A Study on Prediction of Source Code Changes with Natural Language Processing

Dissertation submitted in partial fulfillment for the
degree of Master of Engineering

Yuto Kaibe

Under the supervision of
Professor Hiroyuki Okamura

Dependable Systems Laboratory,
Informatics and Data Science Program,
Graduate School of Advanced Science and Engineering,
Hiroshima University, Higashi-Hiroshima, Japan

February 2023

Abstract

In this thesis we discuss the applicability of AI-based natural language processing (NLP) to the prediction of source code changes. The NLP is a promising research area in which the AI technology can be applied. In particular, BERT (bidirectional encoder representations from transformers) is one of the most successful NLP models, is based on the concept of embedding in which tokens are mapped into a vector space. For a variety of NLP tasks such as language translation, the embedding-based approach has offered the good performance compared to the conventional approach without embedding. In this thesis, we consider how to use the embedding-based model in the analysis of software repository. Specifically, as training data, we will prepare source code written in the Java language collected from Github and a mapping of the number of changes per token that make up the source code. It will be divided into training data and verification data. Then, the source code for the training data is trained to predict the number of changes per token in the source code using CuBERT, BERT, and the conventional model as input. Predictions are made on the validation data and a comparison of prediction accuracy shows that CuBERT outperforms the prediction accuracy of the other models. the task of predicting the number of source code changes, and conducts experiments to get answers for the following three questions: (i) the effect of pre-training of the embedding models, (ii) fine-turning vs. transfer learning in the embedding models, and (iii) how to extend the limit of input token lengths in the embedding models.

Acknowledgements

I would like to extend my sincere gratitude to Professor Hiroyuki Okamura, the supervisor of my study, for his constant guidance, kind advice and continuous encouragement throughout the progress of this work. I wish to thank Professor Tadashi Dohi, Professor Shaoying Liu and Professor Chuzo Iwamoto for their many invaluable comments, constant support, helpful guidance and warm encouragement. Finally, it is my special pleasure to acknowledge the hospitality and encouragement of the past and present members of the Dependable Systems Laboratory, Informatics and Data Science Program, Graduate School of Advanced Science and Engineering, Hiroshima University.

Contents

Chapter 1

Introduction

Recently, agile development has been much popular in software development. The agile development is the development style in which software is developed with small cycle consisting of building and testing phases. In the agile development, refactoring is one of the most important activities to develop the high quality software. The refactoring is a technique for restructuring and rewriting software codes without any change on their interfaces. Therefore, we observe the frequent changes of source codes on the agile development.

On the other hand, the frequency of source code changes is available as the metric to check the health condition of software development. As is well known, such changes frequently occur in the early phase of software development, because user requirements and associated software architectures are not stable in the early phase. Also, it can be seen that the frequency of source code changes gradually decreases as the software is matured. This phenomenon is similar to the software reliability growth, i.e., the phenomenon that the number of detected bugs decreases as the software testing is elapsed in the traditional water-fall-type software development. In the traditional software reliability engineering, researchers have discussed on the trends of the number of bugs detected in testing process with statistical models [1, 2], and have evaluated the quality of software products statistically. This implies that, even in the agile development, we obtain statistical information on the quality of software products from the empirical data of the number of source code changes.

Popular analyses on the number of bugs are the fault-prone module prediction [3] and the bug prediction [4]. The fault-prone module prediction is the

problem to determine a software module that is expected to involve software bugs, and the bug prediction is to estimate the number of software bugs involved in the software. The fault-prone module prediction and the bug prediction belong to the discriminant and regression problems in statistics, respectively, whose input variables are given by software metrics. The software metrics are quantitative values to summarize the characteristics of software such as the lines of code and complexity. However, in practice, it is difficult to determine the best software metrics for bug analyses. In addition, there is no good practice on what metrics should be measured to improve prediction accuracy.

Apart from programming languages, in the area of natural language processing (NLP), there have been drastic improvements with machine learning techniques. In particular, BERT (bidirectional encoder representations from transformers) is one of the most successful NLP models in a variety of tasks such as language translation. A key point of BERT is the technique of embedding in which tokens are embedded into the vector space numerically. This concept enables us to skip the procedure of extracting and choosing good metrics, and is expected to be useful to analyze programming languages. In fact, Kanade et al. [5] discussed the embedding-based NLP model, called CuBERT, which is BERT whose pre-training used program source codes. CuBERT outperformed the previous models in five classification tasks and one modification task for source codes. This implies that it is effective to use the data collected from the target domain in pre-training. However, there still remains a question this is always to be effective.

This thesis attempts to discuss the applicability of NLP models for the bug prediction task, i.e., BERT and CuBERT can be applied to predict the number of bugs from source codes directly. This thesis considers the following research questions:

1. **How effective is the embedding approach for the bug prediction task?**

   First of all, we reveal how effective are BERT and CuBERT for the prediction of the number of bugs compared to the conventional metrics-based regression approach. In addition, by comparing BERT and CuBERT in terms of prediction accuracy, we discuss how effect the pre-training data

is improved.

2. **Which is the best approach for the re-training; fine-turning or transfer learning?**
   Second, we investigate the difference between fine-turning and transfer learning. Both approaches are commonly used for the model to fit the specific task from the pre-trained model. However, the best practice to choose the appropriate approach is still unknown. Through the bug prediction task, we try to get an answer for this question.

3. **Which is the appropriate model; linear regression or Poison regression?**
   Traditionally, the bug prediction has two different models; linear regression and Poisson regression. Empirically, Poisson regression is known to be superior to linear regression in the task of bug prediction. However, we do not know whether it still holds in the embedding-based model.

4. **How to extend the limit of input tokens?**
   The weakness of BERT-like model is a limitation for the size of input tokens. That is, we have to consider some modifications to handle the long sentences. In particular, compared to the sentence with natural language, programs have long-range dependency that a token is affected from a far token. Therefore, it is more meaningful to discuss how to extend the limitation of input tokens.

Experimental results showed that the NLP model outperformed the Conventional Model. The results also showed that the NLP model outperformed the Conventional Model in the task of predicting the number of changes per token in the source code, with CuBERT as the model, Fine-Tuning as the training method, and Poison Regression as the Regression method. The results also show that CuBERT can perform the regression and classification tasks with the same accuracy as when similar experiments were performed on methods for both the regression and classification tasks.

The organization of this thesis is as follows: Chapter 2 describes BERT and CuBERT. In Chapter 3, we compare the accuracy with the NLP model on the task of predicting the number of changes per token in the source code and show

the superiority of the NLP model. In Chapter 4, we examine how CuBERT handles source code with a number of tokens that exceeds the input limit, and perform regression and classification tasks to see how well it works. Chapter 5 summarizes this thesis and discusses future issues.

Chapter 2

Related Work

2.1 BERT

Natural language processing (NLP) is one of the most developed research fields in recent years. In fact, a well-known NLP model, BERT (Bidirectional Encoder Representations from Transformers), uses deep learning techniques to extract important features from raw text data. In particular, an important feature in NLP models is what is called attention, which is the part of understanding the meaning of text.

Because BERT's architecture is based on Transformer[6], which is different from time series models such as recurrent neural networks (RNNs), BERT essentially generates a fixed-size vector sequence from a fixed-size input sentence BERT basically generates a fixed-size vector sequence from a fixed-size input sentence.

The training of BERT consists of the following two tasks:

- The masked language model (MLM): First some works on an input sentence are replaced by the special token [MASK]. BERT trains to predict what are the original words for the masked ones from the context of sentence. For instance, consider the following sentence:

  ```
  This morning he had breakfast.
  ```

  The training data is generated by replacing randomly selected words with [MASK], e.g.,

```
This morning he [MASK] breakfast.
```

BERT trains to guess the original words only from the words before and after the masked words. In MLM task, for a selected word, it is replaced by [MASK] with the probability 0.8 and is replaced by a another randomly selected word with the probability 0.1. Otherwise, it remains with the probability 0.1.

- The next-sentence prediction (NSP): Consider two sentences which are separated by the special token [SEP]. BERT predicts whether there is a semantic connection between the two sentences or not from the context of two sentences. For example, consider the following sentences:

  ```
  [CLS] I [MASK] to a bookstore.
  [SEP] There I bought three books.
  ```

  There is a semantic connection between the above two sentences. On the other hand, it is clear that there is not a semantic connection between the following two sentences:

  ```
  [CLS] I [MASK] to a bookstore.
  [SEP] People are mammals.
  ```

  BERT learns to predict the semantic connection between them as the discriminant problem. The training data are organized in which 50% are positive examples, i.e., two sentences have a semantic connection, and the remaining 50% are negative examples, i.e., there is not a semantic connection. Also, [CLS] is the special token to express the beginning of sentence.

Pre-trained with MLM and NSP tasks, BERT achieves text embedding. Text embedding is a technique that converts the meaning of each text into a vector representation, and by performing operations, it is possible to compute the semantic similarity between two sentences. In addition, these tasks can be fine-tuned for downstream supervised tasks and can produce high accuracy in many natural language processing tasks.

2.2 CuBERT

CuBERT is a natural language processing model based on BERT. As with BERT, the pre-training tasks are MLM and NSP as described above; while BERT learns a large natural language corpus, CuBERT learns a large source code corpus. Therefore, we can say that CuBERT achieves source code embedding. In this thesis, we use this pre-trained CuBERT. Within the CuBERT[5], CuBERT is trained on a large amount of source code to achieve high accuracy in source code embedding; to evaluate CuBERT's accuracy, five classification tasks and one program modification task are performed and the results are compared to other models. The comparison shows that CuBERT outperforms all models even with short training time and few labeled problems.CuBERT is a 24-layered model with 16 attention heads and 1024 hidden units.

Chapter 3

Experiments on Methods

Until now, the number of lines of code and complexity of source code have been mainly used as software metrics for bug prediction, such as predicting the number of changes to source code. However, it is difficult to determine the best software metrics for bug prediction. In addition, there is no good practice on what metrics should be measured to improve prediction accuracy. On the other hand, in recent years, there has been a lot of research on understanding and improving source code based on machine learning techniques developed in natural languages. Among them, BERT is a very popular NLP (natural language processing) model that has achieved good results in various natural language processing tasks; CuBERT has been shown to have excellent accuracy in various classification tasks in CuBERT[5]. Therefore, in this chapter, we perform the task of predicting the number of changes per token in source code using NLP models (BERT and CuBERT). We compare the results with those obtained using the number of lines of code and the number of if statements in the source code, which are used in conventional prediction, and show the superiority of CuBERT. In doing so, we will also compare several CuBERT training methods and loss functions to identify the optimal combination for this task.

3.1 The Task of Predicting The Number of Changes Per Token of Source Code

This section describes the task of predicting the number of changes per token of source code. The smallest unit of a string in a source code that cannot be

further divided is called a token. A method is modified for reasons such as modification, addition, or deletion of source code. The number of changes is divided by the number of tokens constituting the method to obtain the number of changes per token for the method. This prediction per token is performed in this task. Methods here include not only instance methods and class methods within a class, but also a series of processes. An example of a method is shown in 3.1. The details of the data are described in A.

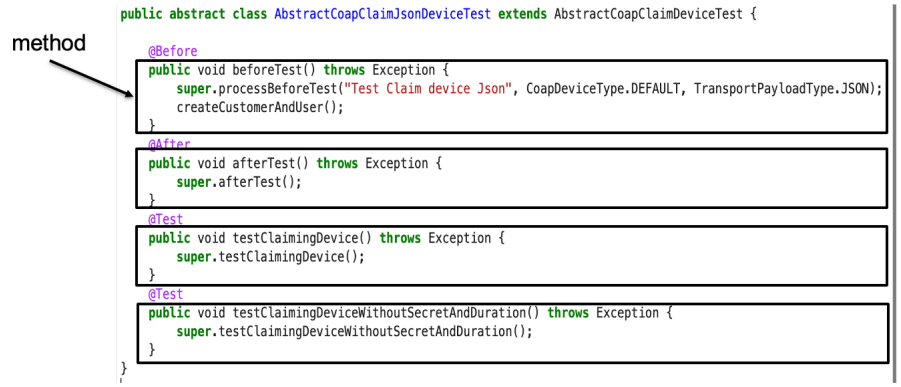


Figure 3.1: Example of method

3.2 Code Corpus for Source Code Change Count Prediction Task

The task of predicting the number of changes per token of source code uses Java files on Github, where classes must be defined and code is written within those classes. The 200 methods that make up each Java file are used in the source code change count prediction task. Of these, 150 are used as training data and 50 as validation data.

3.3 Source-Code Tokenize

The smallest unit of a string in a source code that cannot be further divided is called a token, and the process of dividing a source code into tokens is called tokenizing. The tokenization rules used in this study are based on the standard Java tokenizer.

Using the SubwordTextEncoder from the Tensor2Tensor project[7], the program vocabulary is compressed with the subword vocabulary [8]. Every word in the program vocabulary can be encoded using one or more subword tokens. Encoding refers to the conversion of each token into a vector representation. As mentioned above, we first tokenize the source code into tokens and then encode those tokens one by one into the subword vocabulary. The purpose of encoding in subword tokens is to account for the syntactically meaningful bounds of the token.

3.4 CuBERT for Predicting the Number of Source Code Changes

The model created to perform the task of predicting the number of changes per token in the source code by CuBERT is a model with 24 layers with 16 attention heads and 1024 hidden units, CuBERT, and an additional Linear layer in the output layer. The value passed to the Linear layer is the value of the [CLS] token.

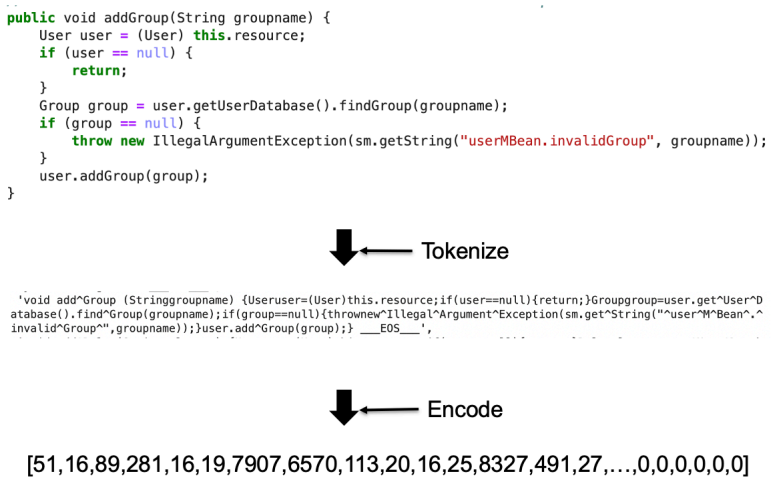


Figure 3.2: A sequence of tokenize and encode

3.5 Baselines

3.5.1 BERT

The structure of the model created for the task of predicting the number of changes per token in the source code by BERT is the same as the CuBERT model described above: BERT, a model with 24 layers with 16 attention heads and 768 hidden units, and a Linear layer added to the output layer. The value passed from BERT to the Linear layer is the value of the [CLS] token. In this thesis, we use BERT, which has been pre-trained with English documents.

3.5.2 Conventional model

The conventional model uses four explanatory variables to predict the number of changes per token in the source code. Specifically, they are the number of tokens comprising a method, the number of if statements, the number of loops, and 1. 1 is a parameter for adjustment. The structure of the model is one Linear layer.

3.6 Learning Method

In this experiment, CuBERT and BERT use pre-trained models. Therefore, we will compare and verify whether Fine-Tuning or Transfer Learning is more

suitable for the task of predicting the number of changes per token of source code.

3.6.1 Fine-Tuning

Fine tuning is a method of adding new layers to a pre-trained model and re-training the entire model. Because Fine-Tuning is a method of reusing a pre-trained model, it does not require training from scratch, thus reducing computation time. In the task of predicting the number of changes per token in the source code using Fine-Tuning as the learning method in this thesis, a linear layer is added to the pre-trained CuBERT and BERT, and the entire model is trained by Fine-Tuning.

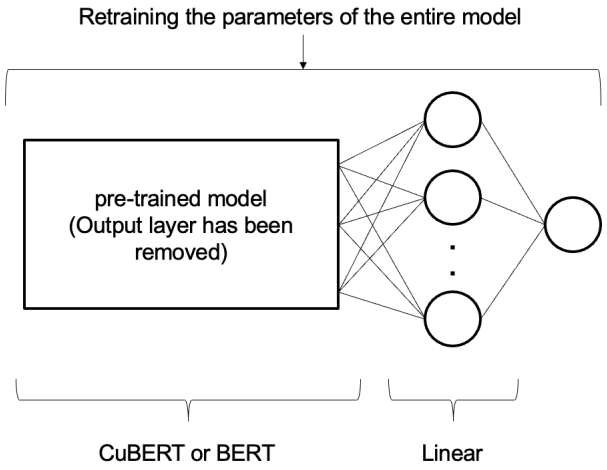


Figure 3.3: Schematic of Fine-Tuning

3.6.2 Transfer Learning

Transfer learning is a method in which the parameters of a previously trained model are not relearned, but only the parameters of the added layer are learned. By using a model that has already been trained with a large amount of high-quality data, Transfer Learning can create a highly accurate model with only a small amount of data. In the task of predicting the number of changes per token of source code using Transfer Learning as the learning method in this thesis, the Linear layer is added to the pre-trained CuBERT and BERT as in

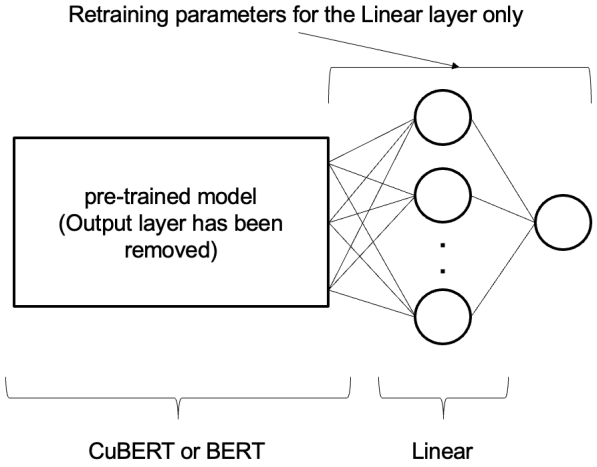


Figure 3.4: Schematic of transfer learning

the Fine-Tuning model above, and the parameters of CuBERT and BERT are The parameters of CuBERT and BERT are not updated during training, but only the parameters of the Linear layer are updated.

3.7 Loss Function

In this thesis, we compare two loss functions for the task of predicting the number of changes per token of source code, MSE (Mean Squared Error) and the negative value of the log-likelihood function of Poison regression (hereinafter called Poison Loss), and evaluate which is more suitable for this task.

3.7.1 MSE(Mean Squared Error)

MSE (Mean Squared Error) is a basic loss function that is often used to make regression predictions using machine learning models. MSE is a function that calculates the square value of the "difference between the predicted value and the correct value (=error)" for each data, and outputs the value (=average value) obtained by dividing the sum by the number of data, which is expressed by the following formula

$$MSE = \frac{1}{m}\sum_{i=0}^{m-1}(y_i - \hat{y_i})^2 \tag{3.1}$$

where y_i is the number of actual source code changes and $\hat{y_i}$ is the model's prediction. The smaller this value, the smaller the model is, the smaller the error. The MSE was used as the evaluation function for the experiment.

3.7.2 Poison Loss

Here we introduce the bug prediction model with software metrics based on the Poisson regression [9]. They assumed that the probability mass function of the number of bugs in the software follows

$$P(Y = y) = \frac{\xi(\boldsymbol{x})^y}{y!}\exp(-\xi(\boldsymbol{x})), \tag{3.2}$$

where $\boldsymbol{x}$ is a vector of software metrics and $\xi(\cdot)$ is called a link function. Generally, in the context of Poisson regression, the link function is given by an exponential function, namely,

$$\log \xi(\boldsymbol{x}) = \beta_0 + \boldsymbol{\beta}\boldsymbol{x}, \tag{3.3}$$

where $(\beta_0, \boldsymbol{\beta})$ are regression coefficients that are the parameters to be estimated from data. Let $(y_i, \boldsymbol{x}_i)$ be observed data, i.e., a tuple of the number of bugs and the software metrics of corresponding software. Based on the maximum

likelihood (ML) estimation, the coefficients can be determined by maximizing the log-likelihood function:

$$\mathcal{L}(\beta_0, \boldsymbol{\beta}) = \sum_{i=1}^{m} \left(y_i \log \xi(\boldsymbol{x}_i) - y_i! - \xi(\boldsymbol{x}_i) \right). \tag{3.4}$$

Alternatively, we find the parameters minimizing the following loss function:

$$L(\beta_0, \boldsymbol{\beta}) = -\sum_{i=1}^{m} \left(y_i \log \xi(\boldsymbol{x}_i) - \xi(\boldsymbol{x}_i) \right). \tag{3.5}$$

3.8 Trainig Details

This thesis examines which of three items, machine learning model (CuBERT, BERT, Conventional model), its learning method (Fine-Tuning, Transfer Leaning), and loss function (MSE, Poison Loss), respectively, are suitable for the task of predicting the number of changes per token of source code.

This section describes the task of predicting the number of changes per token in the source code with CuBERT as the model and Fine-Tuning as the learning method.The model consists of CuBERT and two Linear layers.CuBERT is used as the feature extractor to extract the value of the [CLS] token.The number of hidden units in the two Linear layers is (1024,50).When the loss function is Poison Loss, the output value is calculated by exp(exponential) and the value is returned.The output dropout probability between Linear layers is 0.1 and the learning rate is 2×10^{-5}.AdamW was used as the optimizer throughout. The batch size was 2 and the number of epochs was 10 and 20. This configuration is the same whether the loss function is MSE or Poison Loss.

Describe the task of predicting the number of changes per token in the source code with CuBERT as the model and Transfer Learning as the learning method.The model consists of CuBERT and two Linear layers.CuBERT is used as the feature extractor to extract the value of the [CLS] token.The number of hidden units in the two Linear layers is (1024,5). When the loss function is Poison Loss, the output value is calculated by exp(exponential) and the value is returned.The output dropout probability between Linear layers is 0.1 and the learning rate is 2×10^{-5}. AdamW was used as the end-to-end optimizer. The batch size was set to 2 and the number of epochs was set to 10 and 20. This configuration is the same whether the loss function is MSE or Poison Loss.

This section describes the task of predicting the number of changes per token in the source code with BERT as the model and Fine-Tuning as the learning method. The model consists of BERT and two Linear layers. BERT is the default configuration of the BERT Large model, which has 512 tokens that can be input. CuBERT is a 24-layer model with 16 attention heads and 768 hidden units, and and returns the value of the [CLS] token as output. It has 768 dimensions. Therefore, the number of hidden units in the two Linear layers is (768,50). When the loss function is Poison Loss, the output value is computed exp(exponential) and the value is returned. the output dropout probability between Linear layers is 0.1 and the learning rate is 2×10^{-5}. AdamW is used as the optimizer throughout. The batch size was set to 2 and the number of epochs was set to 10 and 20. This configuration is the same whether the loss function is MSE or Poison Loss.

This section describes the task of predicting the number of changes per token in the source code with BERT as the model and Transfer Learning as the learning method. The model consists of BERT and two Linear layers. CuBERT is used as the feature extractor to extract the [CLS] vector, and the number of hidden units in the two Linear layers is (768,5). When the loss function is Poison Loss, the output value is computed exp(exponential) and its value is returned. the output dropout probability between Linear layers is 0.1 and the learning rate is 2×10^{-5}. AdamW is used as an optimizer throughout. The batch size was set to 2 and the number of epochs was set to 10 and 20. This configuration is the same whether the loss function is MSE or Poison Loss.

This section describes the task of predicting the number of changes per token in the source code using the Conventional model. The model consists of one Linear layer. Regression is performed with the number of tokens in the source code, the number of if statements, the number of loops, and the adjustment parameter 1 as explanatory variables. When the loss function is Poison Loss, the output value is calculated by exp(exponential) and the value is returned. The learning rate is set to 2×10^{-5}. AdamW is used as the optimizer throughout. The batch size was set to 2 and the number of epochs was set to 10 and 20. This configuration is the same whether the loss function is MSE or Poison Loss.

3.9 Experimental Results

The following experimental results show that for the task of predicting the number of changes per token of source code, the best accuracy is achieved when the model is CuBERT, the learning method is Fine-Tuning, and the loss function is Poison Loss.

3.9.1 NLP vs. Conventional Model

The purpose of this analysis is to determine if NLP is useful in the task of predicting the number of changes per token of source code. The results are shown in Table 4.2.

The results show that CuBERT outperforms BERT's MSE score of 0.874 and the Conventional model's MSE score of 0.877 in this task. Thus, experiments show that CuBERT is more effective than BERT and the Conventional model in the task of predicting the number of changes per token of source code.

When focusing on the number of epochs in each model, we see that even 10 epochs of forecasting using CuBERT is as good as or better than 20 epochs of the other models.

3.9.2 Fine-Tuning vs. Transfer Learning

Here we discuss the learning method from table 4.2. In the prediction results using CuBERT in this task, the best score was 0.044 as a result of learning by Fine-Tuning, and the best score was 0.821 as a result of learning by Transfer Learning. Therefore, we can say that Fine-Tuning is a better learning method than Transfer Learning for prediction using CuBERT in this task. On the other hand, the prediction results using BERT showed a best score of 2.063 as a result of learning by Fine-Tuning and a best score of 0.749 as a result of learning by Transfer Learning. Therefore, we can say that Transfer Learning is a better learning method than Fine-Tuning for prediction using BERT in this task.

3.9.3 MSE vs. Poison Loss

In this section, the loss functions are discussed. Figure 3.5 - Figure3.9 shows the train loss per epoch for each model. The horizontal axis shows the number of epochs and the vertical axis shows the value of loss for the train data; the value

Table 3.1: Results of each model's prediction of the number of changes per token of source code.

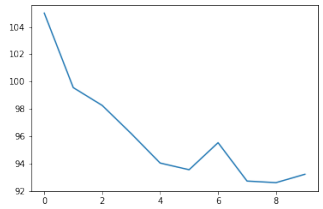
Model	Setting			MSE
CuBERT	Fine-Tuning	MSE	10 epochs	2.833
			20 epochs	0.617
		Poison	10 epochs	0.611
			20 epochs	**0.044**
	Transfer Learing	MSE	10 epochs	0.907
			20 epochs	0.888
		Poison	10 epochs	0.866
			20 epochs	0.821
BERT	Fine-Tuning	MSE	10 epochs	2.063
			20 epochs	2.175
		Poison	10 epochs	2.428
			20 epochs	2.247
	Transfer Learing	MSE	10 epochs	0.918
			20 epochs	**0.749**
		Poison	10 epochs	1.643
			20 epochs	0.873
Conventional Model	MSE		10 epochs	0.958
			20 epocs	1.159
	Poison		10 epochs	**0.921**
			20 epocs	1.124

of loss is the value of MSE when MSE is used as the loss function and the value of Poison when Poison Loss is used as the loss function. Figure3.5 shows the train loss when CuBERT is trained by Fine-Tuning in the prediction of this task, and Figure3.6 shows the train loss when it is trained by Transfer Learning. Figure3.5 shows the train loss when BERT is trained by Fine-Tuning, and Figure3.6 shows the train loss when it is trained by Transfer Learning. Figure3.9 shows the train loss of the Conventional model.

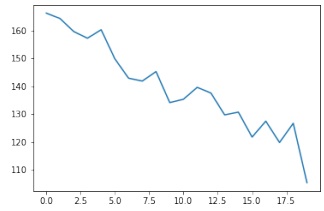
In this task, the accuracy of prediction using CuBERT is better when Poison Loss is used as the loss function than when MSE is used as the loss function, regardless of whether Fine-Tuning or transfer learning is used, as Table 4.2 shows that the accuracy is better when Poison Loss is used as the loss function than when MSE is used as the loss function. In addition, the train loss progress in Fig.3.5 shows that when the learning method is Fine-Tuning, the convergence of loss is stable as the number of epochs increases when Poison Loss is used as the loss function. On the other hand, when the loss function is MSE, the convergence of loss is not stable. When the learning method is Transfer Learing, the loss convergence is not stable for both MSE and Poison loss functions, as shown in 3.6.

In the prediction using BERT, it can be seen from the table 4.2 that the accuracy is better when MSE is used as the loss function than when Poison Loss is used as the loss function, regardless of whether Fine-Tuning or Transfer Learning is used as the learning method. In addition, when Fine-Tuning is used as the learning method, the loss does not converge stably regardless of whether MSE or Poison Loss is used as the loss function, as shown in Fig.3.7. On the other hand, when transfer learning is used as the learning method, the loss value decreases as the number of epochs increases for both MSE and Poison Loss.

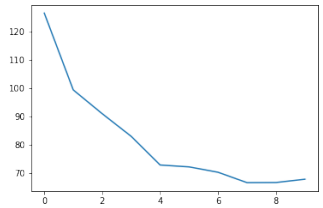
Table 4.2 shows that the accuracy is better when the Poison Loss is used as the loss function than when the MSE is used as the loss function in the prediction using the Conventional model. It can also be seen from Figure3.7 that the loss is stable and converges regardless of whether MSE or Poison is used as the loss function.



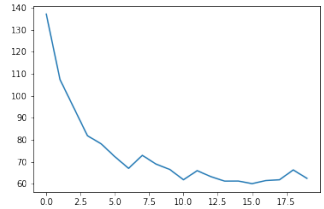
(a) Train Loss with MSE and 10epochs.



(b) Train Loss with MSE and 20epochs.

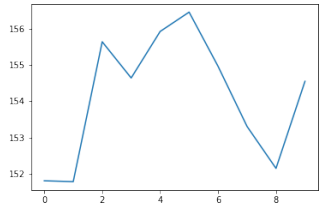


(c) Train Loss with Poison and 10epochs.

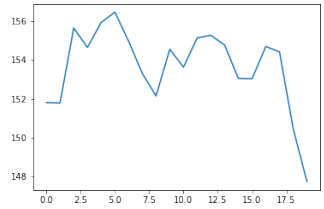


(d) Train Loss with Poison 20epochs.

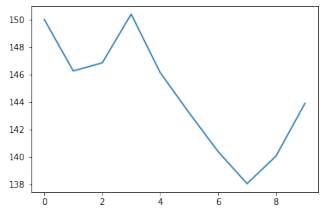
Figure 3.5: Train loss when CuBERT is trained by Fine-Tuning



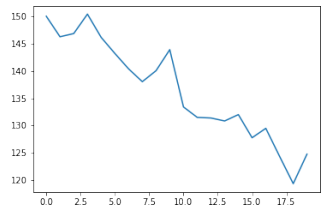
(a) Train Loss with MSE and 10epochs.



(b) Train loss with MSE and 20epochs.

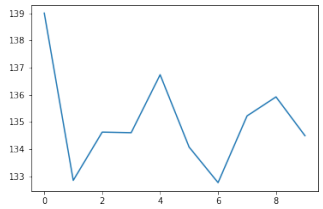


(c) Train Loss with Poison and 10epochs.

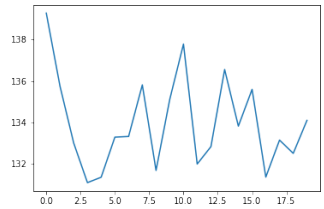


(d) Train Loss with Poison and 20epochs.

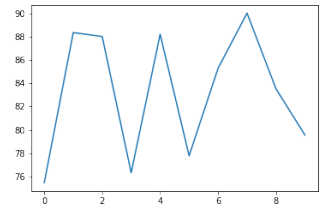
Figure 3.6: Train loss when CuBERT is trained by Transfer Learning



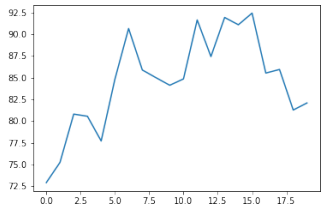
(a) Train Loss with MSE and 10epochs.



(b) Train Loss with MSE and 20epochs.

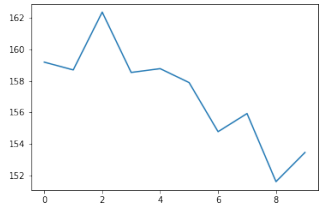


(c) Train Loss with Poison and 10epochs.

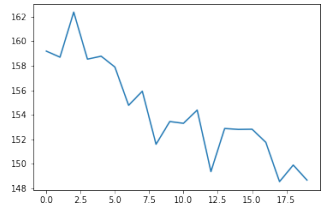


(d) Train Loss with Poison and 20epochs.

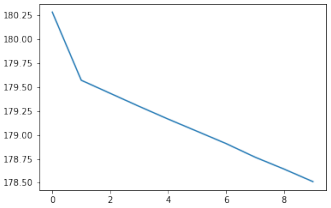
Figure 3.7: Train loss when BERT is trained by Fine-Tuning



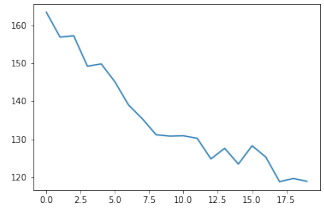
(a) Train Loss with MSE and 10epochs.



(b) Train Loss with MSE and 20epochs.

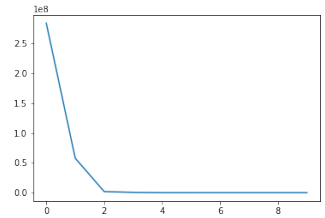


(c) Train Loss with Poison and 10epochs.

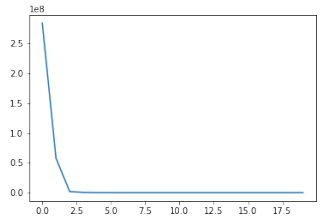


(d) Train Loss with Poison and 20epochs.

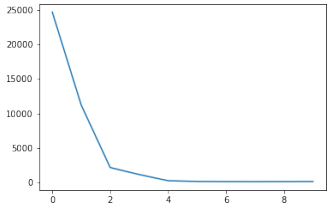
Figure 3.8: Train loss when BERT is trained by Transfer Learning



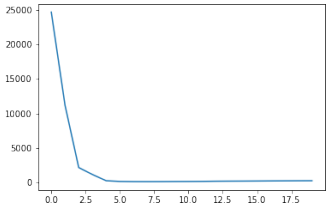
(a) Train Loss with MSE and 10epochs.



(b) Train Loss with MSE and 20epochs.



(c) Train Loss with Poison and 10epochs.



(d) Train Loss with Poison and 20epochs.

Figure 3.9: Train loss which Conventional model

Chapter 4

Experiments on Class

In the thesis 3, the task of predicting the number of changes per token in the source code that makes up the method was performed for various settings, and it was found that the best model for this task was CuBERT, the training method was fine-tuning, and the loss function was Poison Loss. The thesis CuBERT[5] also performs various classification tasks on methods and shows that CuBERT has superior accuracy compared to other models. The reason for using methods as the target in these experiments is that CuBERT has a limit on the number of tokens that can be entered. At the time of this writing, CuBERT is available as open source software with input limits of 512, 1024, and 2048 tokens. No attempts have yet been made to release source code with a number of tokens that exceeds these input limits. Therefore, this chapter discusses how CuBERT handles tokens that exceed the input limit. Specifically, we will apply methods for handling sentences that exceed the token count limit in the field of natural language processing, and verify through experiments whether these methods are also effective for CuBERT, which handles programming languages.In general, when preparing source code for CuBERT experiments, source code data is often collected from Github. In addition, Github repositories often manage software in units of files or classes. These files or classes generally consist of a number of tokens that exceeds the input limit. Therefore, it is useful for CuBERT to be able to handle source code with a large number of tokens.

4.1 Consideration of how to handle classes using CuBERT

CuBERT has a limit on the number of tokens that can be entered. However, there is no standard approach for handling data with more tokens than the limit. Therefore, we apply a method for handling sentences that exceed the token limit in the field of natural language processing, and experimentally verify whether the method is also effective for CuBERT for programming languages. Two methods are commonly used in the field of natural language processing to handle sentences that exceed the token count limit: The first method uses only a portion of the sentence for the task. The second method is to split sentences. Since this chapter performs tasks on source code, rather than natural language sentences, these methods are applied to source code. First, consider which method is more appropriate for this task.

4.1.1 Use part of the source code

A portion of the tokens that make up the source code should be input to CuBERT so that they fall within the limit. As an example, this section describes a method in which the number of tokens that can be input to CuBERT is counted from the beginning of the source code and the number of tokens up to the limit is used as input.

- How to use the tokens from the beginning to the limit as input:

  Of the source code written with the number of tokens exceeding the limit, the tokens from the beginning of the source code to the limit are considered as input. An example is given in the following source code, where the input limit for CuBERT is 512 tokens. If the end of the first 10 lines (or ; in the following code) is the 512th token, then the first 10 lines are input and the remaining 11 lines and beyond are not input.

Program 4.1: sample1.java

```
1 public abstract class AbstractCoapClaimJsonDeviceTest extends
      AbstractCoapClaimDeviceTest {
2
3     @Before
4     public void beforeTest() throws Exception{
```

```
5         super.processBeforeTest("Test Claim device Json",
              CoapDeviceType.DEFAULT, TransportPayloadType.
              JSON);
6         createCustomerAndUser();
7     }
8     @After
9     public void afterTest() throws Exception{
10        super.afterTest();
11    }
12    @Test
13    public void testClaimingDevice() throws Exception{
14        super.testClaimingDevice();
15    }
16    @Test
17    public void testClaimingDeviceSecretAnd() throws
          Exception {
18        super.testClaimingDeviceSecretAnd();
19    }
20 }
```

This is the simplest method, but since only tokens from the beginning to the limit can be entered into CuBERT, the source code beyond the limit is not considered.

4.1.2 How to use the entire source code

This section introduces the method of splitting source code.

- Use each method as an input

The methods in the source code basically consist of a small number of tokens. Therefore, the number of tokens constituting a method is within the input limit of CuBERT. Therefore, the class is divided by the methods that make up the class, and these are used as input to CuBERT.

For example, in the following source code, lines 4 to 7, 9 to 11, 13 to 15, and 17 to 19 are methods, respectively. Enter each of these methods in CuBERT.

Program 4.2: sample1.java

```
1 public abstract class AbstractCoapClaimJsonDeviceTest extends
      AbstractCoapClaimDeviceTest {
2
3     @Before
4     public void beforeTest() throws Exception{
5         super.processBeforeTest("Test Claim device Json",
              CoapDeviceType.DEFAULT, TransportPayloadType.
              JSON);
6         createCustomerAndUser();
```

```
7       }
8       @After
9       public void afterTest() throws Exception{
10          super.afterTest();
11      }
12      @Test
13      public void testClaimingDevice() throws Exception{
14          super.testClaimingDevice();
15      }
16      @Test
17      public void testClaimingDeviceSecretAnd() throws
            Exception {
18          super.testClaimingDeviceSecretAnd();
19      }
20  }
```

This method can consider the entire source code. However, CuBERT takes a vector of Encoded tokens as input and outputs a vector of the result of that calculation. Since these vectors correspond to each other on a one-to-one basis, inputting CuBERT for each method will output a vector for the number of methods. Therefore, it is necessary to consider how to handle the resulting output.

In 4.1.1, we introduced a method to input and handle a part of the source code into CuBERT. The method in 4.1.1 does not take into account the part of the source code beyond the limit. Therefore, this method may not be suitable for tasks that require information on tokens in the entire source code. On the other hand, the method 4.1.2 can be used for many tasks because it can use tokens of the entire source code. Therefore, this task uses the method of 4.1.2 to handle source codes with the number of tokens exceeding the input limit in CuBERT. Classes are used as source code with the number of tokens exceeding the input limit. Apply this method to classes and use CuBERT to perform regression and classification tasks. Perform regression and classification tasks using CuBERT on the methods as a baseline. Compare the results of each task for classes and each task for methods, and confirm the effectiveness of the method we have applied here, 4.1.2, as a method for handling source code with a number of tokens that exceeds the input limit.

4.2 Regression Task

In order to check the validity of the method of 4.1.2 to treat classes with CuBERT, we perform the task of predicting the number of changes per token in the source code of 3 chapters. Changes are made to a class because of source code modifications, additions, deletions, etc. This number of changes is divided by the number of tokens constituting the class to obtain the number of changes per token for the class. This prediction per token is performed in this task.

4.3 Classification Task

To check the effectiveness of the 4.1.2 method of treating classes with CuBERT, a Swapped Operand task is performed as a classification prediction task. The Swapped Operand task is a task that recognizes the swapping of operands before and after a noncommutative binary operator. For example, consider the following source code as a positive example.

Program 4.3: sample2.java

```
1 public abstract class sample2 {
2     public static void main(String[] args){
3         double a=0.0;
4         double b=2.0;
5         System.out.println(a/b);
6     }
7 }
```

Operands refer to a and b in the above source code. As a negative example, the following is a source code in which the operands a and b are replaced with / as a border.

Program 4.4: sample2Swap.java

```
1 public abstract class sample2Swap {
2     public static void main(String[] args){
3         double a=0.0;
4         double b=2.0;
5         System.out.println(b/a);
6     }
7 }
```

As shown in the example above, the Swapped Operand task prepares positive and negative examples for a single source code and identifies them.

4.4 Code Corpus for A Method To Handle Per-Class Source Code that Exceeds The Limit of The Number of Tokens Using CuBERT

4.4.1 Code Corpus for Regression Task

The Regression Task uses Java files on Github as in Chapter 3.1. 74 Java classes are used in this task. Of these, 52 are used as training data and 22 as validation data. An example class is shown in 4.1. In addition, the source code is tokenized using the same procedure as in chapter 3.2.

4.4.2 Code Corpus for Classification Task

The Classification Task uses Java files on Github as in Chapter 3.1. 130 Java classes are used in this task. Of these, 97 are used as training data and 33 as validation data. An example class is shown in 4.1. In addition, the source code is tokenized using the same procedure as in chapter 3.2.

Figure 4.1: Example of class

4.5 Model for Task

4.5.1 Model for Regression Task

This section describes the model created to perform the regression forecasting task for the class. The created model consists of CuBERT and two Linear layers. The following is a description of the series of processes performed by the model created in this task. First, the class is divided into methods. Tokenize and encode each method using the same rules as in3.3. CuBERT outputs a vector of the number of methods, which is the result of the calculation. This is pooled by max pooling and then the result is passed to the Linear layer. The number of hidden units in the Linear layer is (1024,50). AdamW was used as the optimizer from beginning to end. The batch size was set to 2 and the number of epochs was set to 10. An image diagram of the series of processes is shown in Figure4.2.

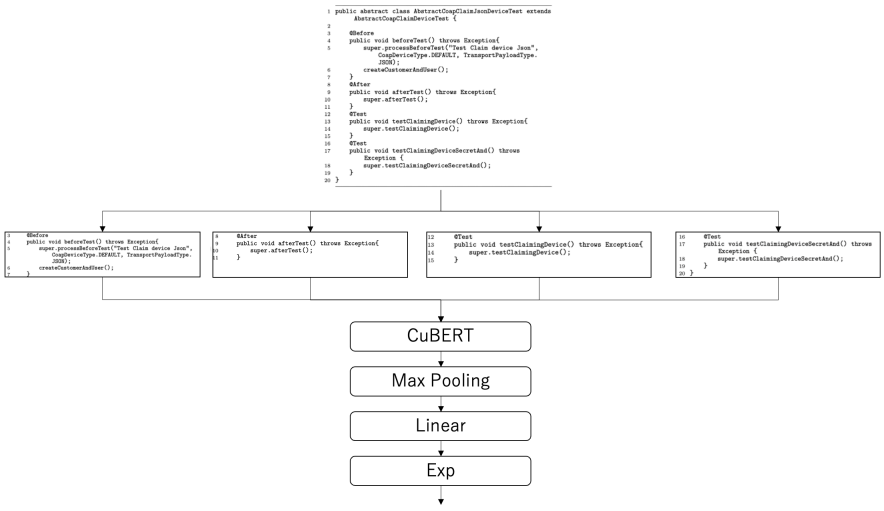


Figure 4.2: Diagram of a series of processes performed by the model in the regression task

4.5.2 Model for Classification Task

This section describes the model created to perform the classification task for the class. The created model consists of CuBERT, one linear layer and one log softmax layer. The series of processes performed by the model in this task are the same as in the case of 4.5.1. The number of hidden units in the linear layer is 1024, the output dropout probability between the linear layers is 0.1, and the learning rate is $2^{t}imes10^{-5}$. AdamW was used as the optimizer throughout. The batch size was set to 2 and the number of epochs was set to 30. An image diagram of the series of processes is shown in Figure4.3.

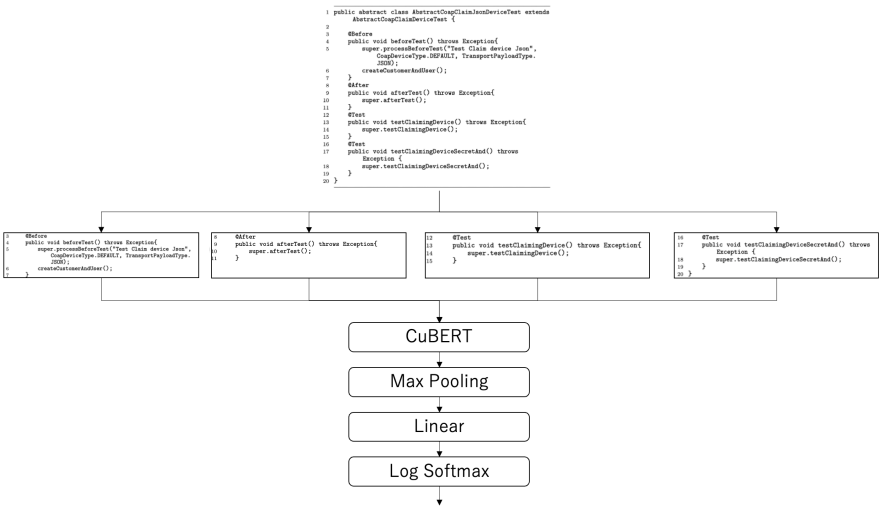


Figure 4.3: Diagram of a series of processes performed by the model in the classification and task

4.6 Baselines

4.6.1 Regression Task

In Chapter 3, the best accuracy in the task of predicting the number of changes per token of the source code was obtained when the model was CuBERT, the learning method was fine-tuning, and the loss function was Poison Loss, with an MSE score of 0.044. The results of the task of predicting the number of changes per token in the source code for this score and class using the Conventional model were used as the baseline. To perform this task with CuBERT, we use fine-tuning and Poison Loss as the learning method and the loss function, as these were the most accurate in the experiments in chapter 3.

4.6.2 Classification Task

Perform a Swapped Operand task on a method. The results will be compared with the results of the Swapped Operand task on the class. To perform this task with CuBERT, the learning method is fine-tuning. The results of the Swapped Operand task for methods using CuBERT showed an accuracy of 87.4371%. This score is used as the baseline when the Swapped Operand task is performed on classes. In this experiment with methods, 794 Java files were used

as experimental data, of which 595 were training data and 199 were validation data. Also, the batch size is set to 4. Note that the number of experimental data and batch size are different compared to the Swapped Operand task for the class.

4.7 Experimental Results

The purpose of this experiment is to clarify whether CuBERT can handle source code with a number of tokens that exceeds the input limit, such as classes, using the 4.1.2 method. The following are the results of experiments on regression and classification tasks that were conducted to confirm the effectiveness of the 4.1.2 method.

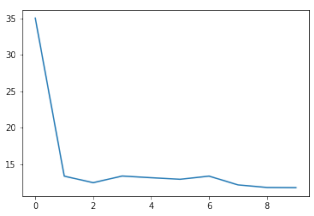
4.7.1 Regression Task

The MSE score was 0.054 after performing the task of predicting the number of changes per token in the source code for the class using the method of 4.1.2. The MSE score was 0.117 when this task was performed using the Conventional model. The result of this task with the method of 4.1.2 is 0.01 lower than the MSE score when the task of predicting the number of changes per token of the source code was performed for the method, and is 0.063 higher than the MSE score when this task was performed with the Conventional model. It is 0.063 higher than the MSE score when this task was performed with the Conventional model. Thus, although the results of the 4.1.2 method are inferior to the results of the task of predicting the number of changes per token of the source code for the method, the experiments show that it is more effective than the conventional model, which uses the conventional prediction method.

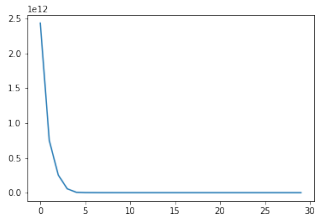
Table 4.1: MSE score on the change prediction task per token of source code for Classes and Methods.

	Setting	**MSE**
Method	CuBERT Fine-Tuning Poison 20 epochs	0.044
Class	The Way of 4.1.2	0.054
	Conventional model	0.117

Looking at the progress of train loss in Fig. 4.4, we can see that loss decreases with each increase in the number of epochs in both cases when this task is performed for a class and when forecasting is performed using the method described in 4.2.1 and the conventional model. Also, in both cases, the convergence of loss begins to stabilize at approximately the same timing, and from the third epoch onward, loss is stable and no longer decreases.



(a) Train Loss of The Way of 4.1.2



(b) Train Loss of Conventional model

Figure 4.4: Graph of Train Loss in the task of predicting the number of changes per token of source code for the class

4.7.2 Classification Task

As a result of performing the Swapped Operand task on the class using the method of 4.1.2, the Accuracy was 87.879%. In addition, as a result of performing this task with the method as the target, Accuracy was 87.437%. The result of this task using the method 4.1.2 is 0.342 higher than the Accuracy obtained by performing the task of predicting the number of changes per token in the source code for the methods. Note, however, that the number of epochs was 10 when this task was performed for classes, whereas the number of epochs was 1 when this task was performed for methods. Although it is necessary to increase the number of training cycles to achieve the same level of accuracy as Accuracy for the method-based task, the application of the 4.1.2 method to the class-based task was found to be very effective in this task.

Table 4.2: Accuracy of the Swapped Operand task.

	Setting	**Accuracy**
Method	CuBERT Fine-Tuning 1 epochs	87.437%
Class	The Way of 4.1.2 10 epochs	87.879%

The train loss in Figure 4.2.1 shows that the loss decreases with the number of epochs when this task is performed for the class. After the 16th epoch, the loss is stable and does not decrease.

Below is a mixed matrix of the classification results in this task. In this task, the Swapped Operand label is assigned to source code that includes operand swapping, and the None label is assigned to source code that does not include operand swapping. The mixed matrix shows that the task is able to classify both methods and classes fairly well. Table4.3 shows the mixture matrix of the classification results for this task for methods, and Table4.4 shows the mixture matrix of the classification results for this task for classes.

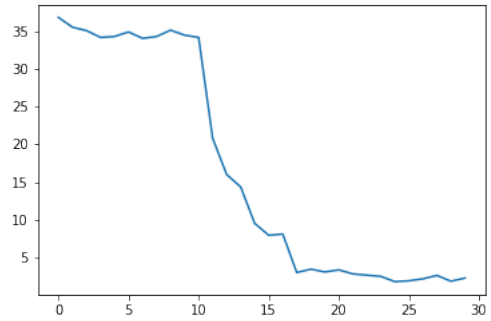


Figure 4.5: Graph of Train Loss in this task for class

Table 4.3: Mixed matrix of classification results in the Swapped Operand task for methods. Swapped Operand in table4.3 means the label of the source code that includes operand swapping, and None means the label of the source code that does not include operand swapping.

	Swapped Operand	None
Swapped Operand	75	25
None	0	99

Table 4.4: Mixed matrix of classification results in the Swapped Operand task for classes.

	Swapped Operand	None
Swapped Operand	17	3
None	1	12

Chapter 5

Conclusion

In Chapter 1 of this thesis, we explained the importance of predicting the number of changes to source code and introduced the fact that until now it has not been clear which software metrics should be used in such predictions. Therefore, this thesis briefly introduces the recent use of CuBERT, a model based on natural language processing models, for prediction, and how its accuracy outperforms that of conventional models.

In Chapter 2, we briefly introduced the architecture and learning tasks for BERT and CuBERT, the base of CuBERT.

In Chapter 3, we performed the task of predicting the number of changes per token in the source code using CuBERT and compared its accuracy with that of other models. In doing so, we also examined the learning method and loss function to see which combination of model, learning method, and loss function would be optimal for this task. From the experimental results, we confirmed that the most suitable combination of CuBERT as the model, fine-tuning as the learning method, and Poison Loss as the loss function for this task in terms of accuracy and learning stability.

In Chapter 4, we explained that it is not clear how CuBERT handles source code with a number of tokens that exceeds the input limit, and examined the possibility of applying a method commonly used in natural language processing models as a solution to this problem. We performed regression and classification tasks using this method, and found that the accuracy of both tasks exceeded the predictions of the baseline conventional method, or were nearly as accurate as the experimental results of this task for the method.

Future work will examine whether predictions can be made with high accuracy for time-series data as well. As mentioned in chapter 1, changes in the source code will transition over time. In the early stages of development, many changes are made, but in the later stages, the frequency of changes decreases as the software matures. In this thesis, the source code change frequency prediction is based on the number of times changes are made to the source code, and this behavior is not taken into account in the learning process. Therefore, it is our future task to create a model that takes this behavior into account by using time-series data for training.

Appendix A

Task Datasets

In this chapter, we describe in detail how we generated the experimental datasets used in the Chapter3 and Chapter4 tasks.

A.1 Task Datasets of Chapter3

This section describes the method used to generate the data used in the experiments in Chapter 3. The task required to generate the data for the experiments in chapter 3 is to split the Java class into separate Java classes for each method. This task is done using ANTLER, ANTLR (ANother Tool for Language Recognition)[1] is a parser generator based on LL syntax rules. According to the grammar rules, it automatically generates a lexer that performs lexical analysis, a parser that performs syntactic analysis, a tree parser that is a traverser of abstract syntax trees (ASTs), and an event listener that searches ASTs and processes when nodes are switched. In this task, Java classes are divided using ANTLER, and those that exclude access modifiers are treated as methods. An example of a method is shown below in FigureA.1.

```
'void ensureCapacity (longmaximumSize) {super.ensureCapacity(maximumSize);
if(period<=0){period=Integer.MAX_VALUE;}eventsToCount=period;} '
```

Figure A.1: Example of method

For the experimental data for the task of predicting the number of changes per token of source code, only methods with a change count of 10 or less were

[1] https://www.antlr.org/

used in the experiments.

A.2 Task Datasets of Chapter4

In the regression task, Java classes are used as experimental data. Note that the access modifiers are excluded as in A.1. The class has multiple methods. In the regression task, classes with less than 10 methods out of the classes were used as experimental data in order to reduce the computation process. Also, in this task, only classes with a source code modification count of 250 or less were used in the experiments.

The classification task also uses Java classes as experimental data, excluding access modifiers, and the Swapped Operand task determines whether the operands before and after a noncommutative binary operator are swapped. In this task, noncommutative binary operators are defined as ”>”,”<”,”>=”,”<=”,”.” and ”%”. In order to reduce the computational complexity of the classification task for classes, classes with 4 or fewer methods were used as experimental data.

Bibliography

[1] M. Xie. Software Reliability Modelling. World Scientific, Singapore, 1991.

[2] M. R. Lyu (ed). Handbook of Software Reliability Engineering. McGraw-Hill, New York, 1996.

[3] O. Mizuno and T. Kikuno. Prediction of fault-prone software modules using a generic text discriminator. *IEICE Transactions on Information and Systems*, E91.D(4):888–896, 2008.

[4] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, 1992.

[5] A. Kanade, P. Maniatis, G. Balakrishnan and K. Shi. Learning and evaluating contextual embedding of source code. *ICML*, 2020.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin. Attention is all you need. *arXiv*:1706.03762, 2017.

[7] Vaswani, A., Bengio, S., Brevdo, E., Chollet, F., Gomez, A. N., Gouws, S., Jones, L., Kaiser, L., Kalchbrenner, N., Parmar, N., Sepassi, R., Shazeer, N., and Uszko- reit, J. Tensor2tensor for neural machine translation. In Proceedings of the 13th Conference of the Associa- tion for Machine Translation in the Americas, AMTA 2018, Boston, MA, USA, March 17-21, 2018 - Volume 1: Research Papers, pp. 193–199, 2018. URL https://www.aclweb.org/anthology/W18-1819/.

[8] Schuster, M. and Nakajima, K. Japanese and korean voice search. In International Conference on Acoustics, Speech and Signal Processing, pp. 5149–5152, 2012.

[9] T. M. Khoshgoftaar, K. Gao, and R. M. Szabo. An application of zero-inflated Poison Regression for software fault prediction.

[10] Ilya Loshchilov, Frank Hutter. Decoupled Weight Decay Regularization. *arXiv*:1711.05101, 2017.

Publication List of the Author

[1] 海部由登，岡村寛之，土肥正，”自然言語処理を用いたソースコード変更予測に関する一考察”，2021 年度 (第 72 回) 電気・情報関連学会中国支部連合大会論文集，2021 年 10 月

[2] Yuto Kaibe，Hiroyuki Okamura，Tadashi Dohi，”A Note on Prediction of Source Code Changes with Natural Language Processing”，Asia Pacific International Symposium on Advanced Reliability and Maintenance Modeling (APARM 2022)，October 2022.