DeepJIT: A Deep Learning Framework for *Just-In-Time* Defect Prediction

ABSTRACT

Most software Quality Assurance (SOA) resources often focus on software modules that are likely to be defective in the future to help developers saving their effort to debug a program. To solve this problem, change-level defect prediction or Just-in-time (JIT) defect prediction is proposed to identify bug in the code changes. JIT models are trained using machine learning techniques which assume that historical changes are similar to future one. Hence, these changes can be used to identify defect-prone software modules (e.g., functions, files, system, etc.). A previous approach relies on manually extracted code changes features. This approach, however, shows only moderate accuracy. In this paper, we propose a novel deep learning framework that is automatically extracting features from commit message and code changes and using them to identify bugs. Our framework takes into account the hierarchical structure of code changes to produce their features. Experiments on two well-known projects (i.e., QT and OPENSTACK) shows that our proposed approach outperforms the state-of-the-art baseline in term of the area under the receiver operator characteristics Curve (AUC).

ACM Reference Format:
. 1997. DeepJIT: A Deep Learning Framework for *Just-In-Time* Defect Prediction. In *Proceedings of ACM Woodstock conference (WOODSTOCK'97).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

As software systems are becoming the backbone of our economy and society, defects existing in those systems may substantially affect businesses and people's lives in many ways. For example, Knight Capital[1], a company which executes automated trading for retail brokers, lost $440 millions in only one morning in 2012 due to an overnight faulty update to its trading software. A flawed code change, introduced into OpenSSL's source code repository, caused the infamous Heartbleed[2] bug which affected billions of Internet users in 2014. As software grows significantly in both size and complexity, finding defects and fixing them become increasingly difficult and costly.

One common best practice for cost saving is identifying defects and fixing them as early as possible, ideally before new code changes (i.e. *commits*) are introduced into codebases. Emerging research has thus developed *Just-In-Time* (JIT) defect prediction models and techniques which help software engineers and testers to quickly narrow down the most likely defective commits to a software codebase [? ?]. JIT defect prediction tools helps provide early feedback to software developer, and prioritize and optimize effort for inspection and (regression) testing, especially when facing with deadlines and limited resources. They have therefore been integrated into the development practice at large software organizations such as Avaya [35], Blackberry [42], and Cisco [44].

Machine learning techniques have been widely used in existing work for building JIT defect prediction models. A common theme of existing work (e.g. [20, 22, 27, 35?]) is carefully crafting a set of features representing a code change, and using them as defectiveness predictors. Those features are mostly derived from the properties of code changes, such as the change size (e.g. lines added or deleted), the change scope (e.g. the number of files or directories modified), the history of changes (e.g. the number of prior changes to the updated files), track record of the author and code reviewers, and the activeness of the code review of the change. Those features are then used by a traditional classifier (e.g. Random Forests or Logistic Regression) to predict the defectiveness of code changes A recent work [46] used a deep learning model (i.e. Deep Belief Network) to improve the performance of JIT defect prediction models. Their approach does not, however, leverage the true notions of deep learning. They still used the same set of features that are manually engineered as in previous work, and their model is *not* end-to-end trainable.

The metric-based features however do not represent the semantic and syntactical structure of the actual code changes. In many cases, two different code changes which have the exactly same metrics (e.g. the number of lines added and deleted) may generate different behaviour when executed, and thus have a different likelihood of defectiveness. Previous studies have showed the usefulness of harvesting the syntactical structure and semantic information hidden in source code to perform various software engineering tasks such as code completion, bug detection and defect prediction [? ? ? ? ?]. This information may enrich representations for defective code changes, and thus improve JIT defect prediction.

In this paper, we present a new JIT defect prediction model (namely DeepJIT) which leverages the powerful deep learning Convolution Neural Network (CNN) architecture to learn a deep representation of commits. Our model processes both a commit message (in natural language) and the associated code changes (in programming languages) and automatically semantic features which represent the "meaning" of the commit. This approach removes software practitioners from manually designing and extracting features, as done in previous work. DeepJIT is a fully end-to-end trainable system where raw data signals (e.g. words or code tokens) are passed from input nodes up to the final output node for predicting defectiveness, and prediction errors are back propagated from the output

[1]https://dealbook.nytimes.com/2012/08/02/knight-capital-says-trading-mishap-cost-it-440-million/
[2]http://heartbleed.com

node down to the input layer. TODOXXX: Results are summarized here ....

2 MOTIVATION

2.1 An example of a commit

2.2 Convolutional Neural Networks

One of the most powerful forms of deep learning neural networks is the Convolutional Neural Network (CNN) [32]. CNNs are widely used to solve image pattern recognition problems and have been achieved significant results [21, 29, 31]. Like traditional deep learning networks, CNNs receive an input and perform a product operation followed by a nonlinear function. The last layer is the output layer containing objective functions [48] associated with the labels of the input.

Figure 1 illustrates a simple CNN for classification task. The simple CNN includes an input layer, a convolutional layer, followed by the rectified linear unit (RELU) which is a nonlinear activation function, a pooling layer, a fully-connected layer, and an output layer in the following paragraphs.

The input layer takes an input as 2-dimensional array or 3-dimensional array and passes it through a of convolution layers.

The convolutional layer plays a vital role in CNN and it takes advantage of the use of learnable filters. These filters are small in spatial dimensionality, but they are applied along the entirety of the depth of the input data. For example, given an input data $\mathbf{I} \in \mathbb{R}^{\mathrm{H}\times\mathrm{W}\times\mathrm{D}}$ and a filter $\mathbf{K} \in \mathbb{R}^{\mathrm{h}\times\mathrm{w}\times\mathrm{D}}$, we produce a new activation map $\mathbf{A} \in \mathbb{R}^{(\mathrm{H}-\mathrm{h})\times(\mathrm{W}-\mathrm{w})\times 1}$. Figure 2 presents a visual representation of a convolutional layer in CNN. The RELU is then applied to each value of the activation map as follows:

$$f(x) = max(0, x) \tag{1}$$

The pooling layer aims to reduce the dimensionality of the activation map, and further reduce the number of parameters and the computational helping to control overfitting problem [45]. The pooling layer spreads along the activation map and scales its dimensionality. There are three different types of pooling layers:

- Max pooling takes the largest element from each region of the activation map.
- Average pooling constructs the average value from each region of the activation map.
- Sum pooling sums all the elements from each region of the activation map.

Figure 1

3 APPROACH

In this section, we first formulate the *Just-In-Time* (JIT) defect prediction and provide an overview of our framework. We then describe the details of each part inside the framework. Finally, we present an algorithm for learning effective settings of our model's parameters.

3.1 Framework Overview

The goal of the just-in-time defect prediction model is to automatically label a commit change as bug or clean to help developers better focus on their efforts on assuring software quality. We consider the just-in-time defect prediction problem as a learning task to construct prediction function $\mathbf{f} : \mathcal{X} \longmapsto \mathcal{Y}$, where $y_i \in \mathcal{Y} = \{0, 1\}$ indicates whether a commit change $x_i \in \mathcal{X}$ cleans ($y_i = 0$) or contains a buggy code ($y_i = 1$). The prediction function $\mathbf{f}$ can be learned by minimizing the following objective function:

$$\min_{\mathbf{f}} \sum_i \mathcal{L}(\mathbf{f}(x_i), y_i) + \lambda\Omega(\mathbf{f}) \tag{2}$$

where $\mathcal{L}(.)$ is the empirical loss function measuring the difference between the predicted and the output label, $\Omega(\mathbf{f})$ is a regularization function to prevent the over fitting problem, and λ the trade-off between $\mathcal{L}(.)$ and $\Omega(\mathbf{f})$. Figure 3 illustrates the overview framework of the just-in-time defect prediction model. The model consists of four parts: input layer, feature extraction layer, feature combination layer, and the output layer. We explain the details of each part in the following subsections.

3.2 Input Layer

To feed the raw textual data to convolutional layers for feature learning, we first encode a commit message and code changes in the input layer. We represent each word in the commit message and code changes as d-dimensional vector. After the preprocessing step, the $\mathcal{X}_i^m$ and $\mathcal{X}_i^c$, which are the encoded data of the commit message and code changes respectively, are passed to the convolutional layers to extract the commit message and code changes features. In the convolutional layers, the commit messages and code changes are processed independently to extract the features based on each type of textual information. These features from the commit messages and code changes are then combined into a unified feature representation, and followed by a linear hidden layer connected to output layer used to produce the output label $\mathcal{Y}$ indicating whether the commit change x_i cleans or contains a buggy code.

The novelty of the just-in-time defect prediction model lies in the convolutional network layers for code changes and the feature combination layers. In the following subsection, we firstly discuss the convolutional layers for the commit message and present the novelty of our model in more details.

3.3 Convolutional Network Architecture for Commit Message

The underlying deep neural network for commit message is a Convolutional Neural Network (CNN). CNN firstly used to automatically learn the salient features in the images from raw pixel values [29]. However, CNN has been also used a lot and showed extraordinary successes in Natural Language Processing (NLP) [8, 18, 19, 24, 47]. The architecture of CNN allowed it to extract the structural information features from raw text data of word embedding. Next, we describe how a simple CNN can be used to learn the commit message's features.

Given a commit message $\mathbf{m}$ which is essentially a sequence of words $[w_1, \ldots, w_{|m|}]$. We aim to obtains its matrix representation $\mathbf{m} \rightarrow \mathbf{M} \in \mathbb{R}^{|m|\times d_m}$, where the matrix $\mathbf{M}$ comprises a set of words $w_i \longrightarrow W_i$, $i = 1, \ldots, |m|$ in the given commit message. Each word w_i now is represented by an embedding vector, i.e., $W_i \in \mathbb{R}^{d_m}$,

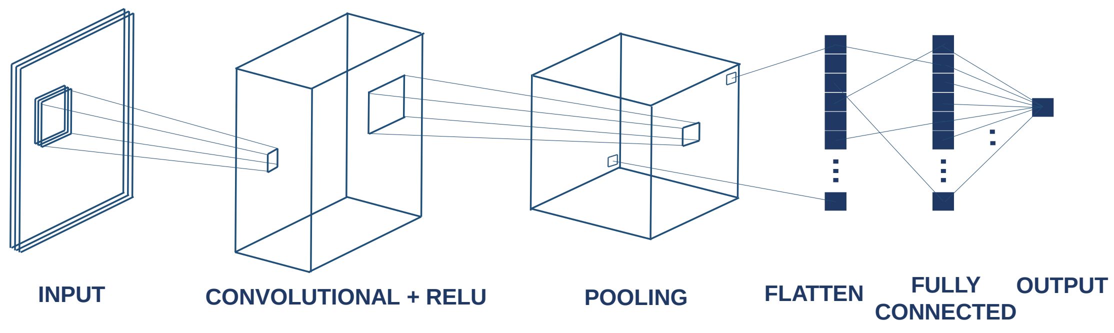


Figure 1: A simple convolutional neural network architecture.

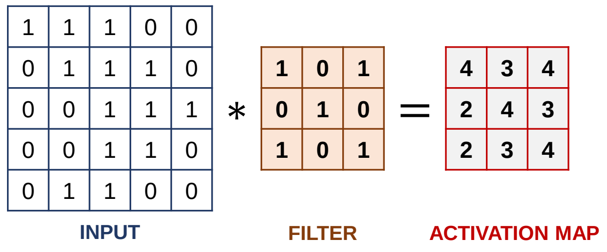


Figure 2: An example of a convolutional layer in CNN.

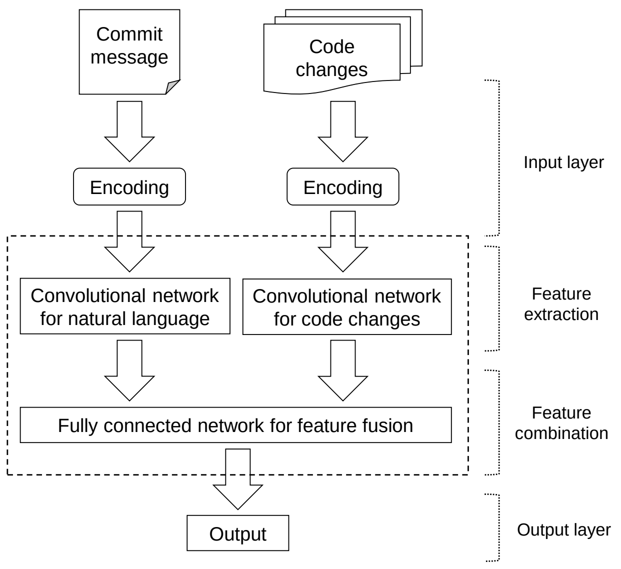

Commit message
Code changes
Encoding
Encoding
Convolutional network for natural language
Convolutional network for code changes
Fully connected network for feature fusion
Output
Input layer
Feature extraction
Feature combination
Output layer

Figure 3: The general framework of just-in-time defect prediction model.

where d_m is a d_m-dimensional vector of a word appearing in the commit message.

Following the previous works [24, 47], the d_m-dimensional representing an embedding vector extracted from an embedding matrix which is randomly initialized and jointly learned with the CNN model. In our paper, the embedding matrix of commit message is randomly initialized and learned during the training process. Hence, the matrix representation $\mathbf{M}$ of the commit message $\mathbf{m}$ with a sequence of $|m|$ words can be represented as follows:

$$\mathbf{M} = [W_1, \ldots, W_{|m|}] \tag{3}$$

For the purpose of parallelization, all commit messages are padded or truncated to the same length $|m|$.

To extract the commit message's salient features, a filter $f \in \mathbb{R}^{k \times d_m}$, followed by a non-linear activation function $\alpha(.)$, is applied to a window of k words to produce a new feature as follows:

$$c_i = \alpha(f * M_{i:i+k-1} + b_i) \tag{4}$$

where $*$ is a sum of element-wise product, and $b_i \in \mathbb{R}$ is the bias value. In our paper, we choose the rectified linear unit (RELU) as our activation function since it achieved a better performance compared to other activation functions [7, 13, 36]. The filter f is applied to every k-words of the commit message, these outputs of this process are then concatenated to product output vector $\mathbf{c}$ such that:

$$\mathbf{c} = [c_1, \ldots, c_{|m|-k+1}] \tag{5}$$

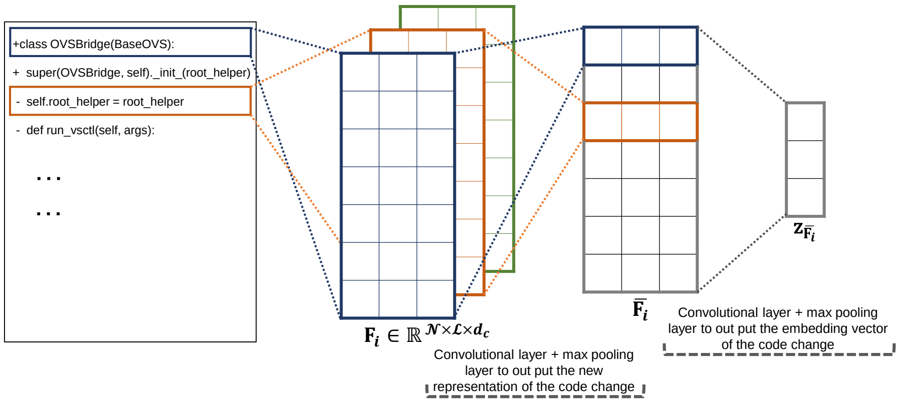
By applying the filter f on every k-words of the commit message, the CNN is able to exploit the semantic information of its input. In practice, the CNN model may include multiple filters with different k. These hyperparameters need to be set by the user before starting the training process. To characterize the commit message, we apply a max pooling operation [32] over the output vector $\mathbf{c}$ to obtain the highest value as follows:

$$\max_{1 \le i \le |m|-k+1} c_i \tag{6}$$

The results of the max pooling operation from each filter are then used to form an embedding vector (i.e., $\mathbf{z_m}$) of the commit message (see Figure 3).

3.4 Convolutional Network Architecture for Code Changes

In this section, we focus on building convolutional networks for code changes to solve the just-in-time defect prediction problem. Code change, although it can be viewed as a sequence of words, differs from natural language mainly because of its structure. The natural language carries sequences of words, and the semantics of the natural language can be inferred from a bag of words [38]. On the other hand, the code change includes a change in different files and different kinds of changes (removals or additions) for each



+class OVSBridge(BaseOVS):
+ super(OVSBridge, self)._init_(root_helper)
- self.root_helper = root_helper
- def run_vsctl(self, args):
…
…
$\mathbf{F}_i \in \mathbb{R}^{\mathcal{N}\times\mathcal{L}\times d_c}$
Convolutional layer + max pooling layer to out put the new representation of the code change
$\overline{\mathbf{F}}_i$
Convolutional layer + max pooling layer to out put the embedding vector of the code change
$\mathbf{z}_{\overline{\mathbf{F}}_i}$

Figure 4: The overall structure of convolutional neural network for each change file in code change. The first convolutional and pooling layers use to learn the semantic features of each added or removed code line based on the words within the added or removed line, and the subsequent convolutional and pooling layers aim to learn the interactions between added or removed code line with respect to the code change structure. The output of the convolutional neural network is the embedding vector $\mathbf{z}_{\overline{\mathbf{F}}_i}$ representing the salient features of the each change file.

file. Hence, to extract salient features from the code changes, the convolutional networks should obey the code changes structure. Based on the aforementioned considerations, we propose novel neural networks for extracting silent features from code changes based on convolutional neural networks.

Given a code change C including a change in different source code files $[\mathrm{F}_1, \ldots, \mathrm{F}_n]$, where n is a number of files in the code change, we aim to extract salient features for each different file F_i. The salient features of each file are then concatenated to each other to represent the features for the given code change. In the rest of this section, we explain how the convolutional networks can extract the salient features for each file in the code change and how these salient features are concatenated.

Suppose F_i represents a change in each different file, F_i contains a number of lines (removals or additions) in a code change file. We also have a sequence of words in each line in F_i. Similar to the section 3.3, we first aim to obtain its matrix representation $\mathrm{F}_i \longrightarrow \mathbf{F}_i \in \mathbb{R}^{\mathcal{N}\times\mathcal{L}\times d_c}$, where $\mathcal{N}$ is the number of lines in a code change file, $\mathcal{L}$ presents a sequence of words in each line, and d_c is a d_c-dimensional vector of a word appearing in the F_i. For the purposed of parallelization, all the source code files are padded or truncated to the same $\mathcal{N}$ and $\mathcal{L}$.

For each line $\mathcal{N}_i \in \mathbb{R}^{\mathcal{L}\times d_c}$, we follow the convolutional network architecture for commit message described in section 3.3 to extract its embedding vector, called $\mathbf{z}_{\mathcal{N}_i}$. The embedding vector $\mathbf{z}_{\mathcal{N}_i}$ aims to learn the salient features or the semantic of a code line based on the words within the code line. These features $\mathbf{z}_{\mathcal{N}_i}$ are then stacked to produce the new representation of the code change file F_i as follows:

$$\overline{\mathbf{F}}_i = [\mathbf{z}_{\mathcal{N}_1}, \ldots, \mathbf{z}_{\mathcal{N}_{|\mathcal{N}|}}] \tag{7}$$

We again apply the convolutional layer and pooling layer on the new representation of the code change (i.e., $\overline{\mathbf{F}}_i$) to extract its embedding vector, namely $\mathbf{z}_{\overline{\mathbf{F}}_i}$. The $\mathbf{z}_{\overline{\mathbf{F}}_i}$ aims to learn the salient features or the semantics conveyed by the interactions between added or removed lines. Figure 4 presents an overall convolutional network architecture for each change file F_i in code changes. The first convolutional and pooling layers aim to learn a new representation of the file, and the subsequent convolutional and pooling layers aim to extract the salient features from the new representation of the change file.

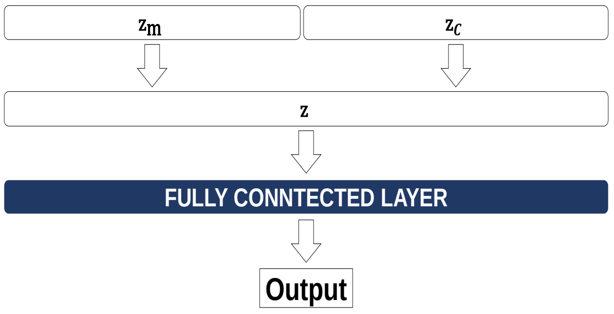


Figure 5: The structure of fully-connected network for feature combination. The embedding vector of commit message $\mathbf{z}_\mathbf{m}$ and code change $\mathbf{z}_C$ are concatenated to generate a single vector (i.e., z).

For each change file $\mathrm{F}_i \in C$, we build its embedding vector $\mathbf{z}_{\overline{\mathbf{F}}_i}$. These embedding vectors are then concatenated to build a new embedding vector representing the salient features of the code change C as follows:

$$\mathbf{z}_C = \mathbf{z}_{\overline{\mathbf{F}}_1} \oplus \cdots \oplus \mathbf{z}_{\overline{\mathbf{F}}_n} \tag{8}$$

where $\oplus$ is the concatenation operator.

3.5 Feature Combination

Figure 5 shows the details of architecture of the feature combination. The inputs of this architecture are the two embedding vectors $\mathbf{z}_\mathbf{m}$ and $\mathbf{z}_C$ which represent the salient features extracted from the commit message and code change, respectively.

These vectors are then concatenated to generate a unified feature representation, i.e., a new vector ($\mathbf{z}$), representing the commit change:

$$\mathbf{z} = \mathbf{z_m} \oplus \mathbf{z}_C \tag{9}$$

The new vector then feed into a fully-connected (FC) layer, which outputs a vector $\mathbf{h}$ as follows:

$$\mathbf{h} = \alpha(\mathbf{w_h} \cdot \mathbf{z} + b_\mathbf{h}) \tag{10}$$

where $\cdot$ is a dot product, $\mathbf{w}_h$ is a weight matrix of the vector $\mathbf{h}$ and the FC layer, $b_\mathbf{h}$ is the bias value, and $\alpha(\cdot)$ is the RELU activation function. The vector $\mathbf{h}$ is passed to an output layer to compute a probability score for a given commit:

Finally, the vector $\mathbf{h}$ is passed to an output layer, which computes a probability score for a given patch:

$$p(y_i = 1|x_i) = \frac{1}{1 + \exp(-\mathbf{h} \cdot \mathbf{w_o})} \tag{11}$$

where $\mathbf{w_o}$ is the weight matrix between the FC layer and the output layer.

3.6 Parameters Learning

In the training process, DeepJIT aims to learn the following parameters: the word embedding matrices of commit messages and commit code in a given commit, the convolutional layers matrices, the weights and bias of the fully connected layer and the output layer.

In the *Just-In-Time* defect prediction, only a few commits contain a buggy code while a large number of commits are clean. Such an imbalance nature increases the difficulty in learning a prediction function [6]. Inspired by [30, 49], we propose an unequal misclassification loss function which helps to reduce the negative influence of the imbalanced data.

Let $\mathbf{w_n}$ and $\mathbf{w_p}$ are the cost of incorrectly associating a commit change and the cost of missing a buggy commit change, respectively. The parameters of DeepJIT can be learned by minimizing the following objective function:

$$\begin{aligned} O &= -\log\left(\prod_{i=1} p(y_i|x_i)\right) + \frac{\lambda}{2}\|\theta\|_2^2 \\ &= -\sum_{i=1}[\mathbf{w_n}(1 - y_i)\log(1 - p(y_i|x_i)) \\ &\quad + \mathbf{w_p} y_i \log(p(y_i|x_i))] + \frac{\lambda}{2}\|\theta\|_2^2 \end{aligned} \tag{12}$$

where $p(y_i|x_i)$ is the probability score from the output layer and θ contains all parameters our model. The term $\frac{\lambda}{2}\|\theta\|_2^2$ is used to mitigate data overfitting in training deep neural networks [5]. We also apply the dropout technique [43] to improve the robustness of our model.

We choose Adam [25], which is a variant of stochastic gradient descent (SGD) [4], to minimize the objective function in the equation 12. We choose Adam due to its computational efficiency and low memory requirements compared to other optimization techniques [1, 2, 25]. To efficiently compute the gradients in linear time (with respect to the neural network size), we use backpropagation [12], which is a simple implementation of the chain rule of partial derivatives.

4 EXPERIMENTS

In this section, we first describe the dataset used in our paper. We then introduce all baselines and evaluation metric. Finally, we present our research questions and results.

4.1 Dataset

TODO: add the information about the dataset

4.2 Baseline

We compared DeepJIT with two other state-of-the-art baselines in the *Just-In-Time* (JIT) defect prediction:

- JIT: The method for identifying fix-inducing code changes was proposed by McIntosh and Kamei [34]. The method used a nonlinear variant of multiple regression modeling [9] to build a classification model for automatically identifying defects in commits. The set of code features, using six families of code change properties, were primarily derived from prior studies [20, 22, 27, 35]. These properties were: the magnitude of change, the dispersion of the changes, the defect proneness of prior changes, the experience of the author, the code reviewers, and the degree of participation in the code review.
- DBN-JIT: The model adopted Deep Belief Network (DBN) [14], one of the state-of-the-art deep learning approaches in performing nonlinear dimensionality reduction, to generate a more expressive feature set from the initial feature set [46]. The generated feature set, a complicated nonlinear combination of the initial features, was put to a machine learning classifier [37] to predict defects in commits.

4.3 Training and hyperparameters

For the size of the convolutional filters, we choose 64. The size of DeepJIT's fully-connected layer described in Section 3.5 is set to 512. The word vectors dimension of the commit message (d_m) and code changes (d_c) are set to 64. We train DeepJIT using Adam [25] with shuffled mini-batches. The batch size is set to 32. We train DeepJIT for 100 epochs. We also apply the early stopping strategy [5**?**] to avoid overfitting problem during the training process. Typically, we stop the training if the value of the objective function (see Equation 12) has been no update in the last 5 epochs. All these hyperparameters in our paper are widely used in the deep learning community [15–17, 41].

4.4 Evaluation Metric

To evaluate the accuracy of *Just-In-Time* (JIT) models, we calculate threshold-independent measures of model performance. Since our dataset is imbalanced data, we avoid using threshold-dependent measures (i.e., precision, recall, or F1) since these measures strongly depend on arbitraily thresholds [11, 39]. Following the previous work [34], we use the Area Under the receiver operator characteristics Curve (AUC) to measure the power of models' discriminatory, i.e., their ability to differentiate between defects or clean commits. AUC computes the area under the curve plotting the true positive rate against the false positive rate, while applying multiple thresholds to determine if a commit is buggy or not. The values of

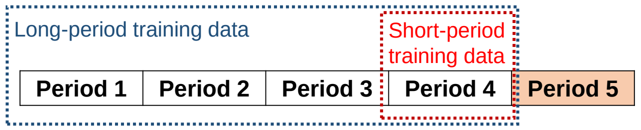


Figure 6: An example of choosing the training data for short-period and long-period models. The last period will be used as testing data.

AUC normally are ranged between 0 (worst discrimination) and 1 (perfect discrimination).

4.5 Research Questions and Results

RQ1: How effective is DeepJIT compared to the state-of-the-art baseline?

To address RQ1, we evaluate how a JIT model, which is trained by a train data, can be used to predict a test data. Typically, we train three types of JIT models:

- **Random models:** To evaluate machine learning algorithm, most people use k-fold cross-validation [26] in which a dataset is randomly divided to k folds, each fold is considered as test data for evaluating JIT model while $k - 1$ folds are considered as train data. In this case, the JIT model is trained on a mixture of past and future data. In our paper, we set $k = 5$.
- **Short-period models:** The JIT model is trained using commits that occurred at one time period. We assume that older commits changes may have characteristics that no longer effects to the latest commits.
- **Long-period models:** Inspired by the work [40], suggesting that larger amounts of training data tend to achieve a better performance in defect prediction problem, we train the JIT model using all commits that occurred before a particular period. We discover whether additional data may improve the performance of the JIT model.

Figure 6 describes how the training data is selected to train short-period and long-period models. We use the last period (i.e., period 5) as a testing data. While the short-period model is trained using the commits that occurred during period 4, the long-period model is trained using the commits that occurred from period 1 to period 4. After training the short-period and long-period model, we measure their performance of these models using AUC described in Section 4.4.

Table 1 shows the AUC results of DeepJIT as well as other baselines in three types of JIT models setting: random, short-period, and long-period. The difference between random models compared to short-period and long-period models is quite small (i.e., below 2.2%) which suggests that there is no difference between training on past or future data. **TODO: Prof. Hoa, do you have any explaination about it?** In the QT project, DeepJIT achieves AUC scores of 0.768, 0.764, and 0.765 in three different JIT settings: random, short-period, and long-period, respectively. Comparing them to the best performing baseline (i.e., DBNJIT), DeepJIT constitutes improvements of 8.96%, 7.00%, and 8.05% in term of AUC. In the OPENSTACK project, DeepJIT also constitutes improvements of 8.21%, 9.08%, and 8.29% in term of AUC compared to DBNJIT (the best performing baseline).

RQ2: Does the proposed model benefit both commit message and the code changes?

To answer this question, we employ an ablation test [28, 33], by ignoring the commit message and the code change in a commit and then evaluate the AUC performance. Specifically, we create two different variants of DeepJIT, namely DeepJIT-Msg and DeeJIT-Code. DeepJIT-Msg only considers commit message information while DeepJIT-Code only uses commit code information. We again use the three types of model settings (i.e., random, short-period, and long period) and the AUC scores to evaluate the performance of our models. Table 2 shows the performance of DeepJIT degrades if we ignore any one of the considered types of information (i.e., commit message or code changes). The AUC scores drop by 19.81%, 28.45%, and 19.01% in the project QT and drop by 33.96%, 16.99%, and 9.01% in the project OPENSTACK on the three types of JIT models if we ignore commit messages. The AUC scores drop by 4.07%, 4.09%, and 5.23% in the project QT and drop by 1.56%, 4.47%, and 3.02% in the project OPENSTACK on the three types of JIT models if we ignore code changes information. It suggests that each kind of information contributes to DeepJIT's performance. Moreover, it also indicates that the code changes are more important to detect defects in a commit than the commit message information.

RQ3: Does DeepJIT benefit from the manually extracted code changes features?

To address this question, we incorporate the code features, derived from [34], into our proposed model. Specifically, the code features, namely $\mathbf{z_r}$, are concatenated with the two embedding vectors $\mathbf{z_m}$ and $\mathbf{z}_C$, representing the salient features of commit message and code change (see Section 3.5), to build a new single vector $\mathbf{z}$ as follows:

$$\mathbf{z} = \mathbf{z_m} \oplus \mathbf{z}_C \oplus \mathbf{z_r} \quad (13)$$

where $\oplus$ is the concatenation operator. Table 3 shows the AUC results of DeepJIT combining with the code features. The AUC scores increase by 1.43%, 3.14%, and 2.75% in the project QT and increase by 1.20%, 4.23%, and 3.63% in the project OPENSTACK on the three types of JIT models settings (i.e., random, short-period, long-period). Compared to the best baseline model (i.e., DBNJIT), DeepJIT constitutes improvements of 10.50%, 10.36%, and 11.02% in the project QT and 9.51%, 13.69%, 12.22% in the project OPENSTACK. It suggests that the manually extracted code features are important and can be used to improve the performance of JIT's models.

RQ4: How are the time costs of DeepJIT?

We train and test DeepJIT on NVIDIA DGX1 with Tesla P100 [10]. Table 4 shows the time costs of DeepJIT in three types of JIT's models (i.e., random, short-period, and long-period) on QT and OPENSTACK. **Prof. Hoa: do you have any idea how to describe the time cost?**

5 THREATS TO VALIDITY

6 RELATED WORK

6.1 JIT Defect Prediction

Some previous studies focus on change-level defect prediction (i.e., JIT defect prediction). For example, Mockus and Weiss [35] predict

Table 1: The AUC results of DeepJIT vs. with other baselines in three types of JIT models: random, short-period, and long-period.

	QT			OPENSTACK		
	Random	Short-period	Long-period	Random	Short-period	Long-period
JIT	0.701	0.703	0.702	0.691	0.711	0.706
DBNJIT	0.705	0.714	0.708	0.694	0.716	0.712
DeepJIT	**0.768**	**0.764**	**0.765**	**0.751**	**0.781**	**0.771**

Table 2: Contribution of feature components in DeepJIT

	QT			OPENSTACK		
	Random	Short-period	Long-period	Random	Short-period	Long-period
DeepJIT-Msg	0.641	0.609	0.638	0.583	0.659	0.689
DeepJIT-Code	0.738	0.734	0.727	0.769	0.738	0.729
DeepJIT	**0.768**	**0.764**	**0.765**	**0.781**	**0.771**	**0.751**

Table 3: Combination of DeepJIT with the code features derived from [34]

	QT			OPENSTACK		
	Random	Short-period	Long-period	Random	Short-Period	Long-period
DeepJIT	0.768	0.764	0.765	0.751	0.781	0.771
DeepJIT-Combined	**0.779**	**0.788**	**0.786**	0.760	**0.814**	**0.799**

Table 4: Time costs of DeepJIT

Dataset	Random		Short-period		Long-period	
	Training time	Testing time	Training time	Testing time	Training time	Testing time
QT	5 hours 43 mins	36.2 mins	17.2 mins	3.2 mins	1 hours 18 mins	8.1 mins
OPENSTACK	12 hours 15 mins	1 hours 6 mins	10.1 mins	2.3 mins	2 hours 37 mins	12.4 mins

commits as being buggy or not in an industrial project. They use metric-based features, such as the number of subsystems touched, the number of files modified, the number of lines of added code, and the number of modification requests. Motivated by their previous work, Kamei et al. [20] built upon the set of code change features, reporting that the addition of a variety of features that were extracted from the Version Control System (VCS) and the Issue Tracking System (ITS) helped to improve the prediction accuracy. They conduct an empirical study of the effectiveness of JIT defect prediction on a set of six open source and five commercial projects and also evaluate their findings when considering the effort required to review the changes.

Aversano *et al.* [3] and Kim *et al.* [23] use source code change logs to predict the risk of a software change.

7 CONCLUSION

REFERENCES

[1] Marios Anthimopoulos, Stergios Christodoulidis, Lukas Ebner, Andreas Christe, and Stavroula Mougiakakou. 2016. Lung pattern classification for interstitial lung diseases using a deep convolutional neural network. *IEEE transactions on medical imaging* 35, 5 (2016), 1207–1216.

[2] Sanjeev Arora, Nadav Cohen, and Elad Hazan. 2018. On the optimization of deep networks: Implicit acceleration by overparameterization. In *35th International Conference on Machine Learning (ICML).* 244–253.

[3] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. 2007. Learning from bug-introducing changes to prevent fault prone code. In *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE'07).* 19–26.

[4] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT).* Springer, 177–186.

[5] Rich Caruana, Steve Lawrence, and C Lee Giles. 2001. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in neural information processing systems.* 402–408.

[6] Nitesh V Chawla, Nathalie Japkowicz, and Aleksander Kotcz. 2004. Special issue on learning from imbalanced data sets. *ACM Sigkdd Explorations Newsletter* 6, 1 (2004), 1–6.

[7] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. 2013. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on.* IEEE, 8609–8613.

[8] Cicero dos Santos and Maira Gatti. 2014. Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers.* 69–78.

[9] John Fox. 1997. *Applied regression analysis, linear models, and related methods.* Sage Publications, Inc.

[10] Nitin A Gawande, Jeff A Daily, Charles Siegel, Nathan R Tallent, and Abhinav Vishnu. 2018. Scaling deep learning workloads: NVIDIA DGX-1/Pascal and intel knights landing. *Future Generation Computer Systems* (2018).

[11] Qiong Gu, Zhihua Cai, Li Zhu, and Bo Huang. 2008. Data mining on imbalanced data sets. In *Advanced Computer Theory and Engineering, 2008. ICACTE'08. International Conference on.* IEEE, 1020–1024.

[12] Martin T Hagan and Mohammad B Menhaj. 1994. Training feedforward networks with the Marquardt algorithm. *IEEE Transactions on Neural Networks* 5, 6 (1994), 989–993.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer*

vision and pattern recognition. 770–778.

[14] Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *science* 313, 5786 (2006), 504–507.

[15] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580* (2012).

[16] Xuan Huo and Ming Li. 2017. Enhancing the Unified Features to Locate Buggy Files by Exploiting the Sequential Nature of Source Code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press, 1909–1915.

[17] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code.. In *IJCAI*. 1606–1612.

[18] Rie Johnson and Tong Zhang. 2014. Effective use of word order for text categorization with convolutional neural networks. *arXiv preprint arXiv:1412.1058* (2014).

[19] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. 2014. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188* (2014).

[20] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Trans. Softw. Eng.* 39, 6 (June 2013), 757–773. https://doi.org/10.1109/TSE.2012.70

[21] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 1725–1732.

[22] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Trans. Softw. Eng.* 34, 2 (March 2008), 181–196. https://doi.org/10.1109/TSE.2007.70773

[23] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Trans. Softw. Eng.* 34, 2 (2008), 181–196.

[24] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[25] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of 3rd International Conference on Learning Representations (ICLR)*.

[26] Ron Kohavi et al. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, Vol. 14. Montreal, Canada, 1137–1145.

[27] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. 2015. Investigating Code Review Quality: Do People and Participation Matter?. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME '15)*. IEEE Computer Society, Washington, DC, USA, 111–120. https://doi.org/10.1109/ICSM.2015.7332457

[28] Bruno Korbar, Andrea M Olofson, Allen P Miraflor, Catherine M Nicka, Matthew A Suriawinata, Lorenzo Torresani, Arief A Suriawinata, and Saeed Hassanpour. 2017. Deep learning for classification of colorectal polyps on whole-slide images. *Journal of pathology informatics* 8 (2017).

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[30] Matjaz Kukar, Igor Kononenko, et al. 1998. Cost-Sensitive Learning with Neural Networks.. In *ECAI*. 445–449.

[31] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. 1997. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks* 8, 1 (1997), 98–113.

[32] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.

[33] Jingzhou Liu, Wei-Cheng Chang, Yuexin Wu, and Yiming Yang. 2017. Deep learning for extreme multi-label text classification. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 115–124.

[34] Shane McIntosh and Yasutaka Kamei. 2018. Are Fix-inducing Changes a Moving Target?: A Longitudinal Case Study of Just-in-time Defect Prediction. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 560–560. https://doi.org/10.1145/3180155.3182514

[35] A. Mockus and D. M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (April 2000), 169–180. https://doi.org/10.1002/bltj.2229

[36] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.

[37] Nasser M Nasrabadi. 2007. Pattern recognition and machine learning. *Journal of electronic imaging* 16, 4 (2007), 049901.

[38] Hwee Tou Ng and John Zelle. 1997. Corpus-based approaches to semantic interpretation in NLP. *AI magazine* 18, 4 (1997), 45.

[39] Giang Hoang Nguyen, Abdesselam Bouzerdoum, and Son Lam Phung. 2009. Learning pattern classification tasks with imbalanced data sets. In *Pattern recognition*. InTech.

[40] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. 2013. Sample size vs. bias in defect prediction. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ACM, 147–157.

[41] Aliaksei Severyn and Alessandro Moschitti. 2015. Learning to rank short text pairs with convolutional deep neural networks. In *Proceedings of the 38th International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 373–382.

[42] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An Industrial Study on the Risk of Software Changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 62, 11 pages. https://doi.org/10.1145/2393596.2393670

[43] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[44] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, Akinori Ihara, and Kenichi Matsumoto. 2015. The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 812–823. http://dl.acm.org/citation.cfm?id=2818754.2818852

[45] Giorgos Tolias, Ronan Sicre, and Hervé Jégou. 2015. Particular object retrieval with integral max-pooling of CNN activations. *arXiv preprint arXiv:1511.05879* (2015).

[46] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS '15)*. IEEE Computer Society, Washington, DC, USA, 17–26. https://doi.org/10.1109/QRS.2015.14

[47] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*. 649–657.

[48] Hang Zhao, Orazio Gallo, Iuri Frosio, and Jan Kautz. 2017. Loss functions for image restoration with neural networks. *IEEE Transactions on Computational Imaging* 3, 1 (2017), 47–57.

[49] Zhi-Hua Zhou and Xu-Ying Liu. 2006. Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transactions on Knowledge and Data Engineering* 18, 1 (2006), 63–77.