

# CSCI 680 AI for Software Engineering: Assignment 2

Enze Xu  
exu03@wm.edu

Yi Lin  
ylin13@wm.edu

## 1 Introduction

The If-statement prediction task aims to generate missing If-statements in source code automatically. As shown in the two code snippets below, the input is a Python method code snippet containing a special placeholder <fill-in>, where the If-statements should be (left). The model needs to predict the specific If-statement replacing <fill-in> (right).

```
# The input example
def _send_from_command_queue(self):
    <fill-in>
        if not len(entry) == 4:
            continue
        cmd, cmd_type, callback, tags = entry
    else:
        cmd = entry
        cmd_type = None
        callback = None
        tags = None

# The output example
if isinstance ( entry , tuple ) :
```

In this assignment, we train a Transformer model, CodeT5, to predict If-statements in Python methods. Before fine-tuning CodeT5 for the prediction task, we first randomly mask 15% of the tokens in complete Python methods and pre-train the model using the masked language modeling (MLM) method, where the model should predict the masked tokens. This step helps the model gain a deeper understanding of the structure and semantics of the corpus, which enhances the prediction performance and improves the model’s generalizability. We then fine-tune the model on an If-statement prediction dataset. We scraped and processed a total of 148,814 of Python methods from public GitHub Repositories (powered by GitHub Search Tool) for experiments. The source code for our work can be found at [https://github.com/EnzeXu/CSCI680\\_If\\_Statement](https://github.com/EnzeXu/CSCI680_If_Statement). All dataset files necessary for our work are available on OneDrive.

## 2 Implementation

### 2.1 Dataset Preparation

**GitHub Repository Selection.** We begin by using the GitHub Search Tool (<https://searx-github.usi.ch/>) to compile a list of well-developed repositories, applying the following filters: language=“Python”, minimum number of commits=20, maximum number of commits=50, minimum number of stars=50. These criteria yield a total of 15,485 repositories.

**Repository Cloning and Preprocessing.** Using Python’s `subprocess` package, we clone the selected repositories from GitHub. We then filter out any files that did not have a “.py” extension. The remaining Python files are renamed, and we apply the `re` package to strip out comments and blank lines. After preprocessing, we determine that the 15,485 repositories contain 148,814 Python functions.

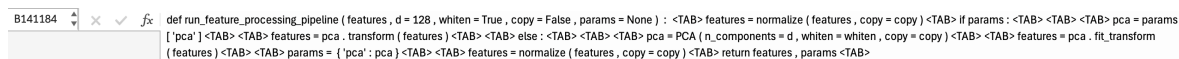
**Code Tokenization.** We apply the built-in `re` and `tokenizer` package to tokenize the code at this step, splitting terms with space and replace the tabs by <TAB> separators.

**Data splitting.** We randomly divided the processed Python functions into 148,814 functions as the pre-training dataset and 50,000 functions as the fine-tuning dataset. The two datasets were further divided into training set, validation set and test set in a ratio of 8:1:1, respectively.

## 2.2 Pre-training

We pre-train the CodeT5 model by using the masked language modeling method. We randomly mask roughly 15% of the Python method in the pre-training dataset and ask the CodeT5 to guess them. In statistics, the average number of tokens in each function-level code is 186.3 (max: 821, min: 100, std: 75.0), and we fix 30 as the number of tokens to be masked. To be specific, pre-training includes the following steps.

**Load data, tokenizer, and model.** We first load the data from the .csv file using pandas, where the Python methods in this step don’t contain the special placeholder `<fill-in>`, as shown as Figure 1. We utilize the pre-trained tokenizer and model of CodeT5 to do the tokenization and pre-training. This tokenizer allows us to extract tokens by interpreting the syntactic structure of the code so that some word like “method\_name” can be split into “method”, “\_”, and “name”. It can also give each split token an ID. We use the “Salesforce/codet5-base” version of CodeT5 model.



```

B141184 x ✓ fx def run_feature_processing_pipeline ( features , d = 128 , whiten = True , copy = False , params = None ) : <TAB> features = normalize ( features , copy = copy ) <TAB> if params : <TAB> <TAB> <TAB> pca = params
[ 'pca' ] <TAB> <TAB> features = pca . transform ( features ) <TAB> <TAB> else : <TAB> <TAB> <TAB> pca = PCA ( n_components = d , whiten = whiten , copy = copy ) <TAB> <TAB> features = pca . fit_transform
( features ) <TAB> <TAB> params = { 'pca' : pca } <TAB> <TAB> features = normalize ( features , copy = copy ) <TAB> return features , params <TAB>

```

Figure 1: Example of collected raw Python function

**Mask the pre-training dataset.** We first filter out all the `<TAB>` in the original input methods. Then we mask 15% of the input method texts by using `random.sample()` function. Since we notice that the average method length of our pre-training data is about 186.3, in order to prevent the length of the masked words lists in the ground truth from varying too much, we fix the number of masked words to be 30, approximately 15% of the average method length. We then save the processed input masked texts (with `<extra.id.X>` placeholders) and the corresponding labels with masked word lists in the pickle format.

**Pre-train the model.** We first load the saved pickled data. Then split the data into train, val, and test sets in a ratio of 8:1:1 and create a dataloader using the `DataLoader` function from `torch.utils.data`. By using the tokenizer introduced above, the input of the model is the ID lists of masked method texts and the objective of the pre-training is that the output of the model’s prediction IDs should be the same as the IDs in the masked word lists in ground truth. CodeT5 calculates the loss by comparing the model’s predicted token probabilities (logits) at each position with the actual target tokens provided in the sequence, so we applies the token-level cross-entropy loss to compute a loss for each masked word individually. We pre-train the CodeT5 model with the following configurations: `epoch=100`, `batch_size=16`, `lr=0.001`. We use AdamW optimizer and a linear scheduler. We finally save the pre-trained weights (`state_dict`) of the CodeT5 model.

## 2.3 Fine-tuning

**Data Preprocessing.** In the data collection process, we filtered Python functions to include only those containing an if statement. For each function snippet, we then randomly select one if statement and apply the CodeT5 tokenizer to tokenize both the masked code and the `target_block`. For instance, the code shown in Figure 1 is masked as depicted in Figure 2, with the corresponding `target_block` illustrated in Figure 3.

**Train the model.** We load the saved data and filter samples where the token-level length of the selected if statement falls between the **25th and 75th percentiles**. From these filtered samples, we randomly select 50,000 entries. The data is then split into training, validation, and test sets in an 8:1:1 ratio (later we will use the test set to generate the `generated-testset.csv` file, which has

Figure 2: Example of Python function that the if statement has been masked as `<fill-in>` (we use the same format as the given test set)

Figure 3: Example of collected raw Python function

50,000 \* 10% = 5,000 entities), and a DataLoader is created using `torch.utils.data.DataLoader`. For training, we apply token-level cross-entropy loss to compute an individual loss for each masked word. The CodeT5 model is trained with the following configurations: epochs=100, batch size=32, and learning rate=0.001. We use the AdamW optimizer along with a linear learning rate scheduler. Finally, we save the model’s pretrained weights (`state_dict`) for evaluation use.

### 3 Evaluation

#### 3.1 Dataset

We have evaluated the performance of our trained model on two datasets:

- **sample.csv**: This provided file contains 30 entities. The results for this dataset are saved as `provided-testset.csv`.
- **test\_dataset.csv**: This file, containing 5,000 entities, is generated during the fine-tuning (training) step. The results for this dataset are saved as `generated-testset.csv`.

#### 3.2 Metrics

To evaluate how well the predicted “if statement” matches the ground truth “if statement”, we use two types of similarity metrics:

- **Score 1 (embedding cosine similarity)**: We use a pre-trained CodeBERT model to extract the embedding vector (last hidden state) of each string, then compute the cosine similarity between the two vectors. Since the cosine similarity ranges from  $[-1, 1]$ , we transform it to a  $[0, 100]$  scale using  $(s_1 + 1) \cdot 50$ .
- **Score 2 (sequence matcher)**: We apply a sequence matcher (`difflib.SequenceMatcher`) to compute the sequence-level similarity ratio between the two strings. This ratio ranges from  $[0, 1]$ , so we convert it to a  $[0, 100]$  scale using  $s_2 \cdot 100$ .

#### 3.3 Experimental Result

The performance on these two test dataset is shown in Table 1. From the result we found the performance of our trained model is not excellent yet good in predict the if statement conditions. Detailed examples can be found in the output file `provided-testset.csv` and `generated-testset.csv`.

Dataset	Dataset Size	Correct Ratio	Score 1		Score 2		Score Avg	
			avg	std	avg	std	avg	std
sample.csv	30	26.67%	93.98	4.89	63.22	27.72	78.60	15.99
test_dataset.csv	5,000	17.40%	92.88	5.94	60.22	27.73	76.55	15.96

Table 1: Performance Evaluation on Different Datasets