GenAI for Software Development: Assignment 2


Ted Tran — tmtran03@wm.edu
Quang Hoang — qhoang@wm.edu
Justin Liu — jliu48@wm.edu


## 1 Introduction

CodeT5 is a pre-trained encoder-decoder Transformer model designed for code understanding and generation. It has been trained on a large corpus of code across multiple programming languages and supports a range of downstream tasks such as code completion, summarization, translation, and generation. In this assignment, we fine-tune CodeT5 specifically for predicting missing if conditions in Python functions. We use the small version of CodeT5, known as codet5-small, which contains approximately 60 million parameters. This lighter variant offers a good trade-off between performance and computational efficiency. The source code for our work can be found at **https://github.com/theantigone/Fine-Tuning-CodeT5**

## 2 Implementation

### 2.1 Dataset Preparation

**Dataset:** We load the cleaned Python datasets (50,000 methods for training, 5,000 methods for validating, and 5,000 methods for testing) as **pandas DataFrames**. Then, we flatten the code to remove variability in spacing and newlines and mask the targeted if statements using **Regex** substitution in those datasets. Moreover, we ensure that the regex was designed to capture the correct portion of the if-statement by handling trailing colons and variable whitespace to correctly identify the targeted conditions.

**Pre-trained Model & Tokenizer:** We load the **CodeT5-Small Transformer model**, convert the DataFrames to **DatasetDicts** from the **datasets** library to prepare them for pre-processing, and then tokenize the cleaned methods using **RoBERTa**'s **RobertaTokenizer** Python package, with max_length=256 for the masked code (inputs) and max_length=128 for the targeted if statement (targets).

### 2.2 Model Fine-Tuning

**Model Training & Testing:** We train the model over 5 epochs, with hyperparameters learning_rate=5e-5 and batch_size=64 to prevent disruption of learned weights and find that the training losses average to 0.19835 and that the validation losses average to 0.174241. The test evaluation loss is 0.07172377407550812. We also implemented early stopping with a patience of 3 epochs to avoid overfitting and ensure robust generalization, which is critical due to the sensitive nature of code generation tasks. Below are the results:

| Epoch | Training Loss | Validation Loss |
|---|---|---|
| 1 | 0.086400 | 0.072825 |
| 2 | 0.080800 | 0.070276 |
| 3 | 0.076500 | 0.068876 |
| 4 | 0.073000 | 0.068293 |
| 5 | 0.072300 | 0.068212 |

**2.3 Model Evaluation**

**Model Evaluation:** We refer to the **SacreBLEU** metric to compute the BLEU score. We find that CodeBLEU's evaluator output is:

- N-gram match: 0.8403698784297634,
- Weighted N-gram match: 0.9028705687141984,
- Syntax match: 0.9057959361283487,
- Dataflow match: 0.878970376226573.

The **CodeBLEU** score is 0.8820016898747209, demonstrating a strong understanding of the syntax and logic of the trained data, and the average BLEU-4 score is 40.23703556, giving us a 26.84% accuracy, indicating a good translation between natural language and machine language. **It is important to note that the CodeBLEU score is computed as a corpus-level metric which means that a single global value is derived from the entire test set**. Consequently, when reporting results in the CSV file, the CodeBLEU prediction score is the same number for every row. In contrast, BLEU-4 can be computed on a per-sample basis, and thus each row may show a different BLEU-4 score. This distinction reflects the inherent design of CodeBLEU to capture structural and semantic nuances across an entire codebase, while BLEU-4 focuses on n-gram overlap at the individual sample level. In addition to these metrics, we also computed the exact match (EM) for the if conditions.