GenAI for Software Development: Assignment 2

Benjamin Normann
benormann@wm.edu

1 Introduction

This project focuses on fine-tuning the CodeT5 Transformer model to predict missing “if” statement conditions in Python code. I utilized the Salesforce/codet5-small model, which contains approximately 60 million parameters, as my base model. The fine-tuning process involved locating and masking the “if” conditions in Python functions and training the model to predict these conditions based on the surrounding context. The model was trained using a sequence-to-sequence approach, taking masked code as input and generating the missing “if” condition as output. Performance was evaluated using multiple metrics including BLEU, CodeBLEU, and exact match accuracy. The source code for this project is available at: https://github.com/BenNormann/if-predictor-CodeT5/.

2 Data

Data Processing

My dataset consists of raw Python functions extracted from github via the DataHub website. The raw data is in JSONL format and contains Python functions with various formats of “if” statements. I preprocess this data by removing any function that does not contain an “if” statement and creating a new JSONL (this is the data you may find at: https://drive.google.com/drive/folders/1_4ouIZYNFAlSmozBxOlgoqmPOW4sX5Qk?usp=sharing). Then I further preprocess this data by extracting the “if” statement from the functions and creating masked versions by replacing the “if” conditions with a “<mask>” token, and generating input-output pairs suitable for training a sequence-to-sequence model.

AST-Based Extraction

I leveraged Python's Abstract Syntax Tree (AST) module to accurately identify and extract “if” statements from the code. This approach preserves the syntactic structure of the code while allowing me to accurately identify the conditions that need to be masked. The AST visitor pattern helps us locate the “if” statements, extract their conditions, and create masked versions with exact positions for replacement.

Masking Strategy

For each identified “if” statement, I replace the condition portion with a “<mask>” token, carefully preserving the surrounding structure including indentation and the trailing colon. This creates paired examples where the input is the function with masked conditions and the target output is

the original condition. I handle both single-line and multi-line conditions to ensure complete coverage of code patterns.

Dataset Splitting

The dataset is divided into training (80%), validation (10%), and test (10%) sets to support effective model training and evaluation. This split ensures sufficient training data while providing separate sets for hyperparameter tuning and final performance assessment. These processed datasets are saved as CSV files with paired examples of masked and original code snippets.

3 Implementation

Model Architecture

I use the CodeT5 model, a Transformer-based encoder-decoder architecture, pre-trained on a large corpus of code. For my task, I fine-tune the codet5-small variant which offers a good balance between performance and computational efficiency.

Tokenization

The input and target sequences are tokenized using CodeT5's specialized tokenizer, which understands code specific tokens and structures. I augment the tokenizer's vocabulary with a special “<mask>” token to represent the position of missing “if” conditions.

Fine-tuning Process

The model is fine-tuned using a sequence-to-sequence approach where the input is a Python function with masked “if” conditions and the output is the predicted condition. I use an early stopping strategy based on validation loss to prevent overfitting while ensuring sufficient learning. The training process uses teacher forcing to provide the model with the correct previous tokens during training to stabilize learning.

Hyperparameter Optimization

Key hyperparameters include learning rate (5e-5), batch size (8), and weight decay (0.01). These values were selected to balance training efficiency and model performance. The training runs for up to 5 epochs with early stopping patience of 3 epochs to find the optimal model weights.

Evaluation Metrics

I evaluate the model using multiple metrics:

- BLEU scores measure n-gram overlap between predicted and reference conditions
- CodeBLEU extends BLEU by incorporating code-specific features like syntax and data flow
- Exact match accuracy identifies perfectly predicted conditions

Results Analysis

The model demonstrates strong performance in predicting common “if” statement patterns and standard comparisons. It handles numerical comparisons, equality checks, and logical operators effectively. The model performs best when the “if” condition follows standard patterns found in training data. It struggles with handling complex or niche specific situations, especially those that rely heavily on specific variable names or unique project logic.