GenAI for Software Development: Assignment 2

Joshua Pomeroy
jmpomeroy@wm.edu

Introduction

The task at hand is to implement a system that can predict missing if statements in Python functions. To do this a pre-trained large language model is fine tuned for this task. The model is fine tuned by feeding it a large corpus of masked Python functions, the masking is just done on a single if statement per function. The corpus of code used to train, evaluate, and test the model have been provided by the instructor. Code must then be preprocessed and tokenized before training the model. The model is then evaluated by multiple metrics such as BLEU score, CodeBLEU Score, syntax match, and data flow match. F1 score is not used as there isn't really any way to consider a false positive vs a true positive and a false negative, instead a percentage of exact matches is calculated.

Implementation

Data Set Preparation

Tokenization: To tokenize the code corpus Pygments is used. The provided data files contain which if statement in each method should be masked, however these if statements have been tokenized while the methods themselves have not yet been. As such, the methods are tokenized before masking.

Flattening: Flattening is done prior to masking as the Pygments tokenizer produces a list where my masking implementation needs a string. A custom join() function is used to flatten the code and rejoin the tokens into a single string. This is required because newlines and tabs need to be joined without adding a space after the token where other tokens require a space after them.

Masking: To mask each method a simple for loop is used in combination with Python's built in .replace() method. This method automatically replaces the first found instance of a given string with another. As such, with both the full method and the target if statement being tokenized in the same format, it is easy to go through every single Python method in the corpus. Any block of 4 consecutive spaces is replaced with a <TAB> character to preserve indentation, new lines on the other hand are simply removed. The final masked method is written to a csv file along with the target if statement.

Model Training: The model training is largely done with the code given to us in class. The only change is to the number of epochs. I decided to train multiple models, one with 5, 7, 8, and 9 epochs respectively.

Evaluation: I evaluated all 4 models. First I ran the CodeXGlue evaluator in the command line on the generated outputs and the target outputs to find which out of the 4 models was performing the best. These are the results:

5 Epoch Model
- ngram match: 0.44076405557821713
- weighted ngram match: 0.43816486064632887
- syntax_match: 0.4700093530153308
- dataflow_match: 0.2876712328767123
- CodeBLEU score: 0.4091523755291473

7 Epoch Model
- ngram match: 0.44505721302663975
- weighted ngram match: 0.44301788524657626
- syntax_match: 0.4743605384083608
- dataflow_match: 0.2950913242009132
- CodeBLEU score: 0.4143817402206225

8 Epoch Model
- ngram match: 0.44667558746118796
- weighted ngram match: 0.4448553710503993
- syntax_match: 0.4776137611321215
- dataflow_match: 0.2819634703196347
- CodeBLEU score: 0.4127770474908359

9 Epoch Model
- ngram match: 0.4500318574765394
- weighted ngram match: 0.4479429270841375
- syntax_match: 0.47993168232280103
- dataflow_match: 0.295662100456621
- CodeBLEU score: 0.4183921418350247

From these results I decided to use the 9 Epoch Model to generate the final results csv as it had the highest scores. The percentage of exact matches from the 9 epoch model is 28.78% with 1439 exact matches.

Example Output: **Recommended to view full output in table form using an extension or downloading the csv and opening in Excel/Google Sheets**

Input Function	Expected If	Predicted If	CodeBLEU Scor	BLEU-4 Score	Exact Match?
def read (self , count = True , timeout = None , ignore_non_errors = True , ignore_timeouts = True) : <T	if ignore_timeouts and is_timeout (e) :	if ignore_timeouts and is_noerr (e) :	75.22	73.49	FALSE

Example Masking and tokenizing:

Before:

```
def _resolve_lib_imported_symbols(self, lib, imported_libs, generic_refs):
    """Resolve the imported symbols in a library."""
    for symbol in lib.elf.imported_symbols:
        imported_lib = self._find_exported_symbol(symbol, imported_libs)
        if not imported_lib:
            lib.unresolved_symbols.add(symbol)
        else:
            lib.linked_symbols[symbol] = imported_lib
            if generic_refs:
                ref_lib = generic_refs.refs.get(imported_lib.path)
                if not ref_lib or not symbol in ref_lib.exported_symbols:
                    lib.imported_ext_symbols[imported_lib].add(symbol)
```

After:

```
def _resolve_lib_imported_symbols ( self , lib , imported_libs , generic_refs ) : <TAB> """Resolve the imported symbols in a library."""  <TAB> for symbol
 in lib . elf . imported_symbols : <TAB> <TAB> imported_lib = self . _find_exported_symbol ( symbol , imported_libs ) <TAB> <TAB> if not imported_lib : <
TAB> <TAB> <TAB> lib . unresolved_symbols . add ( symbol ) <TAB> <TAB> else : <TAB> <TAB> <TAB> lib . linked_symbols [ symbol ] = imported_lib <TAB> <TAB
> <TAB> <MASK> <TAB> <TAB> <TAB> <TAB> ref_lib = generic_refs . refs . get ( imported_lib . path ) <TAB> <TAB> <TAB> <TAB> if not ref_lib or not symbol i
n ref_lib . exported_symbols : <TAB> <TAB> <TAB> <TAB> <TAB> lib . imported_ext_symbols [ imported_lib ] . add ( symbol )
```