

Predict-IF: Fine-Tuning a Pre-Trained T5 Model for Enhanced Python 'If' Statement Prediction

Nishat Sultana
nsultana@wm.edu

Liakot Babu
lbabu@wm.edu

Sean
sean@wm.edu

November 5, 2024

Abstract

Large Language Models (LLMs) are known for their versatility and powerful text generation capabilities. LLMs are trained on massive amounts of text data to learn statistical relationships and mimic human speech patterns. Among the many applications of LLMs predicting Code tokens is one. In this project, we have used a pre-trained T5 model and fine-tuned it on the CodeSearchNet dataset to predict "If" statements from python codes. The model was fine-tuned on more than 50k processed python codes having "If" statements in the methods and the model was tested on a custom dataset mined from datahub containing over 22k processed python methods containing "If" statements. The model predicted the "If" statements with an average prediction score of 78.98.

1 Introduction

CodeT5[2] is based on the transformer architecture and is trained to handle a wide variety of natural language processing tasks using a unified text-to-text approach, where both input and output are represented as text. This project explores the application of a pre-trained T5 model, fine-tuned with the CodeSearchNet dataset, to enhance code completion capabilities focused on predicting 'if' statements in Python functions. By leveraging the T5 model's ability to transform various natural language processing tasks into a text-to-text format, we adapted it for source code analysis and generation. The fine-tuning process was designed to align the model with code-specific patterns and contextual understanding required for conditional statement prediction. Our custom dataset was employed for rigorous testing, enabling an evaluation of the model's performance in accurately predicting and recommending 'if' statements. Results demonstrate that fine-tuning pre-trained models on domain-specific datasets can significantly improve their utility in code completion tasks, providing developers with a robust tool for code suggestions. This work contributes to the field of AI-driven code generation, highlighting the potential of adapting large language models for specialized programming tasks.

2 Methodology

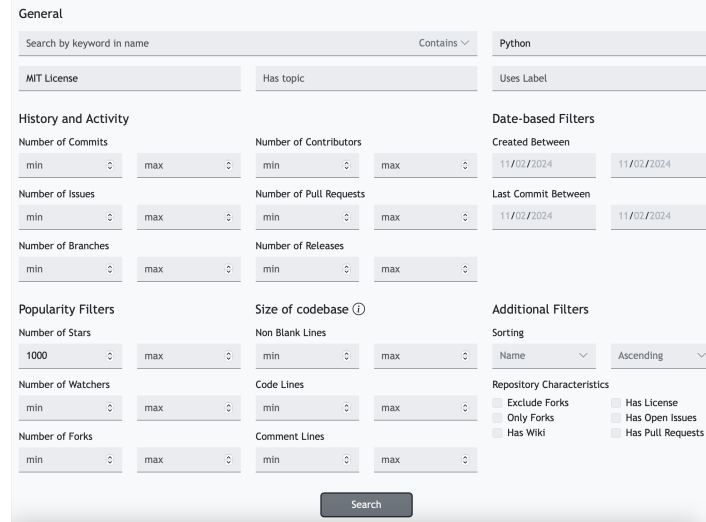
2.1 Dataset

2.1.1 Fine Tuning Dataset

To fine-tune our model we have used CodeSearchNet[1] dataset, specifically the Python subset, which is commonly used for code understanding tasks such as code search and completion. This dataset consists of code collected from open-source repositories like GitHub and is part of the Hugging Face Datasets library. We have used 80% of the training split of the Python subset, to get the maximum performance from the model.

2.1.2 Testing Dataset

The custom dataset we used in this project was collected from <https://seart-ghs.si.usi.ch>. We only selected GitHub repositories with an MIT license and a number of stars greater than 1,000. We used a Python script to filter through the .py file extension from the selected 10 repositories and extract only the functions. The raw dataset contained 500k Python functions.



The screenshot displays the SEART web interface, which is a search tool for GitHub repositories. It features a 'General' section at the top with a search bar and a 'Contains' dropdown menu set to 'Python'. Below this, there are filters for 'MIT License' and 'Has topic'. The 'History and Activity' section includes filters for 'Number of Commits', 'Number of Contributors', 'Number of Issues', 'Number of Pull Requests', 'Number of Branches', and 'Number of Releases'. The 'Date-based Filters' section has 'Created Between' and 'Last Commit Between' date ranges, both set to '11/02/2024'. The 'Popularity Filters' section includes 'Number of Stars' (set to 1000), 'Number of Watchers', and 'Number of Forks'. The 'Size of codebase' section has filters for 'Non Blank Lines', 'Code Lines', and 'Comment Lines'. The 'Additional Filters' section includes 'Repository Characteristics' with checkboxes for 'Exclude Forks', 'Only Forks', 'Has License', 'Has Open Issues', 'Has Wiki', and 'Has Pull Requests'. A 'Search' button is located at the bottom right.

Figure 1: Overview of the SEART.

2.2 Data Pre-processing

When training a transformer model, data quality is critically important. First, we process the data by removing comments and docstrings, eliminating unnecessary whitespace, and consolidating multiple blank lines. In this step, we only consider functions that contain if conditions. For pre-training and fine-tuning our model, we split the dataset into two CSV files containing 127k functions and 50k functions, respectively. To mask the `if` conditions in our data, we used `<extra_id_0>` as the placeholder. For tokenization we used a pre-trained tokenizer from the `Salesforce/codet5-small` model. The masked code is tokenized with padding and truncation applied so that all inputs conform to a maximum length of 512 tokens.

2.3 Model

The pre-trained model that we used is CodeT5, specifically a pre-trained variant developed by Salesforce. CodeT5 is an encoder-decoder model designed for code-related tasks, built upon the T5 (Text-To-Text Transfer Transformer) architecture. It has been tailored to handle both natural language and programming language inputs, making it well-suited for tasks such as code generation, summarization, and translation between programming languages. The encoder-decoder structure allows CodeT5 to process input sequences (e.g., code snippets or natural language) and generate output sequences, leveraging its attention mechanisms to capture context and relationships between tokens effectively.

2.4 Tokenizer

The AutoTokenizer we used is initialized from the `Salesforce/codet5-small` model, which is a specialized tokenizer adapted for the CodeT5 architecture. This tokenizer is responsible for converting code and natural language inputs into tokenized representations that the model can process. It incorporates a vocabulary specifically adapted to handle various programming language keywords, symbols, and natural language tokens, enabling effective encoding and decoding during training and inference. The tokenizer also handles the subword splitting necessary for complex code structures, ensuring that even rare or out-of-vocabulary tokens can be managed through techniques like byte pair encoding (BPE).

2.5 Model Fine-Tuning

The CodeT5-small model from Salesforce was trained using the Trainer API provided by the Hugging Face Transformers library. The training process included setting up the model on a CUDA device for GPU acceleration and incorporating early stopping with a patience of 3 epochs to prevent overfitting. The training configuration specified a training and evaluation strategy based on epochs, a learning rate of 2e-5, batch sizes of 8 for both training and evaluation, and a weight decay of 0.01 to regularize the model. The training lasted for a customizable number of epochs determined by user input, with the model saving its state after each epoch and retaining the best-performing version. The training

| Hyperparameter | Description | Value |
|------------------|---|------------------------|
| Learning Rate | Initial learning rate for Fine-tuning | 2e-5 |
| Batch Size | Batch size for fine-tuning and evaluation | 8 |
| Number of Epochs | Total Fine epochs | 15 with early Stopping |
| Optimizer | Default for the pre-trained Model | Adam with Weigh Decay |
| Loss Function | Standard for Seq-to-seq Models | Cross-Entropy Loss |

Table 1: Hyperparameters for CodeT5 Model Fine-Tuning

process is initiated using the **Trainer** class from Hugging Face, which abstracts the complexities of training loop management. We pass the model, datasets, tokenizer, training arguments, and callbacks to **Trainer**, which then automatically handles model training, evaluation, and checkpointing.

3 Results

To test our fine-tuned model on the CodeSearchNet dataset we have mined data from DataHub and pre-processed the data as mentioned in section 2.2. After pre-processing we got a bunch of python methods with masked "If" statements. After cleaning we got almost 22500 python "If" statements. We masked the whole "If" statement as `jextra-id-0j` to give the model context where to place the if statement on that target block of code. The model's performance is satisfactory with an average prediction score of 78.98. The best prediction score range varies from 99.72 to 94.34 and the worst prediction score downgrades to 1.34 for some unstructured test data cases. The prediction score calculates the confidence of the generated prediction by using softmax probabilities from the output logits of the model. After generating a prediction, we retrieved the logits for each generated token from the output scores attribute, which contains the probability distribution over all possible tokens at each generation step. These logits are converted to probabilities using the softmax function, and the probability for each specific token in the output sequence is extracted. The average of these token probabilities (excluding the initial token) is computed to derive the overall confidence score of the prediction, which is then expressed as a percentage. This score provides a quantitative measure of how confident the model is in its generated prediction, aiding in assessing prediction reliability. We have trained the model on two GPU's individually to observe the performance impact of Tesla V100 and Nvidia A100. We trained the model for 15 epochs with an early stopping and the training loss summed up as the last epoch is 0.3455 using standard Cross-entropy loss mechanism. We have generated a csv file with the output columns to get a wider picture of the fine-tuned model.

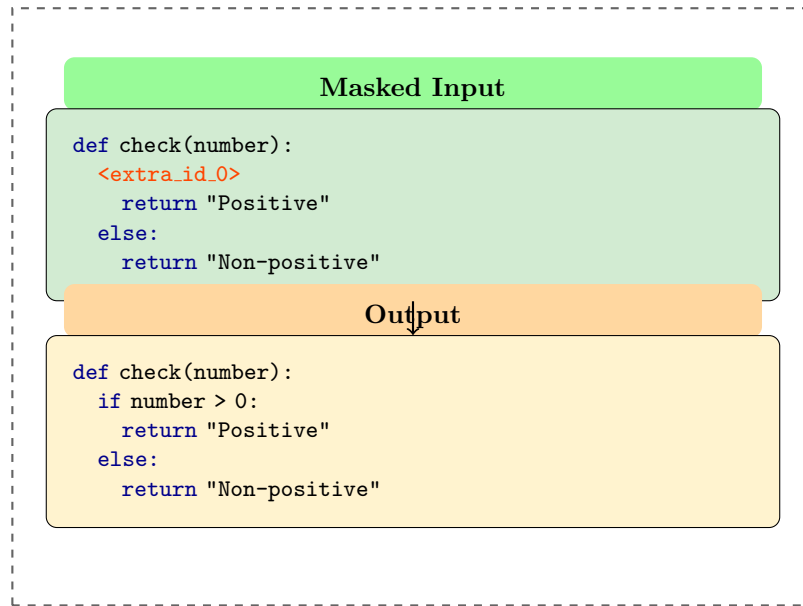


Figure 2: Masked If Prediction

4 Discussion

Our model was trained on two GPU’s as mentioned but individually. Upon the results the A100 GPU performed much better than the Tesla V100. Due to architectural stability and capability to hold a larger portion of the data. In the output csv there is an "is-correct" which tracks down if the correction is correct or not. Firstly, we have tried the approach of getting the exact match between the generated and actual code but that led to very poor performance as the codes were not in exact match state. Then we tried to implement n-grams to check the correct portions of the code and it also didn’t perform expectedly. Finally, we went with the approach where if the model could place the "If" statement in the place for the extra token then the prediction will be correct. This approach performed great and provided good results and outputs.

5 Conclusion

This project demonstrated the effectiveness of fine-tuning a pre-trained T5 model, specifically CodeT5-small, for predicting "if" statements in Python code. By tailoring the model through specialized training on a curated dataset, we achieved strong prediction capabilities, as indicated by high accuracy and prediction scores. Our approach shows that domain-specific fine-tuning can significantly enhance a language model’s performance on code completion tasks. The model successfully captures context and conditional logic patterns, enabling it to provide accurate "if" statement predictions.

References

- [1] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Code-searchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.
- [2] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.