. We first compare the training time, parameter size, and training loss changes. Table IV shows training time and model size respectively in RoBERTaJIT, CodeBERTJIT, GPT2JIT, CodeGPTJIT, BARTJIT, and

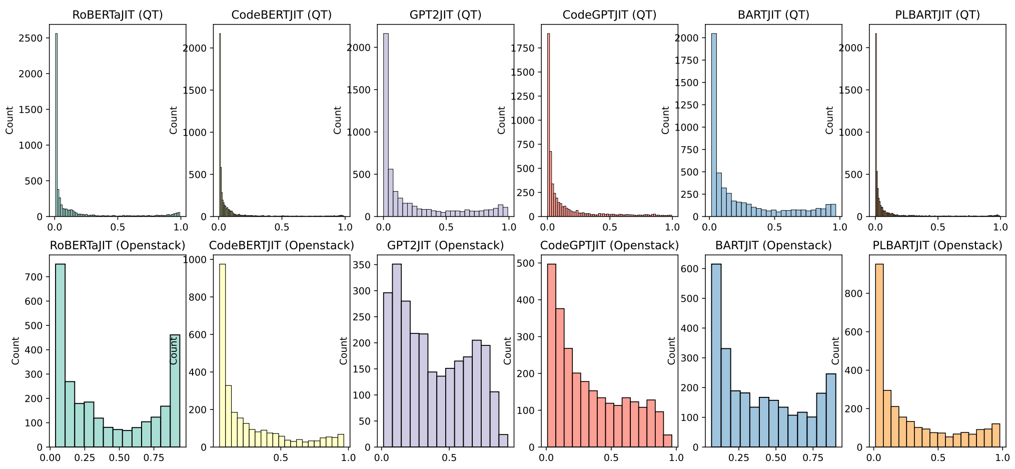


Figure 5. Distribution of prediction results for different models in QT and Openstack datasets

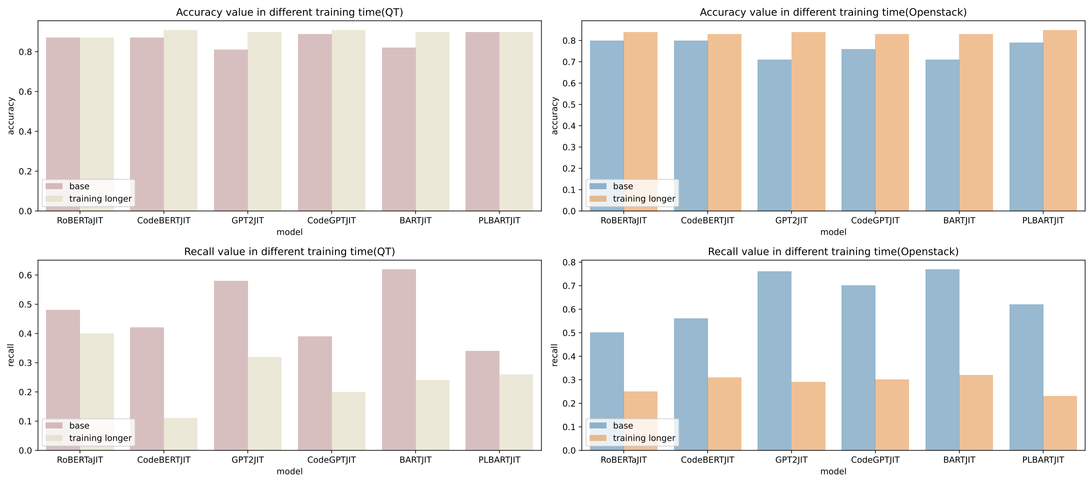


Figure 6. Variation of accuracy and recall with training time on QT and Openstack datasets

TABLE III
EXPERIMENTAL RESULTS OF THE DIFFERENT MODELS ON OPENSTACK PROJECT

Model	auc score	accuracy	precision	recall	F1 score
DeepJIT	0.74	0.61	0.17	0.76	0.28
CC2Vec	0.76	0.27	0.11	0.98	0.19
RoBERTaJIT	0.79	0.80	0.32	0.50	0.39
CodeBERTJIT	0.80	0.80	0.33	0.56	0.42
GPT2JIT	0.79	0.71	0.26	0.76	0.39
CodeGPTJIT	0.81	0.76	0.30	0.70	0.42
BARTJIT	0.81	0.71	0.27	0.77	0.40
PLBARTJIT	0.81	0.79	0.32	0.62	0.42

PLBARTJIT models. Figure 7 and Figure 8 respectively show training loss change of these six models in QT and Openstack projects. We also conduct T-test experiment to analyze the distribution similarity between 6 models, and Table V shows the T-test results between the six models on both two datasets.

Results 1: The training time and model size of different JIT defect prediction models with similar backbone models tend to be closer, backbone model which is code-related converges better than their corresponding non-code-related model. Table IV respectively shows the training time and model size of 6 models. We can find that the training time and model size of different models are not very different in general. BARTJIT and PLBART are a little faster than other models in the same training setting on both datasets. Additionally, it is evident that the training time and model size of JIT defect prediction models with similar backbone models are more comparable. For example, RoBERTaJIT is close to CodeBERTJIT, GPT2JIT is close to CodeGPTJIT, and BARTJIT is close to PLBARTJIT.

Moreover, Figure 7 and Figure 8 show the loss changes during the training process of the six models. It is observed that RoBERTaJIT, GPT2JIT, and BARTJIT perform slightly worse than CodeBERTJIT, CodeGPTJIT, and PLBARTJIT in terms of coverage performance. This indicates that code-related backbones provide better initial representations in the JIT defect prediction task, even though both Commit code and Commit message are treated as token types.

TABLE IV
TRAINING TIME AND MODEL SIZE OF DIFFERENT JIT DEFECT PREDICTION MODELS

Model	**QT**	**Openstack**	**QT**	**Openstack**
$RoBERTaJIT$	1938s	1005s	2.56 GB	804 MB
$CodeBERTJIT$	1942s	1007s	2.57 GB	821 MB
$GPT2JIT$	2184s	1132s	2.57 GB	815 MB
$CodeGPTJIT$	2150s	1114s	2.57 GB	821 MB
$BARTJIT$	1460s	757s	2.61 GB	863 MB
$PLBARTJIT$	1489s	808s	2.61 GB	860 MB

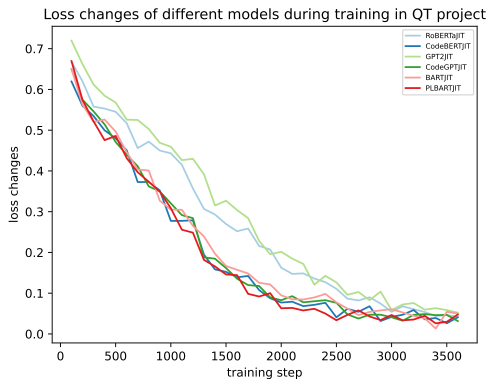


Figure 7. Loss changes during training process on QT project

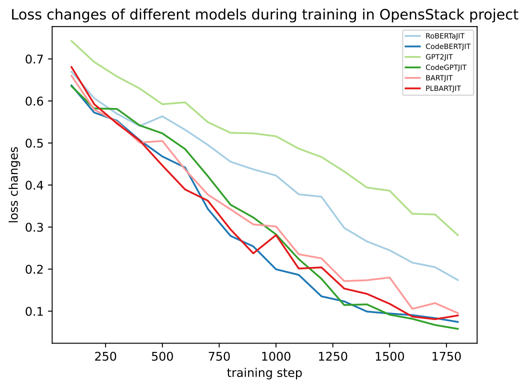


Figure 8. Loss changes during training process on Openstack project

Results 2: Despite the similarity of some backbone models, the distribution of results in JIT defect prediction tasks is significantly different. In RQ2, we explored the relationship between different models in terms of training efficiency, but it remained unclear whether these models exhibited the same distribution in their prediction results. To investigate this further, we conducted T-test experiments between the prediction results of the six models. Table V presents the T-value and P-value for these models, The number on the left represents the T-value between the two models, and the number on the right represents the corresponding p-value. This suggests that despite sharing similar backbone models, the JIT defect prediction models still yield different prediction result distributions in a statistical sense.

Results and analysis on RQ3. In this research question, we aim to explore how the inclusion of Commit code and Commit message as model input affects the performance of defect detection in JIT defect prediction models. To do so, we conduct an ablation experiment where we remove either the Commit code or Commit message from the input data and analyze the impact on recall and precision, which are the most intuitive representations of defect detection and false positives.

Figure 9 displays the experimental results of the six JIT defect prediction models under the ablation settings. Specifically, the "-msg" operation refers to using only Commit code in the model input, while the "-code" operation refers to using only Commit message in the model input. The ablation experiment reveals that Commit code plays a crucial role in defect detection.

Results: Commit code plays the main role in defect detection, both Commit message and Commit code are responsible for prediction precision, accuracy, and F1 score. Recalls mean we correctly predict a defective submission as a defect. When we conduct the "-msg" operation on all six JIT defect prediction models for both QT and Openstack datasets, most of the results show a direct improvement in recall. This indicates that removing the Commit message from the input data helps the models identify more defects correctly. It is worth noting that Commit message, being provided entirely by the developer and not strictly regulated, may introduce information that does not exactly match the code description for a given commit. On the other hand, when we conduct the "-code" operation on the six models, more than half of the results show a decrease in recall value, while other results vary only slightly from the base model results. This further supports the finding that Commit code plays a significant role in defect detection. In addition to the main results, we made an interesting observation during the ablation experiment. Regardless of whether we removed Commit code or Commit message from the model input, the other three evaluation indicators (accuracy, precision, and F1 score) show a decline. This suggests that both parts of the data, Commit code, and Commit message, together play a crucial role in determining the overall performance of these three evaluation indicators. These findings provide valuable insights into the importance of each input data component and can guide researchers in optimizing their models for JIT defect prediction.

Results and analysis on RQ4. In this research question,

TABLE V
T-TEST RESULTS FOR THE DISTRIBUTION OF PREDICTION RESULTS OF DIFFERENT MODELS

Model	RoBERTaJIT	CodeBERTJIT	GPT2JIT	CodeGPTJIT	BARTJIT	PLBARTJIT
RoBERTaJIT	╲	18.41/ 0.0	-8.73/ 0.0	6.48/ 0.0	-11.95/ 0.0	14.88
CodeBERTJIT	╲	╲	-29.21/ 0.0	-13.22/ 0.0	-33.01/ 0.0	-3.21/ 0.001
GPT2JIT	╲	╲	╲	16.38/ 0.0	-3.3/ 0.001	25.01/ 0.0
CodeGPTJIT	╲	╲	╲	╲	-19.98/ 0.0	9.45/ 0.0
BARTJIT	╲	╲	╲	╲	╲	27.6/ 0.0
PLBARTJIT	╲	╲	╲	╲	╲	╲

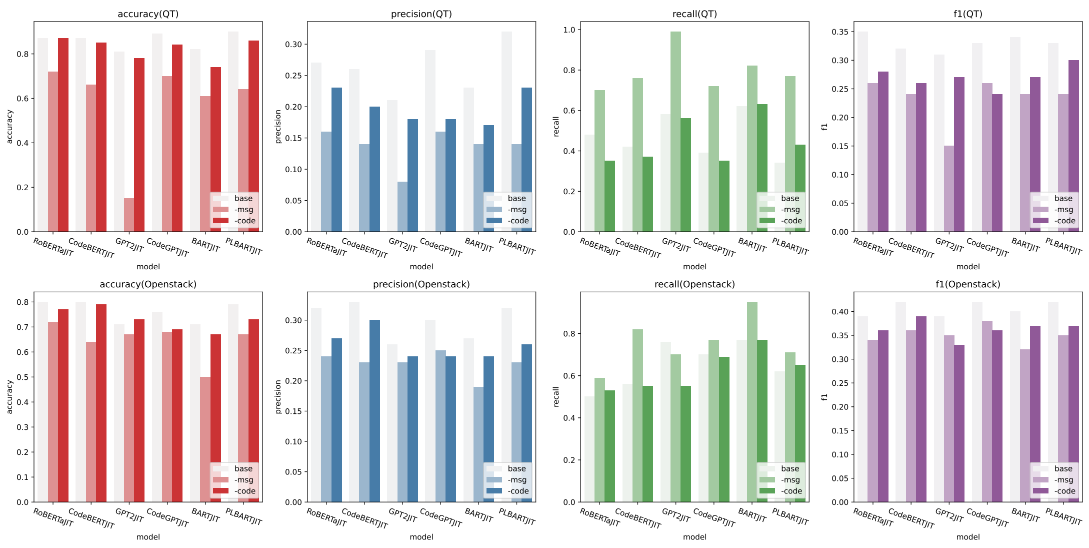


Figure 9. The results of ablation experiments of six models on two QT and Openstack datasets

we aim to explore the performance of JIT defect prediction models with different backbone models in a few-shot scenario with a balanced dataset. To simulate real software engineering projects where defective submissions are less common, we downsampled the two datasets and set different scales (0, 10, 100, 500, 1000, and 2000) to explore the performance of the models. We used the same training setting as in RQ1 and keep the same test datasets. Figure 10, Table VI, and Table VII present the experimental results of these 6 models in the few-shot scenarios. In the zero-shot scenario, the F1 score could not be calculated due to the problem of zero division, so we focused on comparing the model performance under the four evaluation metrics: auc score, accuracy, precision, and recall.

Results 1: With the gradual increase in dataset size, the performance of the six models will tend to be consistent and normal. As shown in Figure 10, we observed that in the zero-shot scenario, the performance of the six models is almost abnormal, with recall and accuracy values taking on extreme values of 0 and 1. This indicates the unavailability of the models in the zero-shot scenario. However, as the dataset size gradually increases, the performance of the six models becomes more consistent and improves significantly. Results 2: Few shot scenarios with balanced data at the right size can bring the capability of defect detection increased. Table VI and Table VII show the results with 2000 training samples. From the results, we can observe interesting patterns when the dataset size shrinks to a fraction of the original datasets

TABLE VI
RESULTS OF QT PROJECT IN 2000-SHOT SCENARIO

Model	auc score	accuracy	precision	recall
RoBERTaJIT	0.81	0.87	0.27	0.48
2000-shot	0.80	0.61	0.14	0.82
CodeBERTJIT	0.81	0.87	0.26	0.42
2000-shot	0.79	0.74	0.17	0.65
GPT2JIT	0.81	0.81	0.21	0.58
2000-shot	0.80	0.75	0.18	0.69
CodeGPTJIT	0.81	0.89	0.29	0.39
2000-shot	0.81	0.73	0.18	0.75
BARTJIT	0.82	0.82	0.23	0.62
2000-shot	0.79	0.63	0.14	0.81
PLBARTJIT	0.82	0.90	0.32	0.34
2000-shot	0.80	0.58	0.13	0.86

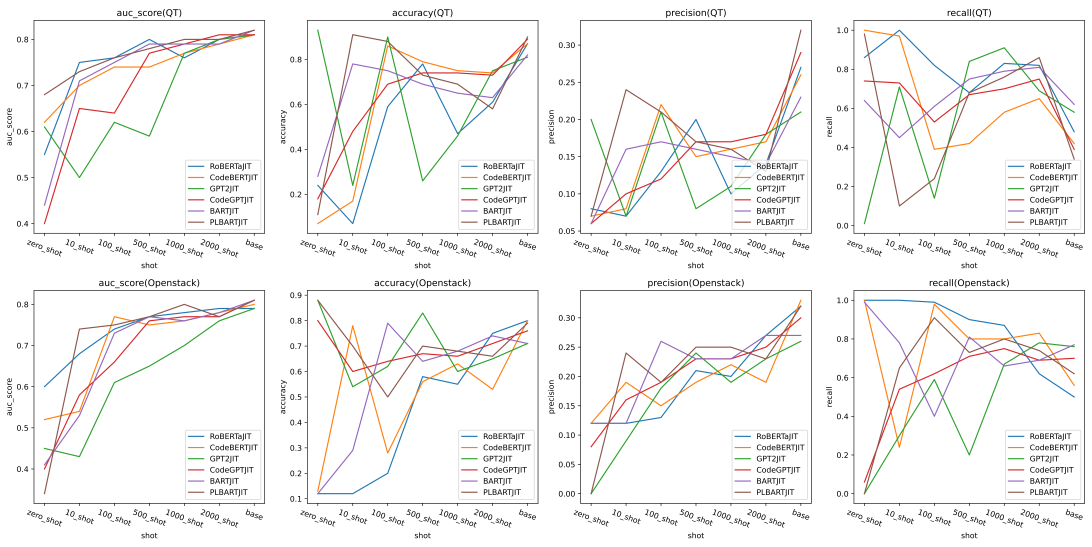


Figure 10. Model performance of different models in few-shot scenario

TABLE VII
RESULTS OF OPENSTACK PROJECT IN 2000-SHOT SCENARIO

Model	auc score	accuracy	precision	recall
RoBERTaJIT	0.79	0.80	0.32	0.50
2000-shot	0.79	0.75	0.27	0.62
CodeBERTJIT	0.80	0.80	0.33	0.56
2000-shot	0.78	0.53	0.19	0.83
GPT2JIT	0.79	0.71	0.26	0.76
2000-shot	0.76	0.65	0.23	0.78
CodeGPTJIT	0.81	0.76	0.30	0.70
2000-shot	0.77	0.71	0.25	0.69
BARTJIT	0.81	0.71	0.27	0.77
2000-shot	0.78	0.74	0.27	0.69
PLBARTJIT	0.81	0.79	0.32	0.62
2000-shot	0.77	0.66	0.23	0.74

TABLE VIII
EXPERIMENTAL RESULT OF CODEGPTJIT IN 2000-SHOT SCENARIO ON QT PROJECT

Model	auc score	accuracy	precision	recall	F1 score
DeepJIT	0.79	0.73	0.17	0.70	0.27
CC2Vec	0.81	0.56	0.13	0.87	0.23
CodeGPT (2000-shot)	**0.81**	**0.73**	**0.18**	0.75	**0.29**

TABLE IX
EXPERIMENTAL RESULT OF GPT2JIT IN 2000-SHOT SCENARIO ON OPENSTACK PROJECT

Model	auc score	accuracy	precision	recall	F1 score
DeepJIT	0.74	0.61	0.17	0.76	0.28
CC2Vec	0.76	0.27	0.11	0.98	0.19
GPT2JIT (2000-shot)	**0.76**	**0.65**	**0.23**	0.78	**0.36**

(0.07 times for QT and 0.15 times for Openstack). While the difference in AUC scores between the models becomes less obvious, ranging from 0 to 0.04, there is a significant difference in accuracy and recall. The significant decrease in accuracy when training the models on a much smaller dataset can be attributed to the convergence of the model. Training on a smaller dataset with the same epoch leads to a less stable convergence compared to training on larger datasets. However, despite the challenges posed by the smaller dataset size, we find that training the models on such limited data can still improve the recall rate significantly. On average, the recall rate improves by 36% in the QT dataset and 14% in the Openstack dataset. This finding suggests that when using a balanced dataset, JIT defect prediction models based on pre-trained models can effectively detect defects even in few-shot scenarios.

We observed that some models achieved lower evaluation scores compared to the base model in this question. However, it is interesting to note that despite lower scores, CodeGPTJIT and GPT2JIT still outperformed the two state-of-the-art models, DeepJIT and CC2Vec, under the few-shot scenario. This is evident from the results presented in Table VIII and Table IX for CodeGPTJIT and GPT2JIT on QT and Openstack datasets, respectively. CodeGPTJIT and GPT2JIT demonstrate optimal performance on QT and Openstack datasets, respectively, indicating their effectiveness in handling few-shot scenarios. These results highlight the potential of fine-tuning based on pre-trained models in situations where training data is limited.

V. RELATED WORK

Finding defects in time can reduce the cost of repairing defects and improve the quality of software. Researchers have proposed a large number of defect prediction techniques, which mainly include module-level, file-level, and change-level defect prediction based on different prediction granularity. In 2008, Kim [1] proposed a code classification method using machine learning, i.e., codes are classified into bug and bug-free categories, and the recently submitted codes are classified with or without defects. In 2013, Kamei officially named the technique predicting defects based on software change characteristics as JIT defect prediction.

The general process of JIT defect prediction technology usually includes data annotation, feature extraction, model construction, and experimental evaluation. Data annotation is used to determine whether code changes introduce defects. Sliwerski et al. first proposed the algorithm to identify the changed code introducing defects in 2005, namely the SZZ algorithm[36]. The algorithm can track the submission history of changes through the version control system and perform the identification of defects. Later, many researchers improved the algorithm. For example, the Annotation Graph SZZ (AG-SZZ) algorithm solved the problem of introducing non-defective code, such as blank lines and comments, to avoid identifying these codes as introduced defects [37]. Based on the AG-SZZ algorithm, the meta-change SZZ (MA-SZZ) algorithm solves the problem of incorrectly identifying the branch change as a defect[38]. Refactoring Aware SZZ (RA-SZZ) algorithm solves the problem of incorrectly identifying code refactoring as a defect introduced into the code[39].

The features used in JIT defect prediction mainly include traditional features extracted manually and features extracted by deep learning methods. Mockus et al. proposed the selection of manually extracted features for the first time and divided them into different categories of features[2], including changes in lines of code, number of files modified, and developer's experience. Some subsequent studies have extended the traditional artificial features by adding features such as code complexity, word frequency of components such as logs and code, and the difference in node number of abstract syntax tree before and after modification of relevant code files to represent changes[1][7–12].

Before 2015, the research on software defects was coarse-grained. With Hinton's research in the field of deep learning[13–15], the trend of cross-fusion among various fields became more obvious. Depending on the development of deep learning technology in recent years[16–19], the proposal of various network structures provides new ideas for feature extraction. In 2015, Yang et al. proposed a method that uses deep learning technology to improve the performance of JIT defect prediction. In this study, Deep Belief Network(DBN) was used to process and map the dimensions of manually extracted features. It was the first case using deep learning in JIT defect prediction technology[40]. In traditional JIT defect prediction technology, researchers often connect the extracted feature with a classifier to construct the model. The classifier is usually a common algorithm in machine learning, such as random forest, support vector machine(SVM), logistic regression(LR), etc.[20, 21]. However, all the studies mentioned above are based on traditional manual features. In recent years, with the development of natural language processing, researchers realize that automatic feature extraction can be carried out from the code and submitted information itself[22][23].

Many researchers have found that in a submission, the commit information provided by the developer is natural language, while the code is a semi-structured language and also has naturalness. These features naturally provide the relevance of JIT defect prediction techniques to natural language processing. Due to the proposed network structures such as convolutional neural network (CNN)[16] and recurrent neural network (RNN)[41], feature extraction in JIT defect prediction technology is not limited to the traditional features extracted. Hoang proposed an end-to-end JIT defect prediction framework, DeepJIT. This framework uses CNN to extract features from Commit message and Commit code in a submission, and then inputs the fused features into a layer full connection for classification, so as to achieve JIT defect prediction without manually extracting features[22].

In 2020, Hoang proposed a distributed vector representation of code change called CC2Vec based on GRU and hierarchical attention Network[34]. CC2Vec vector was obtained by using information of submitted logs as training labels[23]. Both of these JIT defect prediction models using deep learning methods performed well. In view of the excellent performance of CC2Vec, Zeng et al. made a more detailed analysis and comparison between DeepJIT and CC2Vec, and expanded the dataset on the original basis. While confirming the performance of CC2Vec, they also analyzed its defects, and proposed a logistic regression classifier called LApredict. It is confirmed that deep learning methods still have a lot improvement to make[42].

At present, the research of deep JIT defect prediction is still active. The research of embedding model and pre-trained model in natural language field makes it possible to apply many new network structures and models to JIT defect prediction. In 2017, Google proposed Transformer, a neural network structure, which provides a full attention mechanism without using CNN and RNN[33]. Devlin et al. proposed BERT pre-trained model using Transformer structure[24]. Liu et al. improved the BERT model, optimized its parameter selection, expanded the BERT model corpus, and proposed the RoBERTa model[32]. CodeBERT, proposed by Feng et al., provided a bi-modal pre-trained model for both natural and programming languages, which can be used for downstream tasks in software engineering and natural language[25]. GPT[31] developed by OpenAI is based on the Transformer decoder and utilizes unsupervised learning to pre-train on a large corpus of text data. GPT is capable of generating human-like text and can be fine-tuned for various natural language processing tasks such as text completion, text generation, and machine translation. Then researchers extend GPT model with much more natural and programming languages corpus which can assist developer in code completion, code summarization, and other code-related tasks. CodeGPT leverages the strengths of GPT to understand the structure and syntax of programming languages and generate code snippets that match the input context. As another transformer-based style model, BART[44] is a sequence-to-sequence model architecture introduced by Facebook AI Research. It combines the strengths of bidirectional and auto-regressive models, allowing it to be used for both text generation and text comprehension tasks. PLBART[45] is an extension of the BART model designed to facilitate efficient parallel training. It allows for the pre-training of large language models on multiple GPUs, leading to faster convergence and improved performance. PLBART make great improvement in both natural and programming language-related task. These advanced language models leverage the Transformer architecture and have been pre-trained on large datasets. Although much bigger model like ChatGPT, LLaMa, and so on have emerged, these model applied to easy task is still important and significant.

VI. Conclusions and future work

Our work mainly focuses on the task of JIT defect prediction based on a pre-trained model. We first construct six models: RoBERTaJIT, CodeBERTJIT, BARTJIT, PLBARTJIT, GPT2JIT, and CodeGPTJIT. Specifically, we investigate the performance of the models when using Commit code and Commit message as inputs, as well as the relationship between training efficiency and model distribution among these six models. Additionally, we conduct an ablation experiment to explore the sensitivity of each model to inputs. Furthermore, we investigate how the models perform in few-shot scenarios and obtain two models which can outperform DeepJIT and CC2Vec in QT and Openstack datasets, respectively. The findings showed that each model based on different backbones exhibited improvements. Models with similar backbone pre-training showed predictions with closer distributions. Commit code played a significant role in defect detection, and balanced datasets improved defect prediction ability under few-shot scenarios. These results provide new insights for optimizing JIT defect prediction tasks using pre-trained models and highlight the factors that require more attention when constructing such models.

We have also identified some future works. Based on the results of the experiments, we can find that the Commit code largely determines the defect prediction ability of all models with different pre-trained models as the backbone. This provides an idea for our future work, that is, different treatments of Commit code can be adopted in model optimization, which may bring higher benefits. And another important finding is that a balanced dataset can bring improvement in defect prediction capacity under a few-shot scenario. This means that we can further optimize our model in a few-shot scenarios.

References

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

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

[3] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[4] L. Hatton, "Reexamining the fault density-component size connection," *IEEE Software*, vol. 14, no. 2, pp. 89–97, 1997.

[5] Mockus and Votta, "Identifying reasons for software changes using historic databases," in *Proc of International Conference on Software Maintenance*, 2000, pp. 120–130.

[6] N. F. Schneidewind and H. M. Hoffmann, "An experiment in software error data collection and analysis," *IEEE Trans Software Engineering*, vol. 5, no. 3, pp. 276–286, 1979.

[7] 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, 2015, pp. 99–108.

[8] 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.

[9] S. McIntosh and Y. Kamei, "[journal first] are fix-inducing changes a moving target?: A longitudinal case study of just-in-time defect prediction," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 560–560.

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

[11] T. M. Khoshgoftaar and E. B. Allen, "Ordering fault-prone software modules," vol. 11, no. 1, pp. 19–37, 2003.

[12] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[13] Hinton and E. Geoffrey, "How neural networks learn from experience." *Scientific American*, vol. 267, no. 3, pp. 144–51, 1992.

[14] Y. Bengio, 2009.

[15] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[16] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, no. 2, 2012.

[17] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *JMLR.org*, 2015.

[18] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition*

(CVPR), 2016, pp. 770–778.

[20] C. Pornprasit and C. K. Tantithamthavorn, "Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 369–379.

[21] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 82–101, 2022.

[22] T. Hoang, H. Khanh 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)*, 2019, pp. 34–45.

[23] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 518–529.

[24] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "Bert: pre-trained of deep bidirectional transformers for language understanding," 2018.

[25] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, and D. Jiang, "Codebert: A pre-trained model for programming and natural languages," 2020.

[26] T. Mikolov, I. Sutskever, C. Kai, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *arXiv*, 2013.

[27] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *Computer Science*, 2013.

[28] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Conference on Empirical Methods in Natural Language Processing*, 2014.

[29] Q. Liu, M. J. Kusner, and P. Blunsom, "A survey on contextual embeddings," 2020.

[30] M. Peters, M. Neumann, M. Iyyer, M. Gardner, and L. Zettlemoyer, "Deep contextualized word representations," 2018.

[31] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-trained," 2018.

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

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *arXiv*, 2017.

[34] N. Pappas and A. Popescu-Belis, "Multilingual hierarchical attention networks for document classification," 2017.

[35] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 425–436.

[36] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*, 2005.

[37] S. Kim, T. Zimmermann, P. Kai, and E. Jr, "Automatic identification of bug-introducing changes," in *Proc International Conference on Automated Software Engineering*, 2006.

[38] D. Costa, S. Mcintosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING SE*, vol. 43, no. 7, pp. 641–657, 2017.

[39] E. C. Neto, D. Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *IEEE International Conference on Software Analysis*, 2018.

[40] 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*, 2015, pp. 17–26.

[41] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *Eprint Arxiv*, 2014.

[42] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: how far are we?" in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[43] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[44] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," 2019.

[45] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," 2021.

[46] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 608–620.

[47] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020.