Master's Thesis

Deep Learning based Semantics Model for Software Defect Prediction

Jidong Li

17M38124

Graduate Major in Computer Science
School of Computing
Tokyo Institute of Technology

Supervisor: Takashi Kobayashi

January, 2020

Abstract

Software defect prediction (SDP) is an important technique that can help developers to focus on modules which are more likely to occur failure. However, traditional methods mainly focus on statistical metrics and ignore the semantics and syntax of source code. Although existing deep learning-based can generate semantics features, they can not learn the semantics of identical token under different contexts. Besides, Due to the limited amount of data, the deep learning-based models may not be able to perform their corresponding prediction capabilities.

In this paper, to deeply extract the semantics of source code, we propose a technique to extract semantic feature by using BERT model for SDP. We pretrain BERT model with a large code corpus collected from BigCode. Then we convert token sequences into embeded token vectors with our BERT model and utilize Bidirectional Long-Short Term Memory Network (BiLSTM) to learn contexts information among tokens. After that, we adopt max pooling and average pooling to generate features. Both methods can reduce the complexity of our model to relieve the overfitting problem. Besides attention mechanism is adopted to extract information of important nodes. To evaluate our approach, we divide the experiment into two types; one is Within-Project Defect Prediction (WPDP), another is Cross-Project Defect Prediction (CPDP). We compare with two other deep learning-based models, including Convolutional Neural Networks (CNN) and BiLSTM. To train the model, we adopt the logistics regression classifier. Besides, we also compare our model with two statistical-based models — all the comparisons conducted with both WPDP and CPDP type experiments. To relieve the overfitting problem, we adopted a data augmentation strategy to generate more training dataset. Apart from that, we also compare our pretrained BERT model and Word2vec model. Finally, we evaluate two data processing methods, full-token and AST-node. We compare two processing method by conducting the length of coverage on each project from 50% to 90% in both WPDP and CPDP experiment

The results show that the average F1 score on our ten projects of our proposed BERT-based fine-tuning model is 8.8%, 2.3%, 9%, and 7% higher than BiLSTM, CNN and two other traditional models, respectively in WPDP. For CPDP, the results are 12.6%, 4.7%, 10.2% and 7% in F1 score. The data augmentation strategy can improve 0.54 in the F1 score. However, our BERT embedding does not outperform *Word2vec*. Finally, we found that full-token data type can achieve better performance than AST-node as it can present more semantics of sequences.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software defect prediction(SDP) is a kind of technique for software checking and debugging. It will cost much for software maintenance at a later period. Since, By using SDP, the developers can focus on the modules that most likely have bug.

The recent technique mainly based on the machine learning model to predict whether a file or have bug or not. These methods mainly make use of historical data, then construct a series of metrics to train a machine learning model. So that developers can utilize the SDP model to predict potential new defective modules.

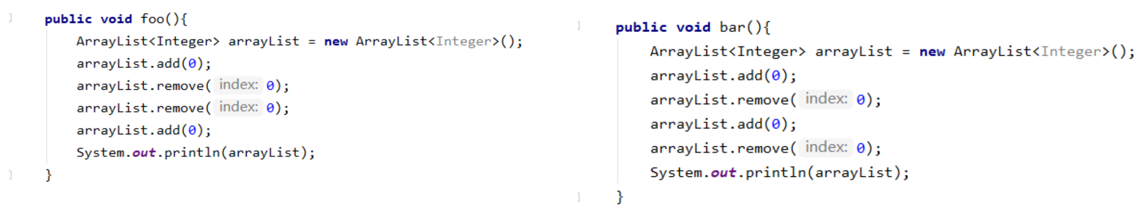
Building effective metrics is essential in the machine learning-based SDP model. Many researchers have proposed their concepts and idea to construct metrics of SDP model. In the early researches, Line Of Code (LOC) [1] was employed to measure if a file that has bug or not. However, such simple metrics ignore the structure of code. Halstead proposed a software complexity metric by counting the occurrence of operator and operand in source codes [10]. McCabe also raised the concept of cyclomatic complexity that the possibility of causing bugs increase as the program becomes complex [20]. CK metrics suite measures the software complexity by considering its inheritance, coupling, and cohesion [4]. MOOD [11, 23] metrics are also be proved as effective in SDP.

As for the machine learning model, many classical machine learning models are adopted for SDP. such as logistics regression (LR), support vector machine (SVM), decision tree (DT), random forest (RF) Adaboost, and Naive Bayes (NB) [7, 9, 29]. Moreover, ensemble learning strategies such as voting, stacking, and averaging perform well in SDP. [3, 15, 17, 26, 30].

However, traditional machine learning-based models may have some problems:

1. Need to construct metrics manually.
2. Do not consider the semantics of source codes.

Traditional machine learning model need to extract features manually to construct a machine learning model while the quality of metrics is directly impact the performance of machine learning model. In addition, traditional metrics do not consider the semantics of source codes, which is important feature for SDP. Take Figure 1.1 as example, function `foo()` and `bar()` has the same of count of line of code. However, when we select `add()` and `remove()` as the information node from these two code snippets, we will generate [`add(0)`,`remove(0)`,`remove(0)`,`add(0)`] and [`add(0)`,`remove(0)`,`add(0)`, `remove(0)`] for `foo()` and `bar()`,respectively. However, `foo()` will occur exception. Such kind of feature that traditional metrics can not recognize. We also call it **semantics feature**. To



```
public void foo(){
    ArrayList<Integer> arrayList = new ArrayList<Integer>();
    arrayList.add(0);
    arrayList.remove( index: 0);
    arrayList.remove( index: 0);
    arrayList.add(0);
    System.out.println(arrayList);
}
```

(a)

```
public void bar(){
    ArrayList<Integer> arrayList = new ArrayList<Integer>();
    arrayList.add(0);
    arrayList.remove( index: 0);
    arrayList.add(0);
    arrayList.remove( index: 0);
    System.out.println(arrayList);
}
```

(b)

Figure 1.1: a motivation example

extract semantics features. Wong et.al [28] extract abstract syntax tree (AST) and select specific nodes from AST to present the code semantics, then they apply Deep Belief Network (DBN) [12] to extract feature from the embeded AST nodes and achieved better performance than traditional models. Li et.al [18] also adopted Convolutional Neural Networks (CNN) to extract the features of source codes, which also perform better than traditional machine learning model. Tree-base LSTM [5] was employed to solve different level semantics of source codes.

Undoubtedly, such existing deep learning models can effectively improve the performance of SDP. However, they still have some problems that unsolved.

- Limited by training data, the token may not be trained sufficiently.
- Not be able to learn identical symbols semantics in different context

Limited by existing public defect datasets, the number of instances in the largest project is no more than 2000. Tokens may not be trained sufficiently in the token embedding phase. To ensure that token can be well-trained, we pretrain large Java source codes using *BERT* semantics of identical token under different semantics. Different from *Word2vec* [21, 22] model, the *BERT* [6] model can learn the deep semantics by short-cut structure to relieve overfitting problem when deep learning model constructed with large amount of parameter. To solve the shortage of datasets, we proposed a data augmentation strategy to generate more data for training.

On the one hand, tokens in training data not appeared in test data, which will impact the performance of the model. On the other hand, existing researches mainly select specific nodes from ASTs tree and generate the sequence. Though such kind of approaches can learn the semantics of source code, it filters out large nodes a file. Such methods will lose much semantics information. The result is that vocabulary size becomes smaller compared with the approach that uses all tokens from source code.

Since attention-based LSTM [13] and CNN show excellent performance in text classification and previous, we make use of both abilities to learn semantics to generate features. BiLSTM can learn context information in a sequence both forward and backward direction while the attention mechanism can focus the token with important information. Such a form of combination is widely applied in NLP tasks. [32, 33] CNN can learn the local information of sequence. Both two approaches are widely applied in academic research and industry.
In this study, we summarize our contribution and show in follows:

- We proposed a data augmentation method for SDP.
- We proposed a *BERT* fine-tuning method for SDP and show that *BERT* pretrained model is applicable in SDP.

- We also compare with two mainstream source code processing methods in SDP.

Base on the problem given above, we raise several research questions to validate the effectiveness of our model.

RQ1: Can data augmentation improve the performance of deep learning models?. To verify our assumption that more data for training can improve the performance, we boost training with different degrees and compare their performance on our proposed model.

RQ2: Can our model outperform other models, including traditional metrics based models and deep learning-based models?. To verify whether our model outperforms existing deep learning models, we select two deep learning models, the CNN model, and attention-based *BiLSTM*, as baselines to compare with our model. For traditional features based models, we construct two benchmarks with logistics regression as the conventional model.

RQ3: Can inputs embedding with our pretrained *BERT* model outperform *Word2vec* model?. We compare *BERT*-based inputs embedding method with *Word2vec*-based inputs embedding method.

RQ4: Which data preprocessing method is better for software defect prediction? Since mainly researches parse source code into AST while another method uses all tokens. We compare two preprocessing methods to investigate which form is more effective for SDP.

The rest part of this paper follows. Chapter 2 introduce the relevant researches and background; Chapter 3 introduces the details of the proposed model; Chapter 4 illustrates how we design the experiments for research questions; Chapter 5 is the results of our experiments; chapter 6 is the discussion of our results; Chapter 7 is the conclusion of our experiments.

Chapter 2

Background & Related Works

2.1 Software Defect Prediction

In this section, we introduce related works of software defect prediction. Since machine learning-based approach many contain two phases, one is defect metric, another is machine learning model. Figure 2.1 shows the general framework of machine learning base model.

2.1.1 Module Extraction

To construct an SDP model, the first step we need to extract modules from the project, the level can be module, file or function level. Using the defect tracking system to label the defective for each module.

2.1.2 Feature Generation

The next step is feature generation; after we obtain modules, we need to generate features for each module, their two primary methods to generate feature, one is to apply designed metric to create the feature, another is using deep learning automatically generate features.

2.1.3 Model Construction

After we obtain features and labels of project, i.e dataset, we need to preprocess the dataset so that it can fit the model we selected. Generally, the main problems in the data preprocessing process are data imbalance, data loss and data normalization. Then adopt a suitable classification algorithm to train a model. Finally, apply the trained model to predict defective of new instances.

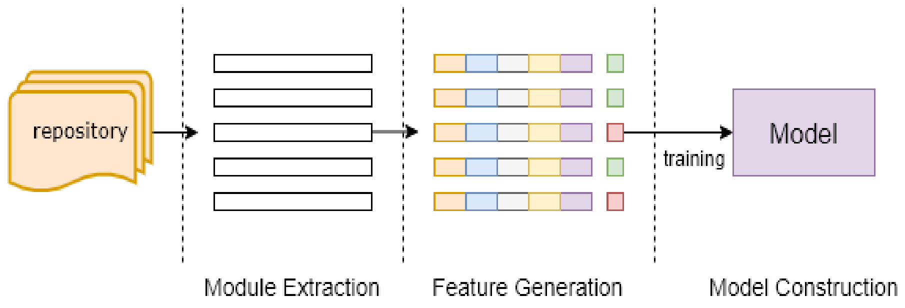


Figure 2.1: general framework of machine learning-based model

2.1.4 Software Defect Metrics

There are many forms of metric construction approach, mainly can be divided into three forms of pattern: code-based approach and process-based approach.

Code-based Approach

In the early age, Akiyama [1] employed *line of code* as metric for SDP. The more codes in a file or module, The more likely a file or module have bug. However, LOC is simple and not enough to indicate the complexity of the program. Thereby, in later research, Halstead and MaCabe metric consider the cyclomatic complexity of programs. Halstead metric [10] believe that the count of operand and operator impact program reading speed for the reader. The more difficult to read a program, the more possible it contains bug. McCabe [20] metric mainly focus on the control flow of program if a control flow has higher in this approach. The complexity of the control flow in a program decides whether it has bug or not. However, and the formula is $v(G) = e - n + 2$, where e is amount of edge, n is amount of node. In addition, it can calculate the essential complexity and design complexity. With the prevalence of OOP, Many researchers begin to design metric for OOP. One of the most important metric is CK metric proposed by Chidamber and Kemerer [4]. CK metric considers the importance of the most coupling, inheritance and cohesion of programs.

Process-based Approach

Software defect is not only associate with code itself but also associate with external factor. developers' experience and churn of code may result from defect problem in software development.

2.2 Deep learning-based Software defect Prediction

With the popularity of NLP development, many researchers begin to apply NLP technique in SDP domain. The advantages of the deep learning approach can learn the semantics and syntax of the program, which is ignored by the traditional model. Such kind of approach shows excellent performance compare with the traditional machine learning approach.

Wang [28] extract specific node from AST which was extracted from source code. Then convert these nodes into sequences as the input of DBN to generate vector to present the semantics of the program. After that, they built a classification model to by using both defect label and the generated features. The performance of this approach is 14.2% and 8.9% higher in F-measurement in terms of WPDP and CPDP, respectively.

Encouraged by Wang's approach, Li [18] utilize Convolutional neural network to construct the SDP model. They learn from wang's approach, generated semantic features by extracting AST nodes from source code. Then the features will be an input of the convolutional neural network. The result shows that this approach improved F1 score compared with Wang's.

However, such approaches did not consider the level of the semantics of programs, Dam's el.al [5] proposed tree-based LSTM to learn different level semantics of programs. They convert the different level of nodes of AST into vectors and define a loss function to train node vector to generate vectors of AST. Using the AST vector to train the SDP model.

This approach achieved a higher score than the existing method in recall.

However, when constructing a deep learning model, embedding is necessary for the downstream task. The general process is padding the sequence into fixed length. However, if a long sequence with more information is condensed into a fixed length, it will lose part of the information. Thereby, Fan et.al [8] propose *BiLSTM+attention* method to solve the problem. They first apply *BiLSTM* to extract the semantics feature, then use *attention* to calculate the weight of hidden state generated by *BiLSTM*. The model outperforms state-of-the-art model 14% and 7% in F-measure and AUC score, respectively.

The models that mentioned above have similar problems, i.e without pretrained token vector. Since the defect datasets are small, tokens may not be well-trained and will impact the performance of the downstream layer. Thereby, Liang el.al [19] pretrained large tokens vector to alleviate this problem. And their proposed model outperforms other models both on within-project defect prediction and cross-project defect prediction.

2.3 Word Emebedding

In NLP domain, word embedding is based process for downstream task. Since present token through one hot vector may spend large storage, thereby, there are many types of research are manage to condense one hot vector into dense vectors and present its semantics simultaneously. In this section, we introduce some typical and classic method to embed word.

Word2Vec [21, 22], This is typical method that transfer token into vector to present its semantics. In *Word2Vec* model, it is kind of supervisor learning and has two forms of models. one is **skip-gram** and another is **Continuous-Bag-Of-Word** (CBOW). Skip-gram model is to predict the context tokens when given the centre word. and **CBOW** is the model to predict centre token given context. *Word2Vec* is widely applying in NLP task. The aim of *Word2Vec* is to train a word vector that can suit multi-task of NLP. However, *Word2Vec* can not handle the problems that tokens in different context present different semantics.

EMLo [24]. To handle this the problem existed in *Word2Vec*. *EMLo* apply **BiLSTM** to train a language model. Since it consider double direction of context, it can learn the semantics for identical tokens in different context.

GPT [25]. *GPT* construct from *Transformer* [27]. Since *Transformer* can learn the longer distance since that can learn distance context information. Compare with *ELMo*, *GPT* leverage the ability of *Transformer*, can learn longer distance semantics features and it outperform *EMLo* compare with other downstream NLP task.

BERT [6]. *BERT* was proposed by *Google*. *GPT* can only learn one side semantics features. *ELMo* just concatenate bi-direction features generated by *BiLSTM*. *BERT* apply bidirectional *Transformer* that truely learn bidirectional context information. In our proposed method, we apply the pretrained *BERT* by using fine tuning to extract the feature.

Chapter 3

Methodology

In this section, we introduce our proposed method for SDP. Figure 3.1 is an overview of our method. we separate our method into six parts.

(a) **Data Preprocessing**. To extract the semantics model, we need to serialize elements of source code for both dataset and corpus files. And we also to filter out tokens with useless information.

(b) **Inputs Embedding**. Inputs embedding will covert token sequences into vectors using our pretrained *BERT* model.

(c) **Context Information Learning**. We will use *BiLSTM* to learn the context information from token vectors and generate hidden sequences.

(d) **Feature Generation**. We use three methods to generate semantics features,**Average Pooling**, **Max Pooling** and **Attention**.

(e) **Feature Concatenation**. We concatenate three generated features.

(f) **Training**. We use logistics regression to training the model.

3.1 Data Preprocessing

In this section, we preprocess our dataset and corpus that fit training; we need to process data and corpus to fit our model. To extract the semantics feature of a program, many research use ASTs (Abstract Syntax Trees) []to present the semantics of files. However, they select some forms of nodes from ASTs then concatenate with them, which destroyed

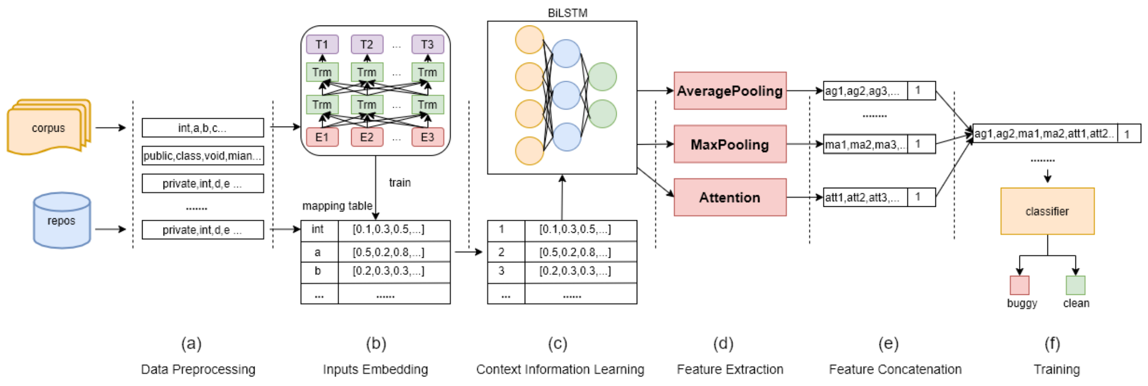


Figure 3.1: Proposed model overview

the order if original programs. Besides, since context information not only hidden in a single token but decided by the sequence of tokens. Based on these problems, we directly tokenize Java files. The step can be described as follow:

- Comment removal. We remove all blank lines and comments.
- Sequence generation. We directly tokenize all the data to generate the sequence for both datasets and corpus.
- We filter out all punctuation and **string** since we believe it contains little information.

During the processing, we remove those tokens that have useless information. such as the `import`,`package` tokens give no help for training.

3.2 Inputs Embedding

In a deep learning process, word embedding is a necessary stage for downstream tasks such as text classification, question answering, and machine translation. It is also necessary to embed inputs for our model to yield suitable data for training. Then, we need to embedding inputs data to generate vectors to present tokens.

3.2.1 BERT Pretraining Language Model

In language model, the probability of sequence of tokens $(t_1, t_2, ..., t_m)$ are

$$p(S) = p(t_1, t_2, ..., w_m) = \prod_{i=1}^{m} p(t_i | t_1, t_2, ..., t_{i-1}) \tag{3.1}$$

Training a neural network language model can obtain vector for token, whereas the pretrained vectors are fixed, which can not learn the semantics of tokens under different context. *BERT* pretraining model can train a character level context presentation, which can handle such problem and present the syntax feature for a sequence. Figure 3.1 demonstrate the structure of *BERT* model. *BERT* adopts bidirectional *Transformer* as encoder so that every tokens can merge both forward and backward information. In Relesed versions of *BERT*, base model contains 12 layers of *Transformer*, large model contains 24 layers of *Transformer*.

The input layer is the addition of token embedding, position embedding, and segment embedding. As timing feature is an important feature in NLP, *Transformer,* adopts a position embedding strategy to add timing information into token embedding, so that *BERT* can present the semantics of the same word in different contexts. Figure 3.4 shows how *BERT* embed inputs.

In position embedding phase, the output vectors can be derived from

$$PE = (pos, 2i) = sin(pos/1000^{2i/d_model}) \tag{3.2}$$

$$PE = (pos, 2i + 1) = cos(pos/1000^{2i/d_model}) \tag{3.3}$$

where the length of sequence is 512,d_model is 64,$2i$ indicates even position and $2i + 1$ indicates odd position. Figure 3.3 show the unit of *Transformer,* **Transformer** blocks is core part of **BERT** model. It is constructed by using attention mechanism.

Formula 4 shows The core part of the encoder block is self-attention.

$$Attention(\mathbf{Q,K,V}) = softmax(\frac{\mathbf{QK}^T}{\sqrt{d_k}})\mathbf{V} \tag{3.4}$$

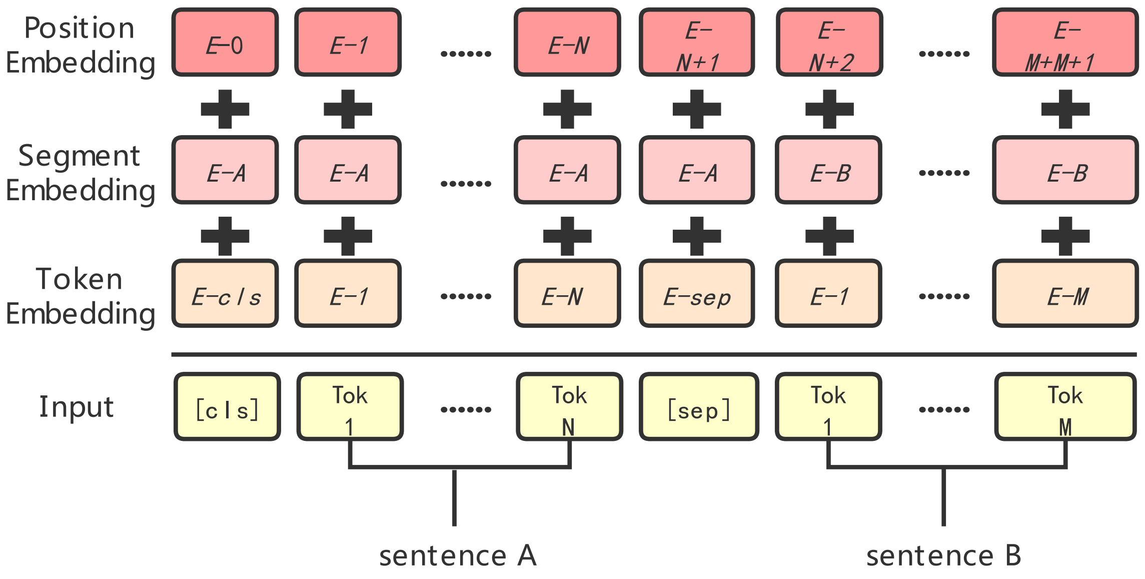


Figure 3.2: BERT input presentation. The embedding is sum of token embedding, segment embedding and position

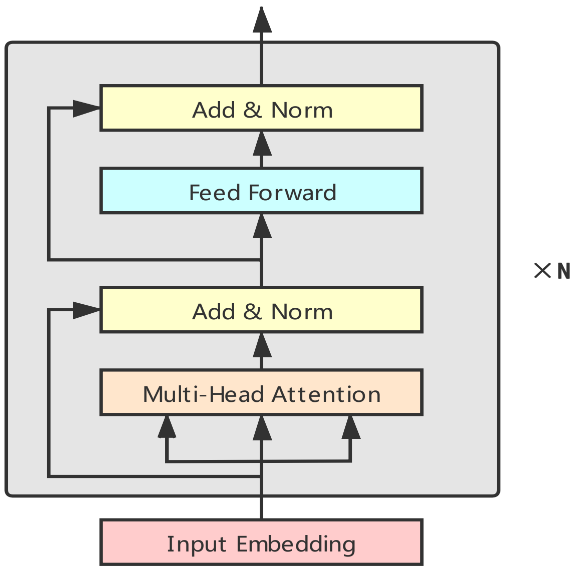


Figure 3.3: Transformer block

where **Q**,**K**,**V** are the input matrix of token vector. d_k is the dimension of vector, $\mathbf{QK}^T$ present the relationship between two input token vectors. The the result will be scaled by d_k and normalized by *softmax* function.

Apart from that, differ from other form attention, **Transformer** adopt **Multi-Head** which allow the model to jointly focus on the information from different representation subspace at different positions. details can be seen as formula 5 and 6.

$$Multi-Head(\mathbf{Q},\mathbf{K},\mathbf{V}) = Concat(head_1, head_2, ..., head_h)\mathbf{W}^O \tag{3.5}$$

$$head_i = Attention(\mathbf{QW}_i^{\mathbf{Q}}, \mathbf{KW}_i^{\mathbf{K}}, \mathbf{VW}_i^{\mathbf{V}}) \tag{3.6}$$

To alleviate the ability degradation of network, **Transformer** also adopt short-cut structure and **Normalization** layer. Formula 7 and 8 display the detail of *Feed Forward* layer.

$$LN(x_i) = \alpha \frac{x_i - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}} + \beta \tag{3.7}$$

$$FFN = max(0, xW_1 + b_1)W_2 + b_2 \tag{3.8}$$

Where α and β are parameters that need to learn, μ and σ are the mean and variance.

There are two pretraining tasks in *BERT* model, One is **Mask Language Model** (MLM), another is **Next Sentence Prediction** (NSP). Differ from **ELMo** such left-to-right language model token pretraining pattern, *BERT* randomly masked 15% tokens in each sequence. For each token to be masked. There three treating methods:

1. 80% time replace the selected token as `[mask]`.
2. 10% time, replace the selected token by other tokens.
3. 10% time, do nothing on the selected token.

Then, there are two sentences for each training example, and we suppose them sentence A and sentence B. *BERT* treats sentence B in two ways:(1) 50% time unchanged; (2) 50% time randomly replace by other sentences from the corpus. Figure 3.3 shows the detail that token embedding derive from the input.

3.2.2 Embedding Weight Initialization

After we pretrained the *BERT* model, we derive vocabulary and weights from the pretrained model. For our SDP task, we primarily convert token sequences into index sequence according to our obtained vocabulary. Since our vocabulary size is large, we restrict our vocabulary in a certain number **V**. For those indexes that are beyond **V**, we replace them with the index of `[unk]`. Besides, the length of token sequences will be set into fix length L. If sequence length is smaller than L, we will add the index of `[pad]`, otherwise we will cut the length into length L.

Then, we pass the index sequences into our pretrained *BERT* model to initialize the weight.

$$F : S \Longrightarrow E^d \tag{3.9}$$

where S is index sequence, E^d is embedding matrix, d is embedding size.

3.3 Context Information Learning

In feature extraction, the model previously extracts feature based on *BiLSTM* model. Figure 3.5 shows the process of extracting three forms of feature by using *BiLSTM*.

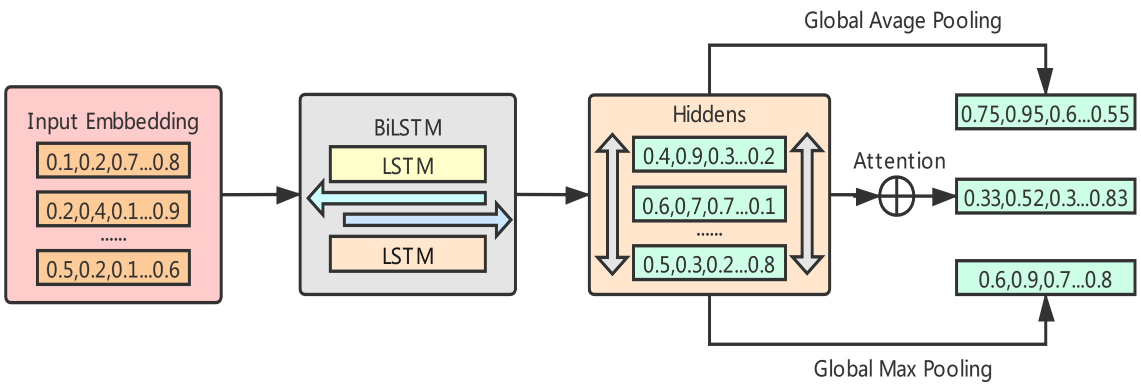


Figure 3.4: Feature extraction process. Extract three kind of feature from output of BiLSTM through global max pooling, global average pooling and attention mechanism

3.3.1 BiLSTM

For the input of standard RNN, currently hidden state computation relies on previous hidden state shown in the following formula:

$$h_t = \sigma(U * x_t, W * h_{t-1} + b) \tag{3.10}$$

where U, W are weights, σ is activation function and b is bias in the network. For h_t, it depends on h_{t-1}. Then with the length of input growing. It will occur gradient vanishing or gradient exploding problem. Because of such problems, standard RNN can not learn longer dependency contexts. LSTM is an upgrade form of RNN, which can learn long-distance dependency. The core design in LSTM is the forget gate and input gate. The mechanism is like this: When there is no important information pass into forget gate, outputs of forget gate is close to 1 while that is close to 0 for input gate. When there is important information in the sequence, the output of input gate is close to 1 while that is close to 0 of forget gate. By such a mechanism, LSTM can learn longer distance dependency semantics and syntax than standard RNN.

3.4 Feature Extraction

In SDP, it is hard to locate the bug code, and defective codes do not usually depend on previous code information but also depend on the next code information. So, To comprehensively learn the semantics information of context, we adopt *BiLSTM*.

3.4.1 Attention Mechanism

The output of *BiLSTM* we can get a sequence of hidden vectors. Since not all tokens share weights of information. To deeply learn extract the information of semantics node, we adopt the attention mechanism on those vectors. Equation 3.11 illustrates the process of attention mechanism.

$$\begin{aligned} u_{it} &= tanh(W_n h_{it} + b_n) \\ \alpha_{it} &= \frac{exp(u_{it}^{\top} u_n)}{\sum_t exp(u_{it}^{\top} u_n)} \\ c_i &= \sum_t \alpha_{it} h_{it}. \end{aligned} \tag{3.11}$$

Where W_n is weight,u_{it}

3.4.2 Global Max Pooling & Global Average Pooling

To reduce the parameter size of the network, we apply global average/max pooling on the output of hidden sequences. Apart from that we need to extract important node information, we also need to extract semantics of sequence from different view.

3.5 Feature Concatenation

After we obtained the feature from three methods described in 3.3 section, we concatenate these three kinds of features and pass them to the logistics regression classifier.

$$f = Att \oplus Max \oplus Avg \tag{3.12}$$

The total dimension of this feature is the sum of these three types feature's dimensions, where $Att, MaxandAvg$ are features generated by attention global max pooling and global average pooling respectively.$\oplus$ is concatenation operation.

3.6 Training

In our training method. We use logistics regression and *softmax* for classification. The formula can be shown as follow:

$$\begin{aligned} p(y|f) &= softmax(W * f + b) \\ o &= \arg\min_{y}(p(y|f)) \end{aligned} \tag{3.13}$$

where f is the feature, W, b are weight and bias respectively, since we have two classes, after we calculate the probability for two labels, The the label with larger probability of will be the predicted label for the instance. For the loss function, we select cross entropy shown in following formula:

$$J = -\sum_{n}[y * log(p) + (1 - y) * log(1 - p)] \tag{3.14}$$

Chapter 4

Experimental Design

To answers the research questions that we raised in chapter 1, we designed the relevant experiments to evaluate our assumption. Most experiments were conducted on our personal computers. We conduct BERT training with 4 NVIDIA RTX 2080Ti. Here we illustrate how we design the experiments for each research question.

4.1 Datasets

To evaluate our model and other baselines, we decide to adopt PROMISE [14] dataset. However, since it can not be accessed now, we use a website called way back machine to visited its history website and then downloaded the defect dataset, base on each project name and its version, we found its source code. We dropped files that can not be found base on the file name and path from defect datasets. Table 4.1 shows the overview of the dataset. And the description of the data shown in Table 4.2

Table 4.1: Dataset Overview

Project	Verison	Average	Average(processed)	Defect rate
ant	1.3 $\sim$ 1.7	338	307	0.23
camel	1.2, 1.4, 1.6	815	367	0.45
ivy	1.1, 1.4	176	151	0.32
jedit	3.2.1 4.1 $\sim$ 4.3	360	325	0.17
lucene	2.2, 2.4	293	265	0.62
poi	1.5, 2.0, 2.5	312	279	0.50
synapse	1.1, 1.2	239	212	0.33
velocity	1.4.1, 1.6.1	213	182	0.59
xalan	2.4 $\sim$ 2.7	830	739	0.58
xerces	1.2.0, 1.3.0, 1.4.4	410	224	0.36

Table 4.2: Statistical dataset columns description

Columns	Description
LOC	The number of methods in the class.
DIT	The position of the class in the inheritance tree.
NOC	The number of immediate descendants of the class.
CBO	The value increases when the methods of one class access services of another.
RFC	Number of methods invoked in response to a message to the object.
LCOM	Number of pairs of methods that do not share a reference to an instance variable.
LOCM3	If *m,a* are the number of methods,attributes in a class number and is the number of methods accessing an attribute, then locm3=$(((1/a)\sum_j^a \mu(a) - m)/(1-m)$.
NPM	The number of all the methods in a class that are declared as public.
DAM	Ratio of the number of private (proteced) attributes to the tatal number of attributes.
MOA	The number of data declarations (class fields) whose types are user defined classes.
MFA	Number of methods inherited by a class plus number of methods accessible by member methods of the class.
CAM	Summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods.
IC	The number of parent classes to which a given class is coupled.
CBM	Total number of new/redefined methods to which all the inherited methods are copuled
AMC	The number of JAVA byte codes.
Ca	How many other classes use the specific class.
Ce	How many other classes are used by the specific class.
Max(CC)	Maximum McCabe's Cyclomatic Complextity values of methods in the same class.
Avg(CC)	Average McCabe's Cyclomatic Complextity values of methods in the same class.
LOC	Measures the volume of code.

Table 4.3: Staticstical dataset column description

4.2 Corpus Pretraining

it is necessary to pretrain a language model to evaluate our model with other deep learning baselines. On the other hand, Since there no pretrained model for source code, especially for Java source codes, we pretrained mini-version of *BERT* to satisfy our requirement. For the counterpart of *BERT*, we also need to pretrain *Word2vec* for our research.

4.2.1 Corpus

To train a language model, we adopt the part of **BigCode** [2], to generate the corpus we need. Table 4.3 is a description of our processed corpus.

Table 4.4: Description of processed corpus

Number of Line	**Total Tokens**	**Vocabulary Size**	**Average Sequence Length**
306,522	84,955,239	2,047,760	277

4.2.2 Word2vec

Word2vec is one of the most frequently used models in NLP domain. Since many research about source code embedding via using *Word2vec* model, our deep learning-based model will adopt the tokens vector table that generated by *Word2vec* model, we use *gensim* [1], an open-source package including *Word2vec* model. Table 4.4 shows the training settings of the Word2vec model.

Table 4.5: Word2vec Pretraining Settings

Dim Size	Window	min_count	epochs
300	5	5	15

4.2.3 BERT

Since there no source code *BERT* pretrained model available, we pretrained small *BERT* model for our experiments, Table 4.5 show the settings for our *BERT* pretraining.

Table 4.6: BERT pretraining Settings

Total Parameter	hidden layer	batch size	epoch
264,241,330	64	8	2

Due to the limitation of time, the training configuration we select meet our training requirement. We open source tools *BERT-pytorch* [2] to complete the pretraining task. We firstly construct *BERT* style vocabulary and then use the vocabulary train our model. The training time took one day.

4.3 Evaluation Metrics

To evaluate the ability of defect prediction for SDP model, we adopt three metrics to measure their capability:

- **Precision** measure the ability that how well a model that can correct defect instance in all predicted defect label.
- **Recall** measure how well that defect instance can be predicted successfully.
- **F1** balances the **Precision** and **Recall** when we need to take both into account.

Below description are the equations of there metrics.

$$\textbf{Precision} = \frac{TP}{TP + FP} \tag{4.1}$$

[1]https://radimrehurek.com/gensim/

[2]https://github.com/codertimo/BERT-pytorch

$$\textbf{Recall} = \frac{TP}{TP + FN} \tag{4.2}$$

$$\textbf{F1} = \frac{2 * Precision * Recall}{Precison + Recall} \tag{4.3}$$

where *TP*, *FP* and *FN* are *True Positive*, *False Positive* and *False Negative*, respectively. *True Positive* is the number of predicted defective example whose true label is defect, *FP* is number of predicted defective example whose true label is clean. *False Negative* is the number of predicted clean example whose true label is defect.

4.4 Baseline metrics

We use two base metrics for experiments for traditional methods.

1. **Tokens**: We generate this token feature by transforming sequence tokens into sequences of the index according to token vocabulary.
2. **Stats**: Statistical feature that directly from defect data described in section 4.1.

4.5 Baseline Methods

The following are the baseline methods we selected to compare with our *BERT* based method.

1. **TextCNN**: Deep learning method for text classification. In our experiments, we use three kind of filters whose size are 2,3 and 4, respectively. The number of every filter is 32.
2. **BiLSTM+ATT**:Widely applied deep learning method for text classification, we use sentence classification for our experiment.
3. **Token+LR**: We use **Tokens** feature to train logistics regression model.
4. **Stat+LR**:We use **Stats** feature to train logistics model.

4.6 Within-Project Defect Prediction

Within-Project Defect Prediction (WPDP) is the one form of SDP that conducted for an individual project. WPDP is a general evaluation strategy for many SDP researches, in our dataset, each project contains several history versions. Since WPDP require at least two history version, we filter out those projects do not meet this requirement. We trained a model by using previous history versions and evaluate models using the latest version in the project dataset. For those projects with multi-version, we concatenate them into the training set. We use the feature generated by our pretrained model.

4.7 Cross-Project Defect Prediction

In SDP domain, the shortage of defect dataset is a key point for the development of research, to relieve this problem, Cross-Project Defect Prediction (CPDP) is another form to evaluate SDP models. CPDP train a model using one or more projects dataset and then use another project to evaluate the performance of the trained model, In our experiments,

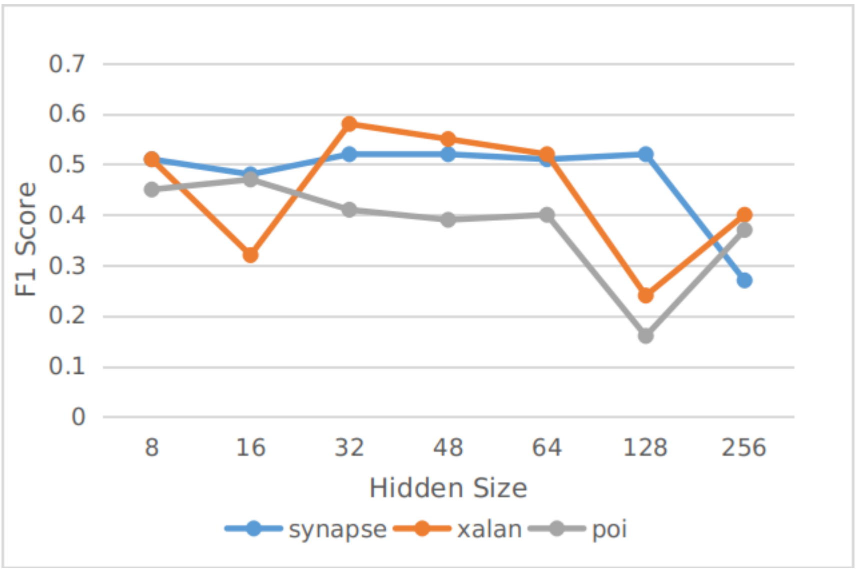


Figure 4.1: The F1 score variation with hidden size

we have ten projects, we first concatenate each history version dataset of a project into project dataset, when conducting the CPDP experiments, each project dataset is like to be the target project dataset. Once a project is selected as the target project, the rest of the nine projects dataset will be concatenated as a training set for models.

4.8 Parameters Setting

The parameter setting for the deep learning method can significantly affect the performance of models. In our proposed model, the number of hidden sizes of *BiLSTM* will model affect the performance of the proposed model. To keep fair of our experiment, the number of hidden sizes we select that can ensure our model can achieve a stable F1 score. For this purpose, we randomly select three projects and conducted CPDP experiments with our model. Figure 4.1 shows the F1 score variation with hidden sizes of *BiLSTM*.

We can see that the hidden size from 32 to 64; the performance of F1 scores is relatively stable. For **xalan** and **poi**, the F1 score is rapidly degrade when the hidden size over 64. Base on the observation, we select 64 as our hidden size for all deep learning models. Then the deep learning models parameter setting and training setting for three deep learning models can be seen in Table 4.6.

Parameter	BERT+BAMA	BiLSTM+ATT	TextCNN
embedding size	64	300	300
input length	512	512	512
vocabulary size	100,000	100,000	100,000
optimizer	Adam	Adam	Adam
learning rate	0.005	0.005	0.005
epoch	5	5	5
hidden size	64	64	
filter size			2,3,4
number filter			32
loss function	cross entropy	cross entropy	cross entropy

Table 4.7: Parameter setting for deep learning

Because of the limitation of our training environment, the embedding size in the proposed model is 64. For optimizer we select *Adam* [16], a optimization algorithm that can adaptively adjust learning rate. For the traditional machine learning method, we selected the default parameters using *sciki-learn* [3], machine learning package.

4.9 Data Augmentation

In our dataset, the amount of instance of most projects is no more than 1,000, previous researches many focus on data imbalance. However, the entire dataset is still not enough. In addition, deep learning method needs large volume dataset to train effective model; otherwise, it would be overfitting and is not convincing. Thereby, to construct well-trained data, we learn from the method proposed by Li [31] to generate specific volume data we need. We generate sequences from one sequence by using the following four strategies:

1. **Similiar Token Replace**. We randomly replace similar tokens for software defect prediction with the large corpus.
2. **Randomly Insert**. We randomly insert tokens into the original sequence.
3. **Randomly Swap**. We randomly swap two tokens in a sequence.
4. **Randomly Delete**. We randomly delete tokens in sequence.

The detail can show by figure 3.2. For each kind of strategy, we set α to control the percentage of a sequence that should be augmented. For example, if α is 0.1, and the length of the sequence is 100, then 10 words will be changed for strategy 1, 2, and 3, α will also be the probability for 4, which indicates the probability that a word will be deleted. And we also set the parameter n indicate that generates n augmented sequence. For **similar token replace**, we randomly select token and pass it to our pretrained *Word2vec* model. It will return the top 10 most similar token; Then, we will select one of them to replace the token we selected. For **Randomly insert**, the token that randomly selects will be pass into *Word2vec* model, and the one of its return will be inserted in pre-position of the selected token.

[3]https://scikit-learn.org/

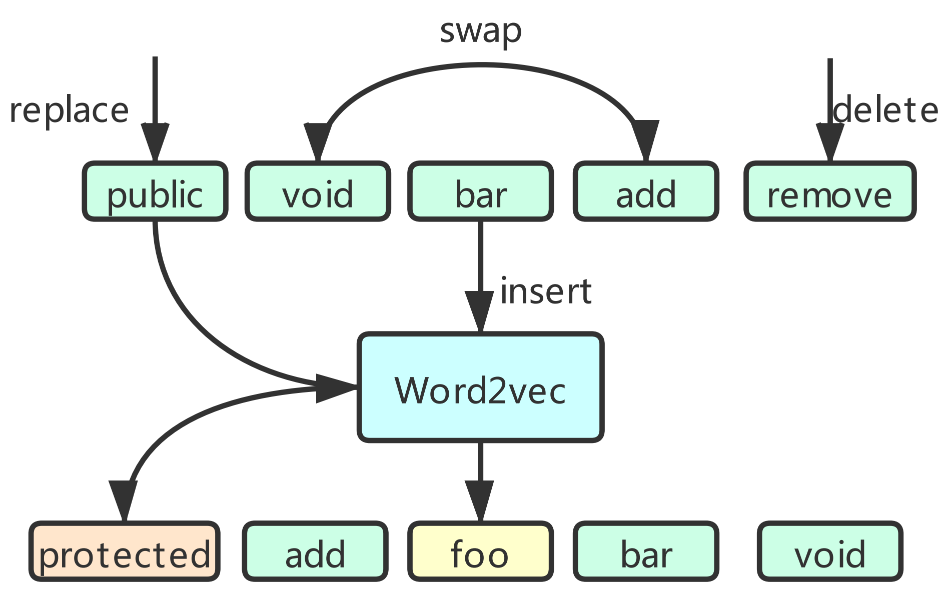


Figure 4.2: Four strategies of data augmentation for sequence data.

4.10 Experiment for RQ1

To verify whether the data augmentation strategy can improve the performance of data, we adopted our data augmentation strategy to generate more instance for each project. Since our data augmentation strategy is suitable for a small dataset, we conduct WPDP experiment to verify whether our data augmentation can improve the performance of models. On the other hand, many augmented data may not be able to learn the raw data semantics feature. So we need to find proper augmented times to ensure the maximum performance of models. In this experiment, we select the number range of augmentation are 2,4,8,16,32 and 32. To eliminate those project whose instance's amount are more than 1000 since we believe that data augmentation is more effective on a small dataset. In this experiment, we conduct WPDP experiment for our research, since the volume of training data is much larger than that of WPDP.

4.11 Experiment for RQ2

To compare with other baseline methods, we conduct the experiment on both WPDP and CPDP we use the F1 score metrics to measure the performance. For the pretrained *BERT* methods, we use the fine-tuning strategy to train our model,i.e., in the training process, the weight of token embedding will be adjusted by training loss, while position embedding will not be adjusted;and we calls our model *BERT+BAMA*.

4.12 Experiment for RQ3

To verify which kind of embedding method is effective, we compared with our trained mini-version *BERT* and *Word2vec* embedding methods, we conducted both WPDP and CPDP for **TextCNN** and **BiLSTM+ATT**. Differ from experiment for RQ2, in this experiment, the weight of embedded tokens will not be changed during whole training.

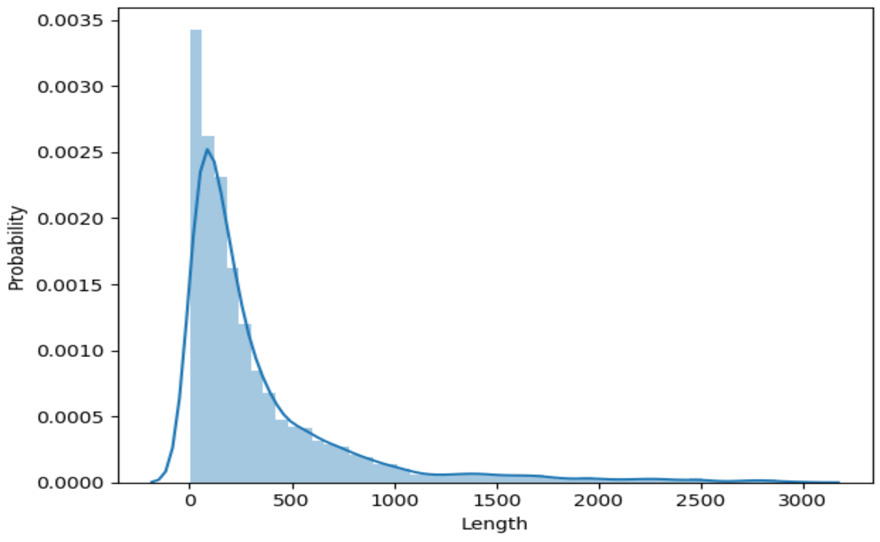


Figure 4.3: length distribution of full token data

4.13 Experiment for RQ4

Since many pieces of research about deep learning-based SDP method are based on AST mentioned in section 1. They select the specific kind of node from AST. Shown in Table 4.7. But which kind of processing method is better for prediction is unclear, we design the experiment to investigate the performance of both methods on CNN with **Word2vec** embedding method. In this thesis, AST-based method we call **AST-node**, processing method, directly tokenize source codes, we call **full-token**.

MethodInvocation	ForStatement	SuperMethodInvocation
PackageDeclaration	BreakStatement	InterfaceDeclaration
ClassDeclaration	ReturnStatement	MethodDeclaration
ConstructorDeclaration	SynchronizedStatement	VariableDeclarator
FormalParameter	SwitchStatement	BasicType
CatchClauseParameter	StatementExpression	MemberReference
SuperMemberReference	CatchClause	ReferenceType
IfStatement	SwitchStatementCase	WhileStatement
DoStatement	EnhancedForControl	AssertStatement
ContinueStatement	ThrowStatement	TryStatement
BlockStatement	TryResource	CatchClauseParameter
ForControl		

Table 4.8: Selected AST node type

We primarily calculate the length distribution of two pieces of the processed dataset. We compute the distribution of length of both process methods.

For the length of distribution of the full-token method, we notice that there some outliers whose length over 20,000, we delete them, and select the length of 3,000, which can cover 98% instance. Figure 4.2 shows the length of the distribution of the full-token method.

For the length of distribution of the AST-node method, since the AST-node method

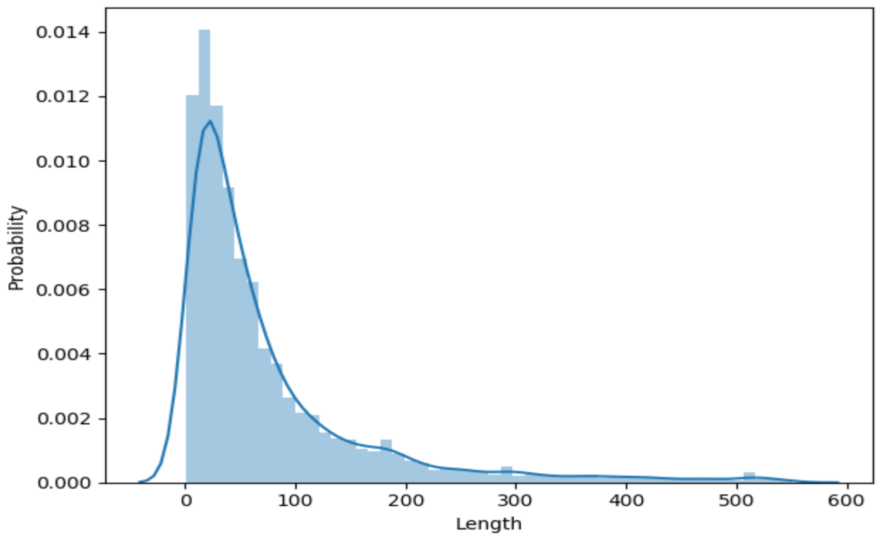


Figure 4.4: length distribution of full AST-node data

filter out some tokens in an instance, the length will be much shorter than that of the full-token method. Similarly, there also some outliers in AST-node tokens. We select the length of 550, which also covers about 98% of the instance. The distribution of length of AST-node method can be shown in Figure 4.3

To confirm the length of the corresponding length of coverage for each project in WPDP experiment. Table 4.8 and Table 4.9 shows the corresponding specific project for full-token and AST-node data type, respectively. To observe the distribution intuitively, the length distribution of each project for two data types shown in Figure 4.5 and Figure 4.6

Project	50%	60%	70%	80%	90%
xerces	152	196	386	705	1786
synapse	192	236	295	431	590
xalan	206	279	416	687	1424
camel	86	118	156	230	364
ivy	186	224	303	435	581
velocity	141	176	242	394	554
ant	195	267	400	602	886
poi	204	289	352	451	671
lucene	180	261	326	440	683
jedit	249	324	488	760	1206

Table 4.9: Corresponding length of coverage for full-token data type

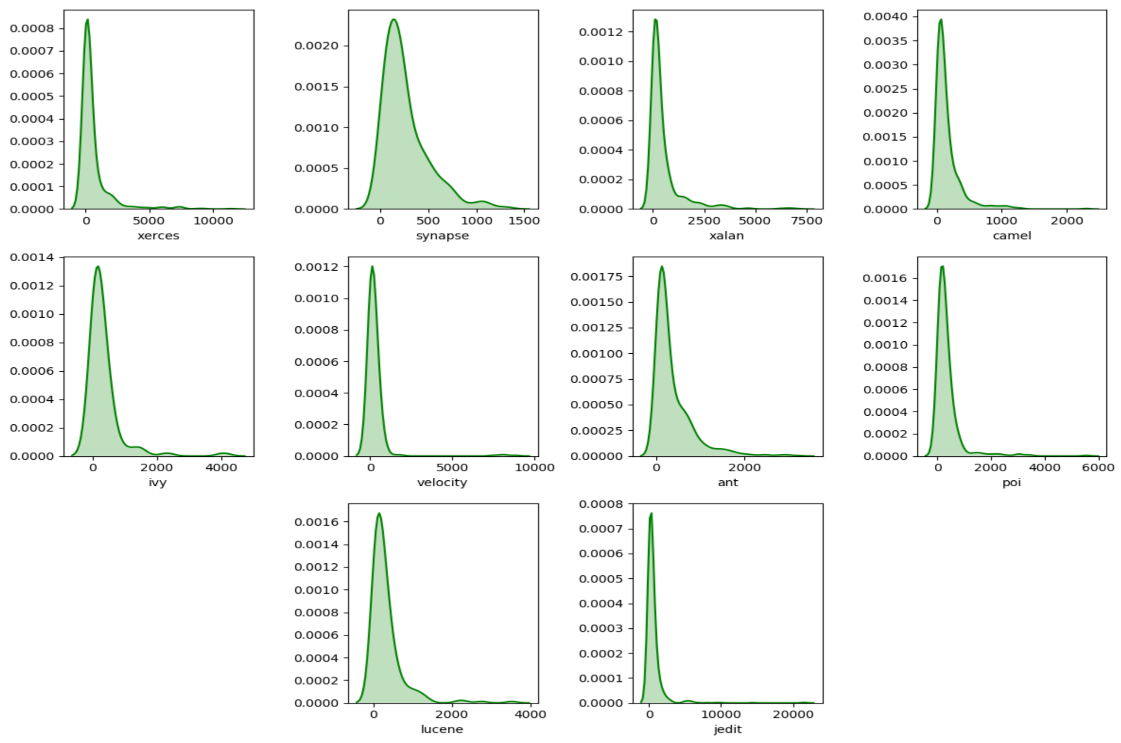


Figure 4.5: Length of distribution of training data of each project in full-token data type

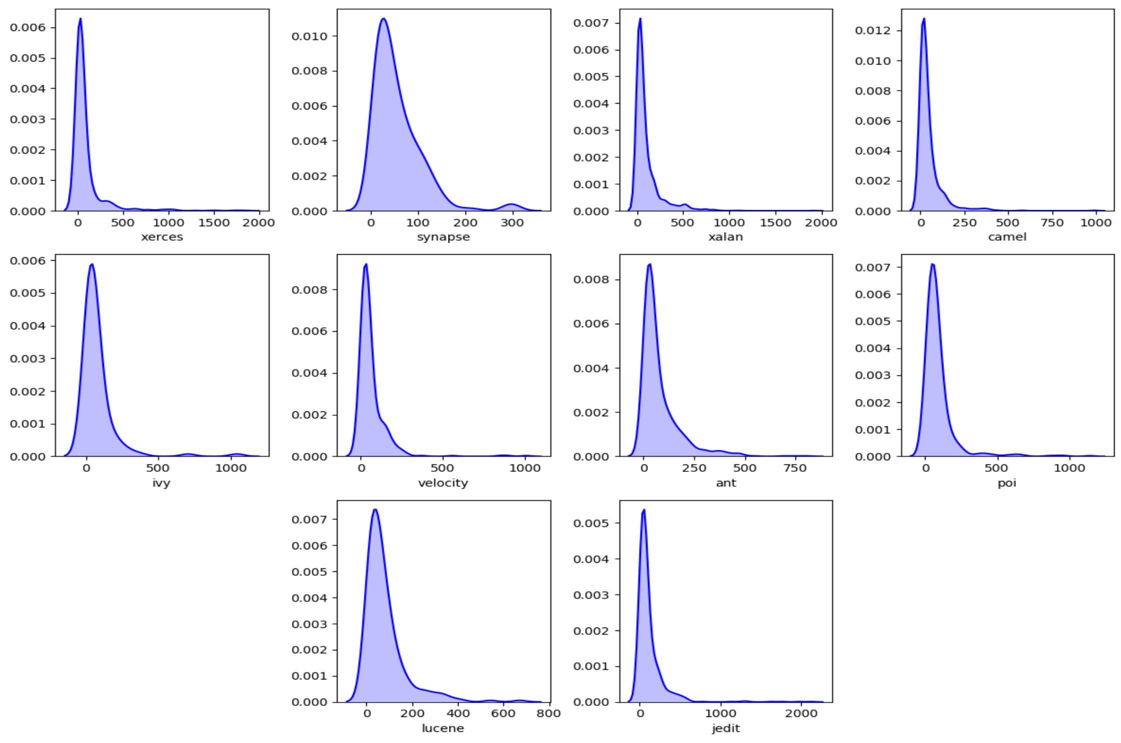


Figure 4.6: Length of distribution of training data of each project in AST-node data type

Project	50%	60%	70%	80%	90%
xerces	32	43	60	90	210
synapse	40	53	68	87	117
xalan	45	64	92	148	260
camel	24	33	44	66	117
ivy	45	55	69	89	173
velocity	36	44	55	86	151
ant	43	58	80	124	193
poi	62	74	89	114	166
lucene	53	68	84	117	178
jedit	61	74	101	166	252

Table 4.10: Corresponding length of coverage for AST-node data type

Then, we confirm the corresponding length of coverage of CPDP experiment and they are shown in Table 4.10

Datatype	50%	60%	70%	80%	90%
full-token	180	250	350	550	1000
AST-node	44	58	80	120	200

Table 4.11: Corresponding length of coverage for CPDP experiment

Chapter 5

Results

5.1 RQ1: Can data augmentation improve the performance of deep learning model?

To answer the research question that we conducted an experiment for RQ1. Figure 5.1 shows the variation trend of F1 score of 4 projects **xerces**, **synapse**, **camel** and **ivy**. The whole trend of the F1 score is growing with the times of augmentation. Compare with original data. F1 scores of 32 times original dataset of 4 project are 0.85, 0.91, 0.81 and 0.80, which improved 0.27 ,0.24, 0.2 and 0.54 respectively, compared with original dataset. especially For **ivy** project, whose amount of instance is 453. This is significantly small for deep learning. The results show that data augmentation the more data, the higher performance for our model. Besides, precision score also improved from 0.48, 0.59, 0.56 and 0.69 from 0.89, 0.88, 0.76 and 0.88 for 4 projects, respectively. Table 5.1 displays the precision score through different boosting times. For recall score, data augmentation also improve that data from data augmentation from 0.74, 0.76, 0.64 and 0.16 to 0.82, 0.93, 0.84 and 0.74 respectively, shown in Table 5.2.

Base on the result given above, we can give the answer to RQ1 that:

Our proposed data augmentation method can improve the performance of deep learning models for SDP.

Table 5.1: Precision of data augmentation for WPDP experiment

Project	0	2	4	8	16	32
xecers	0.48	0.59	0.68	0.76	0.83	0.89
synapse	0.59	0.58	0.61	0.72	0.80	0.88
camel	0.56	0.59	0.61	0.66	0.71	0.76
ivy	0.69	0.64	0.38	0.68	0.74	0.88

5.2 RQ2:Can our model outperform other models including traditional metrics based model and deep learning-based model?

To answer the question, we experiment to compare with other models on Both WPDP and CPDP experiment. We compare the F1 score for our model and baselines. Table 5.3

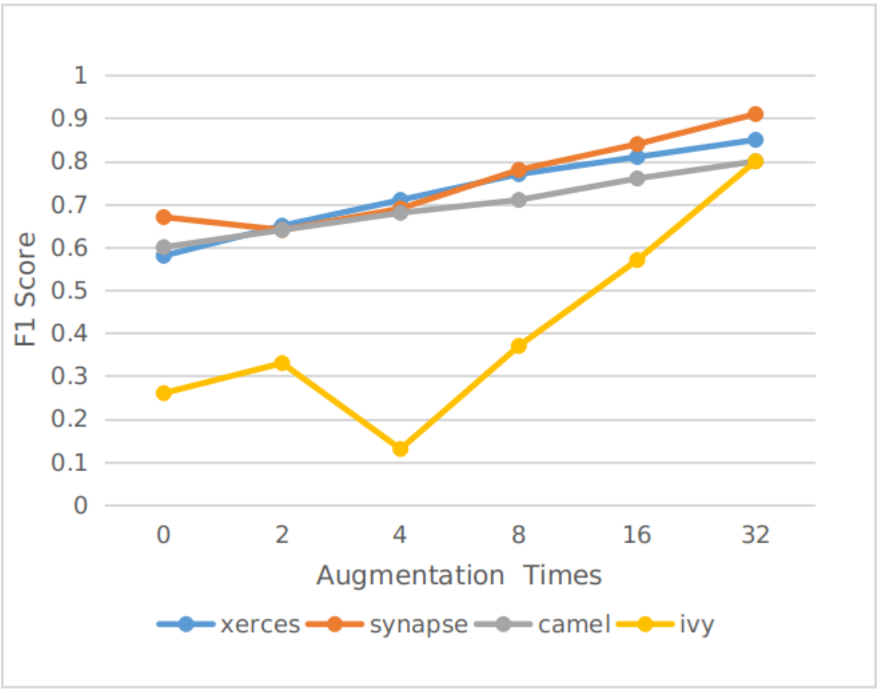


Figure 5.1: F1 score of Data augmentation on WPDP experiment

Table 5.2: Recall of data augmentation for WPDP experiment

Project	0	2	4	8	16	32
xecers	0.74	0.73	0.76	0.78	0.80	0.82
synapse	0.76	0.72	0.80	0.85	0.87	0.93
camel	0.64	0.68	0.74	0.77	0.81	0.84
ivy	0.16	0.22	0.08	0.25	0.46	0.74

and Table 5.4 show the the F1 score of WPDP and CPDP respectively. Here, In WPDP experiment, **BERT+BAMA** model achieve 6 out 10 highest score for both WPDP and CPDP experiment. Note that 0.0 represent that the predicted label is clean. Since it is meaningless, we set it 0.0, For **BERT+ATT** model in WPDP experiment, the highest F1 score is 0.77 achieved on **lucene** project. The average F1 score of **BERT+BAMA** model is 0.446 which is 0.088 higher than that of **BiLSTM+ATT** . 0.023 higher than **TextCNN**, 0.09 higher than **Token+LR** model and 0.07 higher than that of **Stat+LR** model. For CPDP experiment, project **xalan** and **camel** achieve the highest F1 score with the value of 0.58 of **BERT+BAMA**. The average F1 score of **BERT+BAMA** is 0.436 which is 0.126 higher than **BiLSTM+ATT**, and 0.047 higher than of **TextCNN** model,0.102 higher than that of **Token+LR** model and 0.07 higher than that of **Stat+LR** model.

Then we can draw the answer:

Our proposed model performs better than other models including both deep learning-based and traditional model.

Table 5.3: WPDP F1 score

Project	BERT+BAMA	BiLSTM+ATT	TextCNN	Token+LR	Stat+LR
xecers	0.20	0.07	0.39	**0.46**	0.21
synapse	**0.52**	0.37	**0.48**	0.45	**0.48**
xalan	0.36	**0.55**	0.51	0.36	0.39
camel	0.48	0.49	0.50	**0.51**	0.46
ivy	**0.23**	0.20	0.21	0.13	0.20
velocity	**0.65**	0.53	0.43	0.57	0.64
ant	0.48	0.35	0.42	0.23	**0.54**
poi	**0.67**	0.18	0.52	0.0	0.12
lucene	**0.77**	0.74	0.73	0.76	0.73
jedit	**0.10**	**0.10**	0.04	0.09	0.05

Table 5.4: CPDP F1 score

Project	BERT+BAMA	BiLSTM+ATT	TextCNN	Token+LR	Stat+LR
xecers	**0.39**	0.37	0.38	0.29	0.28
synapse	**0.54**	0.51	0.49	0.38	0.40
xalan	**0.58**	0.33	0.45	0.42	0.32
camel	**0.58**	0.38	0.30	0.27	0.25
ivy	**0.52**	0.19	0.38	0.37	0.43
velocity	0.39	0.14	**0.40**	0.19	0.28
ant	0.44	0.21	0.46	0.36	**0.51**
poi	0.33	0.60	0.31	0.34	**0.56**
lucene	**0.47**	0.12	0.34	0.42	0.26
jedit	0.12	0.25	0.38	0.35	**0.37**

5.3 RQ3:Can inputs embedding with our pretrained *BERT* model outperform *Word2vec* model

The Table 5.5 shows the F1 score result of **BERT** model and result. For WPDP experiment of **TextCNN** model, the lowest F1 score is 0.10 based on *Word2vec* model, while that is 0.07 of *BERT* based. On the other hand, the highest F1 score of *Word2vec* is 0.59 while that is 0.6 of *BERT* based. Both two embedding method achieve 5 higher score when compare with each other on one project. However, For the **BiLSTM+ATT** model for WPDP experiment. 9 out 10 the *Word2vec* based achieve higher F1 score compare with that of *BERT* model. Similarly, compare the performance of CPDP experiment with two models. For **TextCNN** based on **BERT**, only three project **synapse**,**camel** and **poi**, whose F1 score is over 0.1, while the lowest of F1 score of *Word2vec* based is 0.34. we. For **BiLSTM+ATT** of CPDP experiment, *BERT* based slightly win with the 6 out 10 higer F1 score. We can draw the conclusion that

Our pretrained BERT model are not perform well in both WPDP experiment and CPDP experiment based on TextCNN and BiLSTM model

Table 5.5: F1 score of models comparison between *BERT* and *Word2vec* embedding methods of TextCNN and BiLSTM+ATT model for bot WPDP and CPDP experiment

Project	WPDP				CPDP			
	TextCNN		BiLSTM+ATT		TextCNN		BiLSTM+ATT	
	BERT	Word2vec	BERT	Word2vec	BERT	Word2vec	BERT	Word2vec
xecers	**0.41**	0.03	0.28	**0.29**	0.20	**0.37**	0.39	**0.40**
synapse	**0.60**	0.32	0.00	0.41	0.36	**0.51**	**0.50**	0.46
xalan	0.43	**0.59**	0.32	**0.65**	0.02	**0.40**	**0.51**	0.30
camel	0.36	**0.50**	0.44	**0.45**	**0.50**	0.43	0.29	**0.30**
ivy	0.13	**0.21**	0.13	**0.19**	0.03	**0.42**	**0.30**	0.26
velocity	**0.60**	0.53	0.49	**0.53**	0.03	**0.41**	**0.27**	0.19
ant	0.30	**0.40**	0.57	**0.56**	0.03	**0.48**	0.27	**0.32**
poi	**0.37**	0.18	0.0	0.20	**0.23**	**0.49**	0.26	**0.33**
lucene	**0.62**	0.54	0.76	0.76	0.01	**0.36**	**0.22**	0.05
jedit	0.07	**0.10**	0.10	**0.11**	0.07	**0.34**	**0.34**	0.19

5.4 RQ4: Which data preprocessing method is better for Software defect prediction?

Figure 5.2 and Figure 5.3 show the average F1 score change with the length of coverage of two types of data processing methods under two different types of experiments, respectively. In the WPDP, the full-token data type is at the highest point with 90 length coverage. From 60% to 90%, average F1 scores of full-token variate stably, without no obvious growth or declination while AST-node type is stably growing at the same period. For CPDP, F1 scores of full-token type are gradually growing until 80% length of coverage. As for AST-node, data type fluctuates overall length of coverage.

From Figure 5.3, both full-token and AST-node data type, the longer length is, the better for F1 score improvement. In addition, the best length of coverage is 90% for both data types in the WPDP experiment, while that is 80% and 70% of full-token and AST-Node respectively in the CPDP experiment. Taking consider that in the WPDP experiment, data size is smaller than CPDP, low length coverage will contain little information for a sequence, which will result in low performance. Especially for AST-node data type, increasing speed is obviously faster than full-token data type in WPDP experiment. In the CPDP experiment, the performance of the AST-node data type is unstable, while that of full-token is stably increasing until 80% length of coverage.

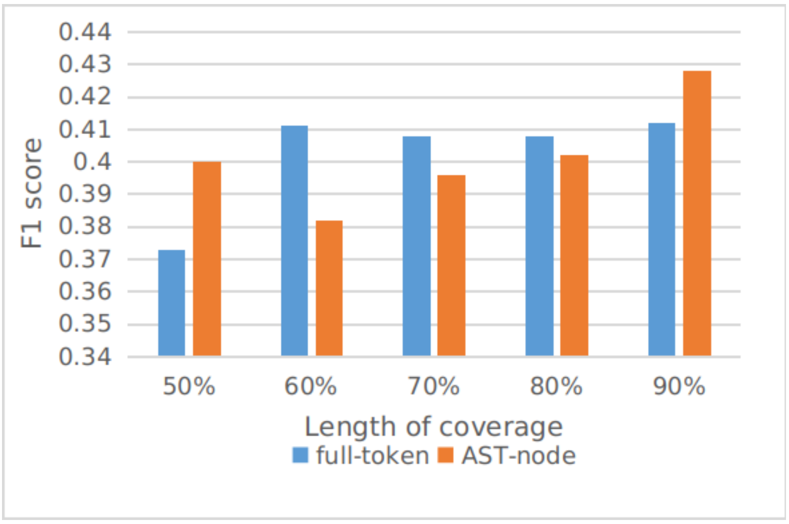


Figure 5.2: Average F1 score over 10 projects for WPDP experiment

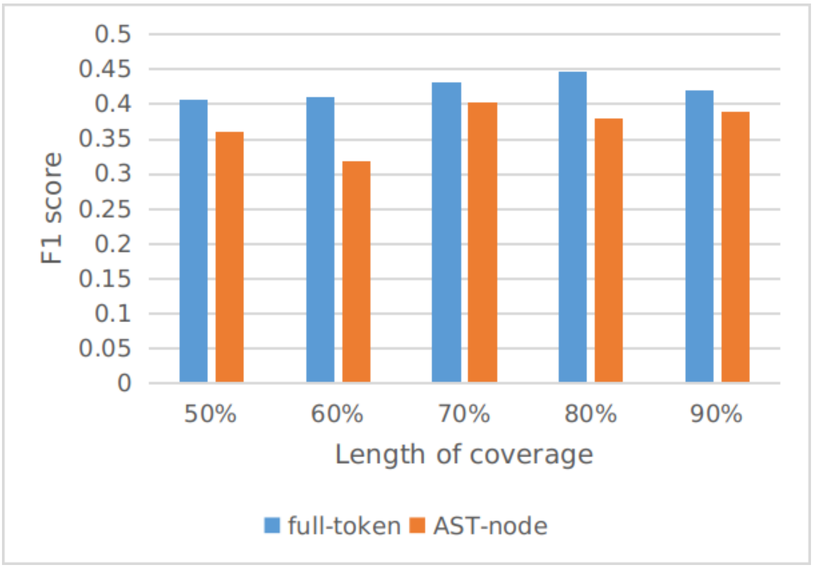


Figure 5.3: Average F1 score over 10 projects for CPDP experiment

Table 5.6 and Table 5.7 show F1 scores of 10 projects of two data types in WPDP and CPDP experiments, respectively. In two tables, "token" and "AST" denote full-token and AST data types, respectively. In Table 5.6, with the increase of length coverage, the number of higher F1 scores gradually more than AST-node until the 80% length coverage. For the CPDP experiment, the full-token data type achieves the number of higher than AST-node all the time. Especially at the 80% length of coverage, full-token data type achieves 9 higher F1 scores. Then we can answer RQ4 that

Full-token processing method achieve better performance than the AST-node method. Generally, the sequence with extensive length coverage can achieve better performance.

Table 5.6: F1 score of 10 projects for both types in CPDP experiment

Project	50%		60%		70%		80%		90%	
	token	AST	token	AST	token	AST	token	AST	token	AST
xerces	**0.31**	0.28	**0.29**	0.24	**0.31**	0.19	**0.28**	0.15	0.31	**0.33**
synapse	0.34	**0.41**	0.44	**0.45**	**0.46**	0.40	**0.48**	0.44	0.46	**0.49**
xalan	0.47	**0.63**	**0.66**	0.41	**0.57**	0.52	**0.54**	0.5	**0.57**	0.45
camel	0.43	**0.45**	**0.46**	0.44	0.42	**0.49**	**0.45**	0.44	0.45	**0.51**
ivy	**0.2**	0.18	**0.22**	0.19	**0.21**	0.17	**0.21**	0.2	**0.22**	0.19
velocity	0.56	**0.57**	0.53	**0.55**	**0.60**	0.58	**0.59**	0.58	0.58	**0.59**
ant	**0.40**	0.38	**0.49**	0.24	**0.55**	0.44	**0.55**	0.53	**0.52**	0.5
poi	0.29	**0.30**	0.26	**0.45**	0.19	**0.38**	0.22	**0.35**	0.23	**0.37**
lucene	0.66	**0.67**	0.68	**0.69**	**0.71**	0.65	0.68	**0.70**	**0.7**	0.69
jedit	0.07	**0.13**	0.08	**0.16**	0.06	**0.14**	0.08	**0.13**	0.08	**0.16**

Table 5.7: F1 score of 10 projects for both types in WPDP experiment

Project	50%		60%		70%		80%		90%	
	token	AST	token	AST	token	AST	token	AST	token	AST
xerces	**0.45**	0.37	**0.38**	0.31	**0.39**	0.29	**0.48**	0.31	**0.33**	0.26
synapse	**0.53**	0.37	**0.53**	0.47	**0.48**	0.46	**0.51**	0.48	**0.53**	0.41
xalan	0.36	**0.38**	**0.41**	0.32	0.49	**0.53**	0.5	**0.52**	0.29	**0.51**
camel	**0.54**	0.43	0.38	**0.4**	0.42	**0.44**	**0.39**	0.38	**0.47**	0.37
ivy	**0.32**	0.3	**0.37**	0.25	0.4	0.4	**0.42**	0.37	**0.45**	0.37
velocity	0.41	**0.52**	**0.48**	0.3	0.43	**0.48**	**0.49**	0.36	**0.42**	0.39
ant	**0.38**	0.37	0.35	0.35	**0.42**	0.37	**0.41**	0.32	**0.46**	0.4
poi	**0.57**	0.35	**0.57**	0.42	**0.5**	0.47	**0.53**	0.52	0.53	**0.54**
lucene	0.19	**0.29**	**0.36**	0.18	**0.45**	0.34	**0.39**	0.28	**0.34**	0.3
jedit	**0.32**	0.23	**0.28**	0.18	**0.34**	0.25	**0.35**	0.25	**0.38**	0.34

Chapter 6

Discussion

In this study, we proposed *BERT* base model and data augmentation strategy to relieve the overfitting and learn deep semantics of programming language to improve the performance of the deep learning-based model.

In our data augmentation experiment, we found that data augmentation can improve the performance of deep learning models for SDP, especially for small size dataset. Besides, we achieve better performance compared with other deep learning-based SDP model with *BERT* fine-tuning strategy. However, if we do not fine-tuning *BERT*, the result much worse than *Word2vec* based model. One of the most likely reasons that our mini-version *BERT* did not train the corpus dramatically, whereas the framework of *BERT* can represent the semantics of tokens stronger than *Word2vec* did. Base on this reason, when adopting fine-tuning mode, the performance is better than *Word2vec*. We finally experiment to investigate whether the full-token data type better than the AST-node data type. The result shows that, in the CPDP, the full-token data type can achieve better results than the AST-node data type. This result verifies our assumption that the full-token precessing method has a stronger ability to present semantics than the AST-node processing method in the CPDP. Generally, the longer sequence can ensure better performances. This reflects our assumption that because full-token data type preserves semantics and syntax of programming code, it should have better performance than AST-node data type.

Although we yield valuable workouts, someplace is no enough. For example, the process method source code of training set and corpus are not enough; we consider the order of token. However, an identical node may be a different node type; we did not add this information. Besides, our *BERT* model is not well-pretrained according to the parameters we set.

In our experiment, the first threat is the implementation of the relevant model. Especially the model that we do not know. The second problem is the BERT language model pretraining. Because of the limitation of our operating environment and time. We can not pretraining a large BERT model the same as the version released by official. On the other hand, In the training phase, many of the results are not stable.

Chapter 7

Conclusion

In this paper, we proposed *BERT* based deep learning for SDP, which outperforms other deep learning-based and traditional models. Our workouts show that compare with existing deep learning-based models; our model can more effectively extract the semantics of source codes. To relieve the shortage of data problems, we adopt a data augmentation strategy and the results that it can improve the performance well. We also compare two embedding model, *BERT* and *Word2vc* based model. The result show that *BERT* without fine-tuning does not outperform *Word2vec* embedding method. This shows that the fine-tuning of *BERT* is important for downstream tasks.

Based on the weakness and improvement of our results, our future research Have two direction.

- Further research of *BERT* application for SDP. *BERT* based technique in the NLP field plays an important role in the development of NLP research. However, whether such kind of technique suit to generate SDP and other research in the programming language is remain to To be proven. However, in our experiment, it shows that it can outperform other deep learning models. On the other hand, our pretrained *BERT* is not complete version of *BERT* released by *Google*, But there is no pretrained *BERT* model for source code. Thereby, training a real *BERT* model for source code is one of our future work.

- Effective source code generation model for SDP. The shortage of defect dataset limits the development of SDP. One feasible method is using deep learning to generate more data. On the one hand, the existing PROMISE dataset comes from old projects; some projects and frameworks are not used anymore. One the other hand, because of the development of software development tools. Generally, there is a little bug in a project. This leads to an imbalanced problem. One approach is applying such defect files to generate more datasets.

Acknowledgment

I would like to thank my thesis advisor, prof. Takashi Kobayashi. He is always accommodating and have a helping hand if I had a question about my research or writing. He shows support in everything I do and teaches me the right direction for my thesis writing.

I would also like to thank my classmate Chen Lang who gave me some help when I met problems implementing my idea. Besides, I also would like to thank all my classmates in the Kobayashi lab. Without their help, I could not complete my academic research.

Finally, I would like to thank my families; without their support, I could not study in Japan can have a good life.

References

[1] Fumio Akiyama. An example of software system debugging. In *IFIP Congress (1)*, volume 71, pages 353–359, 1971.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[3] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, and Yuan Tang. Xgboost: extreme gradient boosting. *R package version 0.4-2*, pages 1–4, 2015.

[4] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[5] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921*, 2018.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[7] Karim O Elish and Mahmoud O Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[8] Guisheng Fan, Xuyang Diao, Huiqun Yu, Kang Yang, and Liqiong Chen. Software defect prediction via attention-based recurrent neural network. *Scientific Programming*, 2019, 2019.

[9] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests. In *15th international symposium on software reliability engineering*, pages 417–428. IEEE, 2004.

[10] Maurice Howard Halstead et al. *Elements of software science*, volume 7. Elsevier, 1977.

[11] Rachel Harrison, Steve J Counsell, and Reuben V Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, 1998.

[12] Geoffrey E Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[14] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pages 1–10, 2010.

[15] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Issam H Laradji, Mohammad Alshayeb, and Lahouari Ghouti. Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, 58:388–402, 2015.

[18] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *Proceedings of 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE, 2017.

[19] Hongliang Liang, Yue Yu, Lin Jiang, and Zhuosi Xie. Seml: A semantic lstm model for software defect prediction. *IEEE Access*, 7:83812–83824, 2019.

[20] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[22] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[23] Ahmet Okutan and Olcay Taner Yıldız. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181, 2014.

[24] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[25] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*, 2018.

[26] Zhongbin Sun, Qinbao Song, and Xiaoyan Zhu. Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1806–1817, 2012.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[28] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th international conference on Software engineering (ICSE)*, pages 297–308, 2016.

[29] Tao Wang and Wei-hua Li. Naive bayes software defect prediction model. In *Proceedings of 2010 International Conference on Computational Intelligence and Software Engineering*, pages 1–4, 2010.

[30] Tiejian Wang, Zhiwu Zhang, Xiaoyuan Jing, and Liqiang Zhang. Multiple kernel ensemble learning for software defect prediction. *Automated Software Engineering*, 23(4):569–590, 2016.

[31] Jason W Wei and Kai Zou. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196*, 2019.

[32] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 1480–1489, 2016.

[33] Peng Zhou, Wei Shi, Jun Tian, Zhenyu Qi, Bingchen Li, Hongwei Hao, and Bo Xu. Attention-based bidirectional long short-term memory networks for relation classification. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 2: Short papers)*, pages 207–212, 2016.