GenAI for Software Development: Assignment 2

Benjamin Tremblay
bptremblay@wm.edu

Rowan Miller
rvmiller@wm.edu

1 Introduction

Building a model capable of performing specific text prediction actions from the ground up requires enormous amounts of training data and time. This is expensive and wasteful. A vastly more efficient option available to programmers and researchers is to fine-tune an existing model to better fit their specific needs. We have created a system of fine-tuning for the publicly available CodeT5 transformer model which makes it well-suited for the specific task of predicting the content of a Python `if` statement.

2 Implementation

2.1 Dataset Preparation

We divided the provided data into three sections: Test, Train, and Validation. The Train and Validation data would be used for fine-tuning CodeT5 to fulfill our tasks. The Test data would be used to score the resulting fine-tuned model. Whitespace around special characters was removed so that the data could be processed by the training algorithm (the target_block and cleaned_method originally did not match up), and the method combined into a single line, a process called flattening. The data was then modified such that the contents of `if` statements were replaced with a mask token called `<MASK>`, using the following pattern:

2.2 Fine-Tuning

The flattened and masked data was then tokenized, and parameters were set for the model training. We then ran a Python Torch trainer through five epochs on the flattened, masked, and tokenized data. After each epoch, an accuracy score was calculated for the training data and validation data. These were outputted as Training Loss and Validation Loss respectively. Over time, both generally decreased, signifying that the model was becoming more accurate. The training loss steadily declined, passing the validation loss on the third epoch. In fact, the validation loss increased on the fifth epoch, suggesting that it had approximately reached a minimum before becoming slightly overfit. During the course of working on the project, we used a Jupyter file rather than a Python one so that we would be able to make modifications to individual pieces of our code without running code redundantly. This can be found in `IfPredicter.ipynb` in our GitHub repository.

This was tremendously helpful in streamlining work, as the training process of five epochs lasted for multiple hours.

Epoch	Training Loss	Validation Loss
1	0.042700	0.037807
2	0.038800	0.035825
3	0.031400	0.035077
4	0.027400	0.034861
5	0.021300	0.035198

3 Results

We evaluated the model on three different metrics: BLEU-4, CodeBLEU, and Exact Match. The scores are provided below:

BLEU-4	CodeBLEU	Exact Match
0.4404222563	0.2896350403	0.2948

The model performed quite well in general. The BLEU scores tended to be highest, with many of the generated predictions achieving a perfect score. The CodeBLEU scores tended to be much more moderate, however. This is likely due to the fact that CodeBLEU takes into account code syntax and conventions. For generated code that fails to meet these parameters but is linguistically highly similar to the test data, the BLEU score will outperform CodeBLEU. Exact Match surprisingly produced the second lowest score of the three; perhaps CodeBLEU fell behind due to the combination of multiple metrics. Since it is impossible to perfectly predict text reliably, an exact match frequency of close to a third is perfectly satisfactory. The `testset-results.csv` file contains the evaluations from each eval metric, giving a score for each individual test case.