2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

DeepJIT: An End-To-End Deep Learning Framework for Just-In-Time Defect Prediction

Thong Hoang[1], Hoa Khanh Dam[2], Yasutaka Kamei[3], David Lo[1], Naoyasu Ubayashi[3]
[1]Singapore Management University, Singapore
[2]University of Wollongong, Australia
[3]Kyushu University, Japan
vdthoang.2016@smu.edu.sg, hoa@uow.edu.au, kamei@ait.kyushu-u.ac.jp,
davidlo@smu.edu.sg, ubayashi@ait.kyushu-u.ac.jp

***Abstract*—Software quality assurance efforts often focus on identifying defective code. To find likely defective code early, change-level defect prediction – aka. *Just-In-Time* (JIT) defect prediction – has been proposed. JIT defect prediction models identify likely defective changes and they are trained using machine learning techniques with the assumption that historical changes are similar to future ones. Most existing JIT defect prediction approaches make use of manually engineered features. Unlike those approaches, in this paper, we propose an end-to-end deep learning framework, named DeepJIT, that automatically extracts features from commit messages and code changes and use them to identify defects. Experiments on two popular software projects (i.e., QT and OPENSTACK) on three evaluation settings (i.e., cross-validation, short-period, and long-period) show that the best variant of DeepJIT (DeepJIT-Combined), compared with the best performing state-of-the-art approach, achieves improvements of 10.36-11.02% for the project QT and 9.51-13.69% for the project OPENSTACK in terms of the Area Under the Curve (AUC).**

I. 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 [1], [2]. JIT defect prediction tools provide early feedback to software developers to 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 [3], Blackberry [4], and Cisco [5].

Machine learning techniques have been widely used in existing work for building JIT defect prediction models. A common theme of existing work (e.g. [3], [6]–[8]) is carefully crafting a set of features to represent a code change, and using them as defectiveness predictors. Those features are mostly derived from properties of code changes, such as change size (e.g. lines deleted or added), change scope (e.g. number of files or directories modified), history of changes (e.g. number of prior changes to the updated files), track record of the author and code reviewers, and activeness of the code review of the change. This set of features can then be used as an input to a traditional classifier (e.g. Random Forests or Logistic Regression) to predict the defectiveness of code changes.

The aforementioned metric-based features however do not represent the semantic and syntactic structure of the actual code changes. In many cases, two different code changes which have exactly the same metrics (e.g. the number of lines deleted and added) may generate different behaviour when executed, and thus have a different likelihood of defectiveness. Previous studies have showed the usefulness of harvesting the semantic information and syntactic structure hidden in source code to perform various software engineering tasks such as code completion, bug detection and defect prediction [9]–[13]. This information may enrich representations for defective code changes, and thus improve JIT defect prediction.

A recent work [14] used a deep learning model (i.e. Deep Belief Network) to improve the performance of JIT defect prediction models. However, their approach does not leverage the true notions of deep learning as they still employ the same set of features that are manually engineered as in previous work, and their model is *not* end-to-end trainable.

To more fully explore the power of deep learning for JIT defect prediction, in this paper, we present a new model (named DeepJIT) which is built upon the well-known deep learning technique, namely Convolutional Neural Network (CNN) [15]. CNN has produced many breakthroughs in Natural Language Processing (NLP) [16]–[20]. Our DeepJIT model processes both a commit message (in natural language, if available) and

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

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00016

the associated code changes (in programming languages) and automatically extracts features which represent the “meaning” of the commit. Unlike commit messages, code changes are more complex as they include a number of deleted and added lines across multiple files. Our model automatically learns the semantic features of each deleted or added line in each changed file. Those features are then aggregated to generate a new representation of the changed file, which is used to construct the features of the code changes in a given commit. This approach removes the need for software practitioners to manually design and extract features, as what was done in previous work [21]. The features extracted from commit messages and code changes are then collectively used to train a model to predict whether a given commit is buggy or not.

The main contributions of our paper include:

- An end-to-end deep learning framework (DeepJIT) to automatically extract features from both commit messages and code changes in a given commit.
- An evaluation of DeepJIT on two software projects (i.e., QT and OPENSTACK). This dataset was originally collected by McIntosh and Kamei to evaluate their proposed technique [21] that we use as one of the baselines. The experiments show the superiority of DeepJIT compared to state-of-the-art baselines.

II. Background

In this section, we first present an example of a buggy change and briefly describe a typical buggy change identification process that is followed by QT and OPENSTACK. We then introduce background knowledge about Convolutional Neural Network (CNN).

A. Buggy Changes and Their Identification

Figure 1 shows an example of a buggy commit in OPENSTACK. The buggy commit contains many pieces of information, i.e., a commit id (line 1), an author name (line 2), a commit date (line 3), a commit message (line 4-10) and a set of code changes (i.e., 11-28). A set of code changes includes changes to multiple files and each file includes a number of deleted and added lines representing the change. In Figure 1, line 16 (starting with −) and lines 17-20 (starting with +) indicate the deleted and added lines of a changed file (namely `3cb5d900c5de_security_groups.py`), respectively. The commit message also plays an important role as a good commit message can help maintainers to speed up the reviewing process and write a good release note.

To review a commit, QT and OPENSTACK use Gerrit [3], which is a code review tool for `git`-based software project. The process of reviewing code changes is as follows:

- Upload change revision: An author of a code changes submits a new change to Gerrit and invites reviewers to comment it.
- Execute sanity tests: Sanity tests verify that the code changes are compliant with the coding style conventions before sending to the reviewers.

[3]https://code.google.com/p/gerrit/

```
1.        commit d60f6efd7f70efba1ccd007d55b1fa740fb98c76
2.        Author: Dan Prince <email address hidden>
3.        Date: Mon Jan 14 12:26:36 2013 -0500
4.             Name the securitygrouprules.direction enum.
5.             Updates to the SecurityGroupRule model and migration so that we
6.             explicitly name the securitygrouprules.direction enum. This fixes
7.             'Postgresql ENUM type requires a name.' errors.
8.
9.             Fixes LP Bug #1099267.
10.            Change-Id: Ia46fe8d4b0793caaabbfc71b7fa5f0cbb8c6d24b
11.       diff --git a/quantum/db/migration/alembic_migrations/versions/3cb5d900c5de
          _security_groups.py
12.       index ff39de84a..cf565af0f 100644
13.       --- a/quantum/db/migration/alembic_migrations/versions/3cb5d900c5de_
          security_groups.py
14.       +++ b/quantum/db/migration/alembic_migrations/versions/3cb5d900c5de_
          security_groups.py
15.       @@ -62,7 +62,10 @@ def upgrade(active_plugin=None, options=None):
16.       -        sa.Column('direction', sa.Enum('ingress', 'egress'), nullable=True),
17.       +        sa.Column('direction',
18.       +                  sa.Enum('ingress', 'egress',
19.       +                          name='securitygrouprules_direction'),
20.       +                  nullable=True),
21.       diff --git a/quantum/db/securitygroups_db.py b/quantum/db/securitygroups_db.py
22.       index 9903a6493..5bd890bbe 100644
23.       --- a/quantum/db/securitygroups_db.py
24.       +++ b/quantum/db/securitygroups_db.py
25.       @@ -62,7 +62,8 @@ class SecurityGroupRule(model_base.BASEV2, models_v2.HasId,
26.       -    direction = sa.Column(sa.Enum('ingress', 'egress'))
27.       +    direction = sa.Column(sa.Enum('ingress', 'egress',
28.       +                                  name='securitygrouprules_direction'))
```

Fig. 1: An example of a buggy commit change in OPENSTACK.

- Solicit peer feedback: The reviewers are asked to examine the code changes after it passes the sanity tests.
- Initiate integration request: Teams are allowed to verify the code changes before integrating it into `git` repositories.
- Execute integration tests: The integration testing system is run to ensure that the code changes that put in the `git` repositories is clean.
- Final integration: After passing the integration testing, Gerrit automatically commits the code changes into the `git` repositories.

B. Convolutional Neural Network

One of the most promising neural networks is the Convolutional Neural Network (CNN) [15]. CNNs have been widely used for many problems (i.e., image pattern recognition, natural language processing, information retrieval, etc.) and demonstrated to achieve promising results [22]–[24]. CNNs receive an input and perform a product operation followed by a nonlinear function. The last layer is the output layer containing objective functions [25] associated with the labels of the input.

Figure 2 illustrates a simple CNN for classification task. The simple CNN includes an input layer, a convolutional layer, followed by the application of the rectified linear unit (RELU) which is a nonlinear activation function, a pooling layer, a fully-connected layer, and an output layer. We briefly explain these layers in the following paragraphs.

The input layer typically takes as an input a 2-dimensional matrix and passes it through a series of convolutional layers. The convolutional layers play 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 an activation map $\mathbf{A} \in \mathbb{R}^{(\mathrm{H}-\mathrm{h}) \times (\mathrm{W}-\mathrm{w}) \times 1}$. The RELU

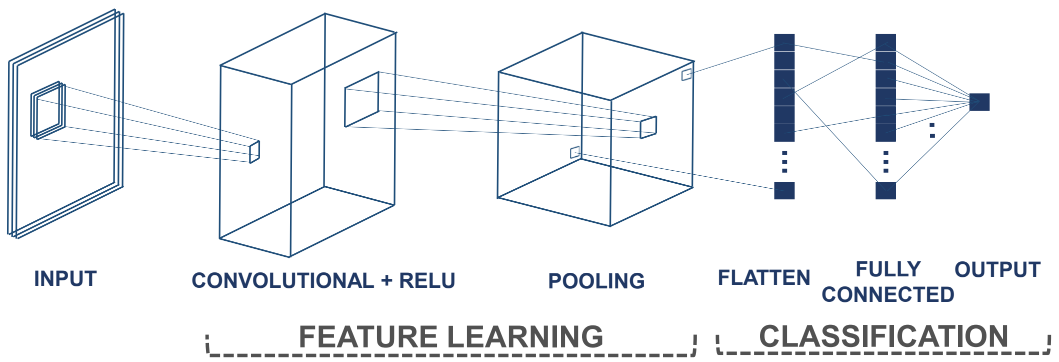


Fig. 2: A simple convolutional neural network architecture.

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 the number of parameters to control overfitting problem [26]. The pooling layer operates on 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.

In practice, max pooling often achieves a better performance compared to the other two pooling techniques [27]. The output of the pooling layer is flatten and directly passed to a fully connected layer. The output of the fully connected layer is passed to the output layer to calculate an objective function (or a loss function). The objective function is normally optimized using stochastic gradient descent (SGD) [28].

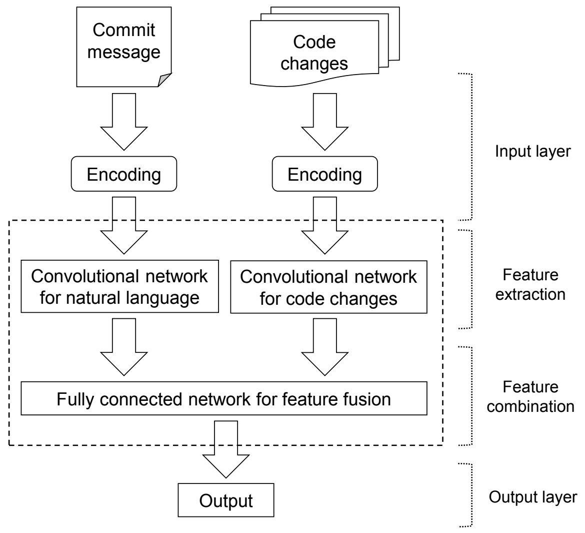
III. Approach

In this section, we first formulate the *Just-In-Time* (JIT) defect prediction problem 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.

A. Framework Overview

The goal of a JIT defect prediction model is to automatically classify a commit change as buggy or clean. This helps software teams prioritize effort and optimize testing and inspection. We consider the JIT 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 $x_i \in \mathcal{X}$ is clean ($y_i = 0$) or contains a buggy code ($y_i = 1$). The prediction function $\mathbf{f}$ can be learned by minimizing the following objective function:

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



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

Fig. 3: The general framework of the *Just-In-Time* defect prediction model.

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 JIT defect prediction model (namely DeepJIT). 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.

B. Parsing a Commit to Input Layer

To feed the raw textual data to convolutional layers for feature learning, we first encode a commit message and code changes into arrays and feed them in the input layer. For the commit message, we use NLTK [29], which is a suite of libraries for natural language processing (NLP), to extract a sequence of words from it. We employ PorterStemmer [30] to produce root forms of words. We also remove stop words

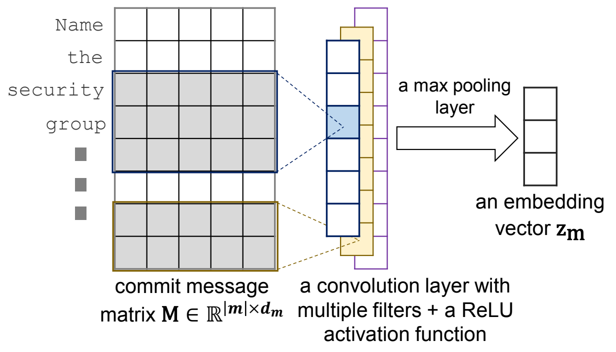

Name
the
security
group
a max pooling layer
an embedding vector $\mathbf{z_m}$
commit message matrix $\mathbf{M} \in \mathbb{R}^{|m| \times d_m}$
a convolution layer with multiple filters + a ReLU activation function

Fig. 4: A convolutional network architecture for commit message.

and rare words (e.g. those occurring less than three times in the commit messages).

We again use NLTK for parsing the code changes of a given commit. In particular, each change file in the code changes is parsed into a set of deleted and added lines, and each line is parsed into a sequence of words. We ignore comments and blank lines in the change file (see Figure 5). Following a previous work [31], we replace a number (i.e., an integer, real number, hexadecimal number) with a special `<num>` token. We also replace rare code token (e.g. those occurring less than three times in the commit codes) and tokens existing in test data but absent in the training data with a special `<unk>` token. We add a `<deleted>` token or a `<added>` token at the beginning of deleted or added line respectively so that DeepJIT recognizes whether this code line is a deleted line or an added line.

We represent each word in the commit message and code changes as a d-dimensional vector. After the preprocessing step, $\mathcal{X}_i^m$ and $\mathcal{X}_i^c$, which are the encoded data of the commit message and code changes respectively, are input to the convolutional layers to generate 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 is clean or contains a buggy code.

The core of the DeepJIT lies in the convolutional network layers for code changes (see Section III-D) and the feature combination layers (see Section III-E). In the following subsections, we firstly discuss the convolutional layers for the commit message and then present the core parts of DeepJIT in Section III-D and Section III-E.

C. *Convolutional Network Architecture for Commit Message*

CNN was first used to automatically learn the salient features in the images from raw pixel values [24]. Recently, CNN has also generated multiple breakthroughs in various Natural Language Processing (NLP) applications [16]–[20]. The architecture of CNN allowed it to extract the structural information features from raw text data of word embedding. Figure 4 presents an architecture of CNN for commit messages. The architecture includes a convolutional layer with multiple filters and a nonlinear activation function (i.e., RELU). We briefly explain it in the following paragraphs.

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 $w_i \rightarrow W_i$, $i = 1, \ldots, |m|$ in the given commit message. Each word w_i now is represented by an embedding vector, i.e., $W_i \in \mathbb{R}^{d_m}$, where d_m is a d_m-dimensional vector of a word appearing in the commit message.

Following the previous works [16], [19], 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 achieves a better performance compared to other activation functions [32]–[34]. 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 [15] over the output vector $\mathbf{c}$ to obtain the highest value as follows:

$$\underset{1 \leq i \leq |m|-k+1}{max} \quad c_i \tag{6}$$

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).

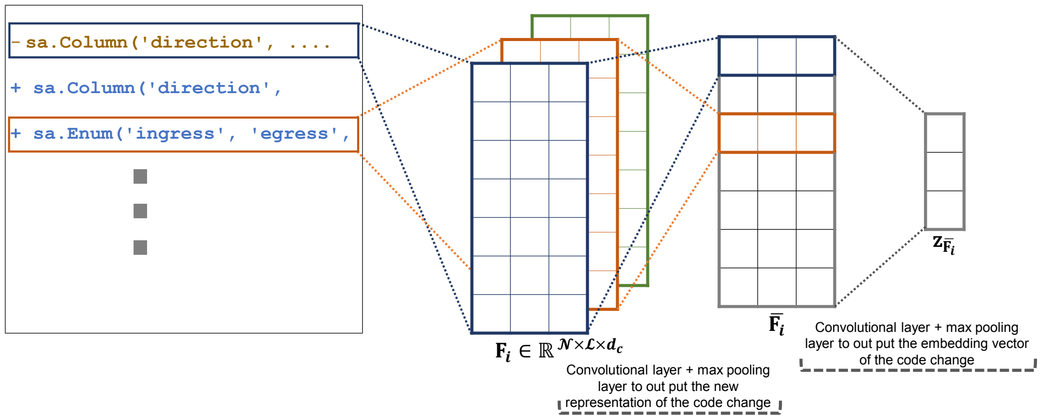

- sa.Column('direction', ...
+ sa.Column('direction',
+ sa.Enum('ingress', 'egress',
$\mathbf{F}_i \in \mathbb{R}^{\mathcal{N}\times\mathcal{L}\times d_c}$
Convolutional layer + max pooling layer to out put the new representation of the code change
$\overline{\mathbf{F}}_i$
Convolutional layer + max pooling layer to out put the embedding vector of the code change
$\mathbf{z}_{\overline{\mathbf{F}}_i}$

Fig. 5: 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.

D. 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 [35]. On the other hand, the code change includes a change in different files and different kinds of changes (removals or additions) for each 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 a deep learning framework for extracting silent features from code changes based on convolutional neural networks.

Given a code change $\mathcal{C}$ 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 III-C, we first aim to obtain its matrix representation $\mathrm{F}_i \rightarrow \mathbf{F}_i \in \mathbb{R}^{\mathcal{N}\times\mathcal{L}\times d_c}$, 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 III-C 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 deleted or added lines. Figure 5 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.

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.

E. Feature Combination

Figure 6 shows the details of architecture of the feature combination. The inputs of this architecture are the two embedding vectors $\mathbf{z_m}$ and $\mathbf{z}_C$ which represent the salient features extracted from the commit message and code change, respectively.

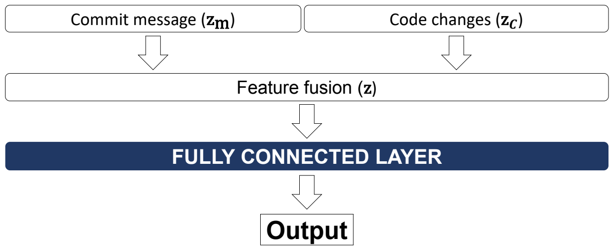

Commit message ($\mathbf{z_m}$)
Code changes ($\mathbf{z}_C$)
Feature fusion (z)
FULLY CONNECTED LAYER
Output

Fig. 6: The structure of fully-connected network for feature combination. The embedding vector of commit message $\mathbf{z_m}$ and code change $\mathbf{z}_C$ are concatenated to generate a single vector (i.e., $\mathbf{z}$).

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.

F. 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.

Just-In-Time defect prediction datasets often suffer the imbalance problem: only a few commits contain a buggy code while a large number of commits are clean. This imbalance nature increases the difficulty in learning a prediction function [36]. Specifically, the imbalance problem may affect the performance of a defect prediction model as the overall accuracy is biased to the majority class (e.g., commits containing a buggy code), leading to misclassification of the minority class (e.g., commits containing a non-buggy code). Inspired by [37], [38], we propose an unequal misclassification loss function which specifically aims to reduce the negative influence of the imbalanced data. Unlike traditional methods, this "cost-senstive" learning technique does *not* treat all misclassification equally. Instead, we impose a higher cost on misclassification of the minority class (i.e., buggy commits) than we do with misclassification of the majority class (i.e., clearn commits). Details of this technique is as follows.

Let $\mathbf{w_n}$ and $\mathbf{w_p}$ denote 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:

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

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 [39]. We also apply the dropout technique [40] to improve the robustness of our model.

We choose Adam [41], which is a variant of stochastic gradient descent (SGD) [28], 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 [41]–[43]. To efficiently compute the gradients in linear time (with respect to the neural network size), we use backpropagation [44], which is a simple implementation of the chain rule of partial derivatives.

IV. Experiments

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

A. Dataset

We used two well-known software projects (i.e., QT and OPENSTACK) to evaluate the performance of *Just-In-Time* (JIT) models. QT [4], developed by the Qt Company, is a cross-platform application framework and allows contributions from individual developers and organizations. On the other hand, OPENSTACK [5] is an open-source software platform for cloud computing and is deployed as an infrastructure-as-a-service which allows customers to access its resources.

TABLE I: Summary of the dataset used in this work

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

Table I briefly summarizes the dataset used in our paper. This dataset was originally collected and cleaned by McIntosh and Kamei [21]. After their cleaning process, the QT dataset contains 25,150 commits, while the OPENSTACK dataset contains 12,374 commits. McIntosh and Kamei stratified the

[4]https://www.qt.io/
[5]https://www.openstack.org/

dataset into six months periods for time-sensitive training-and-testing settings.

B. Baselines

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

- JIT: This method for identifying buggy code changes was proposed by McIntosh and Kamei [21]. The method used a nonlinear variant of multiple regression modeling [56] to build a classification model for automatically identifying defects in commits. McIntosh and Kamei manually designed a set of code features, using six families of code change properties, which were primarily derived from prior studies [3], [6]–[8]. These properties were: the magnitude of changes, 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. Table II briefly summarizes the code features extracted from code change properties.
- DBNJIT: This approach adopted Deep Belief Network (DBN) [57] to generate a more expressive set of features from an initial feature set [14]. The generated feature set, which is a nonlinear combination of the initial features, was put into a machine learning classifier [58] to predict buggy commits. For a fair comparison, we used McIntosh and Kamei [21]'s features as the initial feature set for DBNJIT.

For all the above-mentioned techniques, we employ the same parameters and settings as described in the respective papers.

C. 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, we avoid using threshold-dependent measures (i.e., precision, recall, or F1) since these measures strongly depend on arbitraily thresholds [59], [60]. Following the previous work by McIntosh and Kamei [21], we use the Area Under the receiver operator characteristics Curve (AUC) to measure the discriminatory power of DeepJIT, i.e., their ability to differentiate between defective 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 AUC range between 0 (worst discrimination) and 1 (perfect discrimination).

D. Training and hyperparameters

One of the key challenges in training DeepJIT is how to select the dimension of the word vectors for the commit message (d_m) and code changes (d_c), and the size of the convolution layers (i.e., see Section III-C and Section III-D). We evaluated the performance of DeepJIT, using 5-fold cross validation, across different word dimensions and number of filters. Figure 7 and Figure 8 present the AUC results of DeepJIT for these hyperparameters. The figures show that

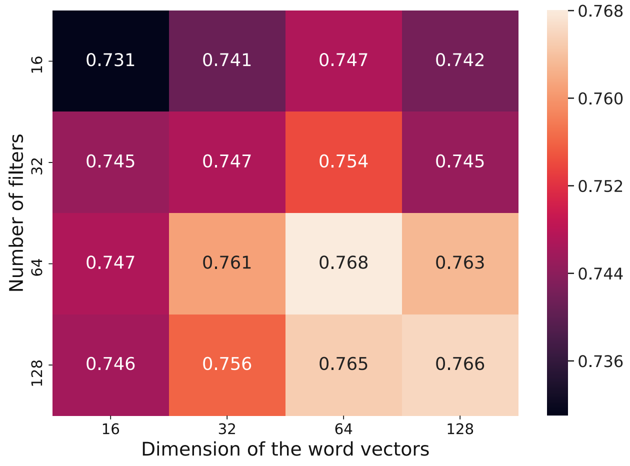


Fig. 7: The AUC results of DeepJIT across two different hyperparameters in QT project.

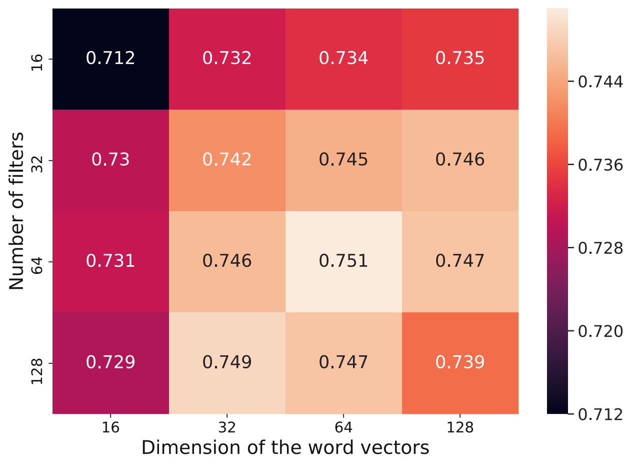


Fig. 8: The AUC results of DeepJIT across two different hyperparameters in OPENSTACK project.

DeepJIT achieves the best AUC results when the dimension of word vectors and the number of filters are set to 64. We set the other hyperparameters as follows: The batch size was set to 32. The size of DeepJIT's fully-connected layer described in Section III-E was set to 512. These hyperparameter settings are commonly used in prior deep learning work [61]–[64].

We trained DeepJIT using Adam method [41] with shuffled mini-batches. We also trained DeepJIT for 100 epochs. We applied an early stopping strategy [39], [65] to avoid overfitting problem during the training process. We stopped the training if the value of the objective function (see Equation 12) has not been updated in the last 5 epochs.

E. Research Questions and Results

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

TABLE II: A summary of McIntosh and Kamei's code features [21].

	Property	Description	Rationale
Size	Lines deleted	The number of deleted lines.	The more deleted or added code, the more likely that defects may appear [45], [46].
	Lines added	The number of added lines.	
Diffusion	Subsystems	The number of modified subsystems.	Scattered changes may have more defects compared to focused one [47], [48].
	Directories	The number of modified directories.	
	Files	The number of modified files.	
	Entropy	The spread of modified lines across file.	
History	Unique changes	The number of prior changes to the modified files.	More changes may lead to have defects since developers need to track many previous changes [6].
	Developers	The number of developers who have changed the modified files in the past.	Files touched by many developers may include defects [49].
	Age	The time interval between the last and current changes.	More recently changed code likely contains defects compared to older code [50].
Author/Rev. Experience	Prior changes	The number of prior changes that an actor has participated in.	Changes produced by novices are likely to be more defective than changes produced by experienced developers [3].
	Recent changes	The number of prior changes that an actor has participated in weighted by the age of the changes (older changes are given less weight than recent ones).	
	Subsystem changes	The number of prior changes to the modified subsystem(s) that an actor has participated in.	
	Awareness	The proportion of the prior changes to the modified subsystem(s) that an actor has participated in.	Changes made by developers who are aware of the prior changes in the impacted subsystems are likely to be less risky.
Review	Iterations	Number of times that a change was revised prior to integration.	The quality of a change likely improves with each iteration. Hence, changes that undergo iterations prior to integration may be less risky [51], [52].
	Reviewers	Number of reviewers who have voted on whether a change should be integrated or abandoned.	Changes observed by many reviewers are likely to be less risky [53].
	Comments	The number of non-automated, non-owner comments posted during the review of a change.	Changes with short discussions may be more risky [54], [55].
	Review window	The length of time between the creation of a review request and its final approval for integration.	Changes with shorter review windows may be more risky [51], [52].

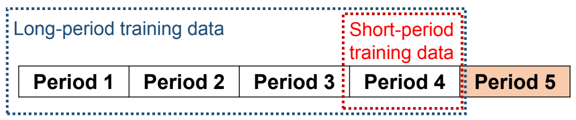


Fig. 9: An example of choosing the training data for short-period and long-period models. The last period is used as testing data.

TABLE III: The AUC results of DeepJIT vs. with other baselines in three types of JIT models: cross-validation, short-period, and long-period.

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

To address RQ1, we evaluated the accuracy of a trained JIT model in predicting buggy changes using test data. In particular, we considered three evaluation settings:

- **Cross-validation:** To evaluate machine learning algorithm, most people use k-fold cross-validation [66] in which a dataset is randomly divided to k folds, each fold is considered as testing data for evaluating JIT model while $k-1$ folds are considered as training 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:** 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:** Inspired by the work [67], 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 9 describes how the training data is selected to train models following the short-period and long-period settings. We used the last period (i.e., period 5) as a testing data. While the short-period model was trained using the commits that occurred during period 4, the long-period model was trained using the commits that occurred from period 1 to 4. After training short-period and long-period models, we measured their performance using AUC evaluation metric described in Section IV-C.

Table III shows the AUC results of DeepJIT as well as other baselines considering the three evaluation settings: cross-validation, short-period, and long-period. The difference between results obtained using cross-validation, short-period, and long-period settings is relatively small (i.e., below 2.2%) which suggests that there is no difference between training on past or future data. In the QT project, DeepJIT achieved AUC scores of 0.768, 0.764, and 0.765 in three different evaluation settings: cross-validation, short-period, and long-period, respectively. Comparing them to the best performing baseline (i.e., DBNJIT), DeepJIT achieved improvements of 8.96%, 7.00%, and 8.05% in terms of AUC. In the OPENSTACK project, DeepJIT also constituted improvements of 8.21%, 9.08%, and 8.29% in terms of AUC compared to DBNJIT (the best performing baseline). We also employed the Scott-Knott test [68] on the cross-validation evaluation setting to statistically compare the differences between the three considered JIT models. The results show that DeepJIT consistently appears in the top Scott-Knott ESD rank in terms of AUC (i.e, DeepJIT > DBNJIT > JIT).

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

TABLE IV: Contribution of feature components in DeepJIT.

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

To answer this question, we employed an ablation test [69], [70], by ignoring the commit message and the code change in a commit and then evaluate the AUC performance. Specifically, we created two different variants of DeepJIT, namely DeepJIT-Msg and DeepJIT-Code. DeepJIT-Msg only considers commit message information while DeepJIT-Code only uses commit code information. We again used the three evaluation settings (i.e., cross-validation, short-period, and long-period) and the AUC scores to evaluate the performance of our models. Table IV shows the performance of DeepJIT degrades if we ignore any one of the considered types of information (i.e. commit messages or code changes). The AUC scores dropped by 19.81%, 28.45%, and 19.01% in the project QT and dropped by 9.00%, 33.96%, and 16.00% in the project OPENSTACK for the three evaluation settings if we ignore commit messages. The AUC scores dropped by 4.07%, 4.09%, and 5.23% in the project QT and dropped by 3.02%, 1.56%, and 4.47% in the project OPENSTACK for the three evaluation settings if we ignore code changes information. It suggests that each information type contributes to DeepJIT's performance. Moreover, it also indicates that code changes are more important to detect buggy commits than commit messages.

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

TABLE V: Combination of DeepJIT with the manually crafted code features from [21].

Settings	**Models**	QT	OPENSTACK
Cross-validation	DeepJIT	0.768	0.751
	DeepJIT-Combined	**0.779**	**0.76**
Short-Period	DeepJIT	0.764	0.781
	DeepJIT-Combined	**0.788**	**0.814**
Long-period	DeepJIT	0.765	0.771
	DeepJIT-Combined	**0.786**	**0.799**

TABLE VI: Training time of DeepJIT

Dataset	Cross-validation	Short-period	Long-period
QT	5 hours 43 mins	17.2 mins	1 hours 18 mins
OPENSTACK	12 hours 15 mins	10.1 mins	2 hours 37 mins

To address this question, we incorporated the code features, derived from [21], 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 III-E), to build a new single vector $\mathbf{z}$ as follows:

$$\mathbf{z} = \mathbf{z_m} \oplus \mathbf{z}_C \oplus \mathbf{z_r} \tag{13}$$

where $\oplus$ is the concatenation operator. Table V shows the AUC results of a DeepJIT variant (referred to as DeepJIT-Combined) that also leverages McIntosh and Kamei [21]'s manually crafted features. We find that the AUC scores increased by 1.43%, 3.14%, and 2.75% in the project QT and they increased by 1.20%, 4.23%, and 3.63% in the project OPENSTACK for the three evaluation settings (i.e. cross-validation, short-period, long-period). DeepJIT-Combined improved the best baseline model (i.e. DBNJIT) by 10.50%, 10.36%, and 11.02% in the project QT and 9.51%, 13.69%, 12.22% in the project OPENSTACK for the there evaluation settings. This suggests that the manually extracted code features are complementary and can be used to improve the performance of our proposed approach.

RQ4: What are the time costs of DeepJIT?

We trained and tested DeepJIT on a NVIDIA DGX1 server with Tesla P100 [71]. Table VI shows the time costs of training DeepJIT for the three evaluation settings (i.e., cross-validation, short-period, and long-period) on QT and OPENSTACK. Cross-validation setting requires longest training time since we performed 5-fold cross-validation to evaluate the performance of DeepJIT. Long-period setting requires more training time than short-period setting since it considers all commits occurring before a particular period. Once DeepJIT has been trained, it only takes a few milliseconds to generate the prediction score for a given commit.

V. Threats to Validity

We mitigated concerns related to construct validity by evaluating our approach on a publicly available dataset that has been used in previous work. This dataset contains commits extracted from real projects (QT and OPENSTACK) and buggy/no-buggy labels on those commits. Threats to conclusion validity was also minimized by using Area Under the Curve (AUC), a standard performance measure recommended for assessing the predictive performance of defect prediction models [72].

We have compared our approach against two baselines which have been proposed and implemented in existing work. Since the source code of their original implementation were not made publicly available, we needed to re-implement our own version of those techniques. Our implementation closely followed the description of their work, it might not have all of the details of the original implementation, specifically those not explicitly presented in their papers. Our study considers two large open source projects which are significantly different in size, complexity and revision history. However, due to small sample sizes, our findings may not generalize to all software projects. Further studies are needed to confirm our results for other types of software projects.

VI. Related Work

A. JIT Defect Prediction

Some previous studies focus on change-level defect prediction (i.e. JIT defect prediction). For example, Mockus and Weiss [3] predict 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. [6] 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.* [73] and Kim *et al.* [74] used source code change logs to predict commits as being buggy or not. For example, Kim *et al.* [74] used the identifiers in added and deleted source code and the words in change logs. The experimental results on the dataset collected from 12 open source software projects show that the proposed approach achieved 78 percent accuracy and a 60 percent recall.

Kononenko et al. [8] find that the addition of code change features that were extracted from code review databases contributed a significant amount of explanatory power to JIT models. McIntosh and Kamei also used 5 families of code and review features in the context of JIT defect prediction. Through a case study of 37,524 changes from QT and OpenStack systems, the paper shows that the importance of impactful families of code change features like Size and Review are consistently under or overestimated in the studied systems.

Comparing with these previous studies, we introduce the JIT defect prediction model (DeepJIT) that learns a deep representation of commits. We also evaluate the prediction performance of DeepJIT comparing with other JIT models on the dataset including code change properties. We extended the dataset that McIntosh and Kamei used to analyze [6] by adding commit messages and code changes.

B. Deep Learning Models in Defect Prediction

Deep learning has recently attracted increasing interests in software defect prediction. Deep Belief Network (DBN) [57] has been commonly used in previous work. For example, a recent work [14] used the Deep Belief Network to build JIT defect prediction models. Their approach still however rely on the same set of metric-based features that are manually engineered as in earlier work. Other studies (e.g., [9], [14], [75]) also used Deep Belief Network to automatically learn features for defect prediction. Unlike our approach, their models are not end-to-end trainable, i.e., features are learned separately (not using the defect ground-truths) and are then input to a separate traditional classifier. This approach has also been used in previous work (e.g. [76], [77]) where two other well-known deep learning architectures (Long Short Term Memory in [77] and Convolutional Neural Network in [76]) was leveraged to automatic feature learning for defect prediction. There is a risk in those approaches that the learned features may not correlate with defect outcomes. End-to-end models like our approach address this issue by enforcing the models to learn and generate features that best correlate with the target label.

VII. Conclusion and Future Work

In this paper, we propose an end-to-end deep learning model (namely DeepJIT) for *Just-In-Time* defect prediction problem. For a given commit, DeepJIT automatically extracts features from the commit message and the set of code changes. These features are then combined to evaluate how likely the commit is buggy. DeepJIT also allows users to add their manually crafted features to make it more robust. We evaluate DeepJIT on two popular software projects (i.e. QT and OPENSTACK) on three evaluation settings (i.e. cross-validation, short-period, and long-period). The evaluation results show that compared to the best performing state-of-the-art baseline (DBNJIT), the best variant of DeepJIT (DeepJIT-Combined) achieves improvements of 10.50%, 10.36%, and 11.02% in the project QT and 9.51%, 13.69%, 12.22% in the project OPENSTACK in terms of the Area Under the Curve (AUC).

Our future work involves extending our evaluation to other open source and commercial projects. We also plan to extend DeepJIT using attention neural network [78] so that our model can explain its predictions to software practitioners. We also plan to implement DeepJIT into a tool (e.g. a GitHub plugin) to assess its usefulness in practice.

Dataset and Code. The dataset and code for DeepJIT are available at https://github.com/AnonymousAccountConf/DeepJTT_MSR.

Acknowledgements. This research was partially supported by Singapore National Research Foundation (award number: NRF2016-NRF-ANR003) and JSPS KAKENHI Grant Numbers JP15H05306 and JP18H0322.

[6]https://github.com/software-rebels/JITMovingTarget

References

[1] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," in *Leaders of Tomorrow Symposium: Future of Software Engineering, FOSE@SANER 2016, Osaka, Japan, March 14, 2016*, 2016, pp. 33–45.

[2] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Engg.*, vol. 17, no. 4-5, pp. 531–577, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9173-9

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

[4] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 62:1–62:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393670

[5] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "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*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 812–823. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818852

[6] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013. [Online]. Available: http://dx.doi.org/10.1109/TSE.2012.70

[7] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar. 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70773

[8] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 111–120. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2015.7332457

[9] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 297–308. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884804

[10] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 269–280. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635875

[11] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 858–868. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818858

[12] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337322

[13] Z. Li and Y. Zhou, "Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 306–315. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081755

[14] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, ser. QRS '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 17–26. [Online]. Available: http://dx.doi.org/10.1109/QRS.2015.14

[15] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[16] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[17] C. dos Santos and M. Gatti, "Deep convolutional neural networks for sentiment analysis of short texts," in *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, 2014, pp. 69–78.

[18] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," *arXiv preprint arXiv:1404.2188*, 2014.

[19] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Advances in neural information processing systems*, 2015, pp. 649–657.

[20] R. Johnson and T. Zhang, "Effective use of word order for text categorization with convolutional neural networks," *arXiv preprint arXiv:1412.1058*, 2014.

[21] S. McIntosh and Y. Kamei, "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*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 560–560. [Online]. Available: http://doi.acm.org/10.1145/3180155.3182514

[22] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[23] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE transactions on neural networks*, vol. 8, no. 1, pp. 98–113, 1997.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[25] H. Zhao, O. Gallo, I. Frosio, and J. Kautz, "Loss functions for image restoration with neural networks," *IEEE Transactions on Computational Imaging*, vol. 3, no. 1, pp. 47–57, 2017.

[26] G. Tolias, R. Sicre, and H. Jégou, "Particular object retrieval with integral max-pooling of cnn activations," *arXiv preprint arXiv:1511.05879*, 2015.

[27] M. D. Zeiler and R. Fergus, "Stochastic pooling for regularization of deep convolutional neural networks," *arXiv preprint arXiv:1301.3557*, 2013.

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

[29] S. Bird and E. Loper, "Nltk: the natural language toolkit," in *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*. Association for Computational Linguistics, 2004, p. 31.

[30] P. Willett, "The porter stemming algorithm: then and now," *Program*, 2006.

[31] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 334–345.

[32] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[33] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "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, 2013, pp. 8609–8613.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[35] H. T. Ng and J. Zelle, "Corpus-based approaches to semantic interpretation in nlp," *AI magazine*, vol. 18, no. 4, p. 45, 1997.

[36] N. V. Chawla, N. Japkowicz, and A. Kotcz, "Special issue on learning from imbalanced data sets," *ACM Sigkdd Explorations Newsletter*, vol. 6, no. 1, pp. 1–6, 2004.

[37] Z.-H. Zhou and X.-Y. Liu, "Training cost-sensitive neural networks with methods addressing the class imbalance problem," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 1, pp. 63–77, 2006.

[38] M. Kukar, I. Kononenko *et al.*, "Cost-sensitive learning with neural networks." in *ECAI*, 1998, pp. 445–449.

[39] R. Caruana, S. Lawrence, and C. L. Giles, "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping," in *Advances in neural information processing systems*, 2001, pp. 402–408.

[40] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from over-

fitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[41] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of 3rd International Conference on Learning Representations (ICLR)*, 2015.

[42] M. Anthimopoulos, S. Christodoulidis, L. Ebner, A. Christe, and S. Mougiakakou, "Lung pattern classification for interstitial lung diseases using a deep convolutional neural network," *IEEE transactions on medical imaging*, vol. 35, no. 5, pp. 1207–1216, 2016.

[43] S. Arora, N. Cohen, and E. Hazan, "On the optimization of deep networks: Implicit acceleration by overparameterization," in *35th International Conference on Machine Learning (ICML)*, 2018, pp. 244–253.

[44] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989–993, 1994.

[45] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 452–461.

[46] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort aware models," in *Proc. Int'l Conf. on Software Maintenance (ICSM'10)*, 2010, pp. 1–10.

[47] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 31–41.

[48] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 78–88.

[49] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 2010, p. 18.

[50] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, 2000.

[51] A. Porter, H. Siy, A. Mockus, and L. Votta, "Understanding the sources of variation in software inspections," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 1, pp. 41–79, 1998.

[52] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the qt system," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 168–179.

[53] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Oreilly & Associates Inc, 2001.

[54] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 192–201.

[55] S. Mcintosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Softw. Engg.*, vol. 21, no. 5, pp. 2146–2189, Oct. 2016. [Online]. Available: http://dx.doi.org/10.1007/s10664-015-9381-9

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

[57] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[58] N. M. Nasrabadi, "Pattern recognition and machine learning," *Journal of electronic imaging*, vol. 16, no. 4, p. 049901, 2007.

[59] G. H. Nguyen, A. Bouzerdoum, and S. L. Phung, "Learning pattern classification tasks with imbalanced data sets," in *Pattern recognition*. InTech, 2009.

[60] Q. Gu, Z. Cai, L. Zhu, and B. Huang, "Data mining on imbalanced data sets," in *Advanced Computer Theory and Engineering, 2008. ICACTE'08. International Conference on*. IEEE, 2008, pp. 1020–1024.

[61] A. Severyn and A. Moschitti, "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, 2015, pp. 373–382.

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

[63] X. Huo and M. Li, "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, 2017, pp. 1909–1915.

[64] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[65] L. Prechelt, "Automatic early stopping using cross validation: quantifying the criteria," *Neural Networks*, vol. 11, no. 4, pp. 761–767, 1998.

[66] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, vol. 14, no. 2. Montreal, Canada, 1995, pp. 1137–1145.

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

[68] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 789–800.

[69] B. Korbar, A. M. Olofson, A. P. Miraflor, C. M. Nicka, M. A. Suriawinata, L. Torresani, A. A. Suriawinata, and S. Hassanpour, "Deep learning for classification of colorectal polyps on whole-slide images," *Journal of pathology informatics*, vol. 8, 2017.

[70] J. Liu, W.-C. Chang, Y. Wu, and Y. Yang, "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, 2017, pp. 115–124.

[71] N. A. Gawande, J. A. Daily, C. Siegel, N. R. Tallent, and A. Vishnu, "Scaling deep learning workloads: Nvidia dgx-1/pascal and intel knights landing," *Future Generation Computer Systems*, 2018.

[72] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," 2018.

[73] L. Aversano, L. Cerulo, and C. Del Grosso, "Learning from bug-introducing changes to prevent fault prone code," in *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE'07)*, 2007, pp. 19–26.

[74] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, 2008.

[75] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

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

[77] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Transactions on Software Engineering*, 2019. [Online]. Available: DOI:10.1109/TSE.2018.2881961.

[78] W. Yin, H. Schütze, B. Xiang, and B. Zhou, "Abcnn: Attention-based convolutional neural network for modeling sentence pairs," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 259–272, 2016.