Source Code Summarizer

Nisha Jacob
Department of Electrical and Computer Engineering
University of Southern California
nejacob@usc.edu

Muskaan Parmar
Department of Electrical and Computer Engineering
University of Southern California
muskaanp@usc.edu

Shreeram Narayanan
Department of Electrical and Computer Engineering
University of Southern California
shreeram@usc.edu

***Abstract*—Summarization is the process of using deep learning techniques on a long text description to generate a short summary about the input text. This project aims to create a deep learning model to generate a natural language summary of source code in Python i.e. it will describe the functionality of a python program. We propose an encoder-decoder architecture with the encoder being a pre-trained Roberta Model and a decoder with a multihead attention mechanism. We also compare the results of our model with CodeT5 transformer and a Roberta pre-trained model. We study the variation in model performance between the said pre-trained models and our model architecture. The performance metrics used to evaluate models is BLEU-4 and EM.**

***Index Terms*—tokenization, encoder, decoder, RoBERTa, Code T5, BLEU Score, EM Score**

I. Introduction

A source code summarizer can help programmers generate a summary for the functionality of the code that they have written. This summary is in the form of short text that highlights the key concepts and structure of the code. Programmers can save time and effort in describing what their code performs by making use of the code summarizer. Having a short summary of the code functionality also helps developers in understanding a piece of code that was written by someone else. As program maintenance is expensive for companies, making use of a code summarizer would definitely help them to reduce this cost. Also, having the code stored in a database along with its summary can help developers easily find a particular piece of code in case they wish to reuse it. It would also allow users to look up codes written by another developer to understand how they had approached the problem.

The code summarizer is a low cost solution for software development and maintenance which can be used by any companies to maintain their codes in an efficient manner. Figure 1 shows the code along with the docstrings present within the three double quotes. We can extend the work of code summarization to different programming languages which would help companies generate summaries of any type of code.

In this work, our goal is to customise the Roberta based model that can be used in various Program and Language Understanding tasks. The Roberta Model employs denoising sequence-to-sequence training. This is where a transformer learns to reconstruct an original text that is corrupted using an arbitrary noise function. We study the effect of a custom decoder in the Roberta Seq2Seq architecture and how it affects the overall model performance.

```
def add_numbers(x, y):
    """
    Adds two numbers together and returns the result.
    """

    return x + y
```

Fig. 1. Example of a code with docstrings

II. Related Work

Most neural network based approaches for code summarization have used sequence to sequence models. Iyer et al.(2016) trained an embedding matrix to show individual code tokens and combined that with an RNN to generate natural language summary [1]. They make use of code snippets collected from StackOverflow. They used METEOR and BLEU-4 to measure the performance of their trained models.

Joty et al.(2021) proposes an encoder-decoder model for source code understanding and generation called CodeT5 [2]. The paper proposes an identifier-aware pre-training task that helps it to distinguish which code tokens are identifiers, set by developers. They claim that CodeT5 outperforms state-of-the-art encoder-only and encoder-decoder frameworks.

Wasi et al.(2020) uses a transformer model that uses a self-attention mechanism which effectively captures long-range dependencies [3]. The architecture that this paper has proposed employs a self attention and a copy attention mechanism. The copy attention allows the transformer to copy certain rare code tokens like function names and variable names. They propose that allowing the transformer to learn the pairwise relationship between source code tokens via relative position representations along with copy attention improves the summarization performance significantly.

The related work shows us that encoder-decoder based models perform better for summarization tasks. We propose to compare the performance of the pre trained CodeT5 and RoBERTa architectures with our custom architecture.

III. Dataset and Preprocessing

We are making use of the CodeSearchNet Corpus Dataset which contains code files for the following languages: Go, Java, Javascript, C#, C++, PHP, Python and Ruby [4]. For our project, we are making use of the 280652 Python code files from the dataset. This data is stored in JSON lines format. Each data sample is made of a dictionary which contain the following keys:

- **repo:**owner/repo
- **path:**full path to the original file
- **func_name:** function name
- **original_string:** the raw string before tokenization or parsing
- **language:** the programming language
- **code:** the part of the original_string that is code
- **code_tokens:** tokenized version of code
- **docstring:** top-level comment or docstring
- **docstring_tokens:** tokenized version of docstring
- **sha:** hash function (feature not used)
- **partition:** flag indicating what partition this datum belongs to of train, valid and test(feature not used)
- **url:** url for the code snippet including the line numbers

Two columns that are heavily used for the purpose of code summarization are code_tokens and docstring_tokens. The initial pre-processing part removes the comments and doc strings from the code section of the code section of each data sample. Indentation of the code is not taken into consideration during the summarization task.

The python files in the dataset are divided into 251820 data samples for train set, 13914 data samples in validation set and 14918 data samples in test set. The models are trained on the training data after which their performance is evaluated on the validation set. Finally, the performance of every model is compared on the test set.

IV. Architecture and Implementation

A. *Tokenization*

Tokenization is an important preprocessing step in many NLP tasks, as it allows the text to be represented in a form that is more amenable to further analysis and processing. It is the process of splitting a piece of text into smaller units called tokens. These tokens can be words, punctuation marks, numbers, or other syntactic units that are meaningful in the context of the text. The goal of tokenization is to divide the text into tokens that can be easily analyzed and processed by NLP algorithms. Our project makes use of the RoBERTa Tokenizer which makes use of a code-specific BPE (Byte-Pair Encoding). BPE looks for the most frequent byte pairs in the text and replaces them with a single byte. This process is repeated until all of the byte pairs in the text have been replaced. Once the text has been split into words and subwords, the tokenizer then assigns each word or subword a unique integer ID, which is used by the Roberta model to represent the words in its internal representation. There are some special tokens like <*pad*>, <*s*>, <*/s*>, <*unk*> and <*mask*> used in our model. These special tokens are directly added to the vocabulary and are treated carefully by the tokenizer. No splits are performed on these special tokens. We have set a vocabulary size of 32000 and the minimum frequency as 3. Minimum frequency refers to the number of times a pair of tokens should occur before a new token could be created out of it.

B. *Embedding*

In token embedding, each word is represented by a vector of real numbers, which is learned during training. The vectors for different words are initialized randomly and are then adjusted based on the context in which the words appear, so that words with similar meanings will have similar vector representations. Positional embedding is a technique in which a word is dependent on its position within the sentence. The two embeddings are performed before input is fed to the encoder and decoder.

C. *Encoder*

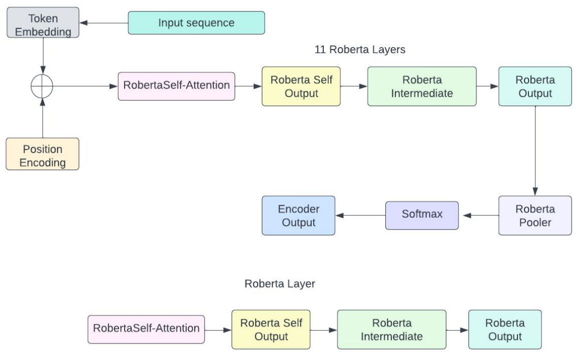


Fig. 2. Pretrained RoBERTA Encoder Model Architecture

The transformer encoder maps an input sequence to a sequence of continuous representations, which are subsequently fed to the decoder. For our implementation, we make use of the Roberta-base pretrained model as an encoder. Since the goal of models like BERT and RoBERTa is to generate a language representation model, they are basically a trained transformer encoder stack. RoBERTa relies solely on self-attention mechanism, which is made possible by the Bidirectional Transformers at the center of its design. The Figure 2 shows the Pretrained RoBERTA Encoder Model Architecture.

Masked Language Model (MLM) is used as a pretraining objective with the encoder wherein the model is trained to distinguish which tokens are identifiers and can recover them when they are masked. This reconstruction is done based on the surrounding information. The output of the encoder is embedded through token embedding (vocabulary IDs for each of the tokens) and position embedding (indicates the position of each word in the sequence) before being sent across to the decoder.

D. Decoder

For our decoder, we employ a Transformer Decoder with self attention and an encoder-decoder attention head. We used the MultiheadAttention head for these attention heads. The target embeddings, encoder output are fed to the decoder and are passed through a self attention layer. This is followed by a LayerNorm to normalize the inputs and avoid the problem of vanishing gradients. It is followed by an encoder-decoder attention head and passed through a position-wise feedforward network. Finally, a linear layer is applied before the decoder outputs are obtained. Figure 3 shows our custom decoder architecture.

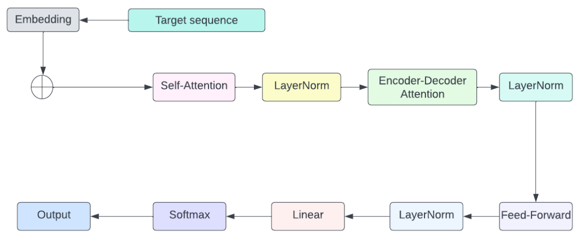


Fig. 3. Custom Decoder Model Architecture

E. Custom model

Our proposed model is has an encoder-decoder architecture employing the Sequence to Sequence architecture with encoders and decoders. The encoder is a Roberta Pretrained model and the decoder is a custom decoder as specified in Section D. The Sequence to Sequence architecture takes as input the encoder and decoder and finally applies a couple of linear layers and a softmax before the outputs are computed. There are 131M parameters in custom model.

V. Pretrained models

CodeT5: CodeT5 model has been created by Salesforce. It is trained with the help of a technique called transfer learning, which allows it to fine-tune its performance on wide range of NLP tasks. There are 60M parameters in Pretrained CodeT5 model.

RoBERTa: RoBERTa (Robustly Optimized BERT Pretraining Approach) is a state-of-the-art language model developed by researchers at Facebook AI. It is an extension of the popular BERT model, and like BERT uses unsupervised learning to learn the patterns and structures of human language from a large corpus of text data. RoBERTa is trained on a much larger dataset than BERT, which allows it to learn more about the patterns and structures of a language.There are 173M parameters in Pretrained RoBERTa model.

VI. Training Details

For our model, we have trained the model on 25 epochs. The optimizer used was Adam with weight decay and the batch size was set to 48. We set a constant learning rate of 5e-5 throughout the training process.Cross Entropy Loss was used as our loss function. It took approximately 15 hours to train the model.

The pre trained model CodeT5 were trained for 5 epochs with a batch size of 48. It took approximately 2 hours to train the pretrained model.

The pre trained model RoBERTa were trained for 5 epochs with a batch size of 48. It took approximately 3 hours to train the pretrained model.

VII. Computation Needs

For training the pre-trained and custom designed networks we used a VM instance with the following specifications-

GPU: *NVIDIA A100 40GB*
Number of GPUs: *1*
Machine Type: *a2-highgpu-1g*
Size: *500GB*
Image: *Ubuntu 18.04 LTS*

We have used Google Cloud Platform (GCP) for the same purpose.

VIII. Performance Metrics

To evaluate the performance of our model we have used two performance metrics- BLEU and EM. BLEU, which stands for ”Bilingual Evaluation Understudy,” is a metric for evaluating the quality of text generated by the model. It works by comparing the machine-generated text with a reference translation and calculating the number of matching n-grams (sequences of n words) between the two. The more matching n-grams, the higher the BLEU score. Exact Match (EM) score is often used as a simple and straightforward way to measure the accuracy of a language translation model for tasks like summarization. It is calculated by comparing the model's output to a reference or ground truth output and counting the number of times the two exactly match.

IX. Outcomes

The Table I shows our results for the validation set. The Table II shows our results for the test set. The best performing model is **CodeT5** because of its highest BLEU scores. Our custom model fails to perform as good as the CodeT5 pre-trained model. Figure 4 shows the test results of our model against two pretrained models for the given code snippet.

Scoring Metric	**Pre-trained CodeT5**	**Pre-trained RoBERTa**	**Custom Model**
BLEU-4 Score	19.3	17.01	15.16
EM Score	1.64	0.35	0.5

TABLE I
Results for models on Validation Set

X. Implementation of MLOps

We have implemented MLOps in our project by giving the users the ability to input a python code and obtain docstrings for the same. We have a web application running that takes in an input python code snippet from the user, reads the function inside it as text and is then fed to the CodeT5 Model to get

Input Code:

```
def shuffle_srv(records):
  if not records:
    return[]
  ret =[]
  while len(records) > 1:
    weight_sum = 0
    for rrecord in records:
      weight_sum += rrecord.weight+0.1
      thres = random.random()*weight_sum
      weight_sum=0
      for rrecord in records:
        weight_sum += rrecord.weight+0.1
          if thres < weight_sum:
            records.remove(rrecord)
            ret.append(rrecord)
            break
        ret.append(records[0])
  return ret
```

Results:
Custom Model: Shuffle a list of records.
Pretrained RoBERTa: Shuffle srv records.
Pretrained CodeT5: Shuffle records.

Fig. 4. Performance of the models

Scoring Metric	Pre-trained CodeT5	Pre-trained RoBERTa	Custom Model
BLEU-4 Score	19.92	17.07	15.96
EM Score	1.6356	0.3419	0.4759

TABLE II
RESULTS FOR MODELS ON TEST SET

back a short description of the functionality of the code. The Figure 5 shows a textbox on the webpage where user can input the python code to get docstrings as output.

XI. Engineering Challenges

While working on our project, we faced a lot of issues with implementing our custom decoder model with the pre-trained encoder model. Understanding how the embeddings are applied to the encoder and decoder architectures was a challenge that we faced which led to issues with the dimensions and the expected outputs of the encoder decoder architecture.

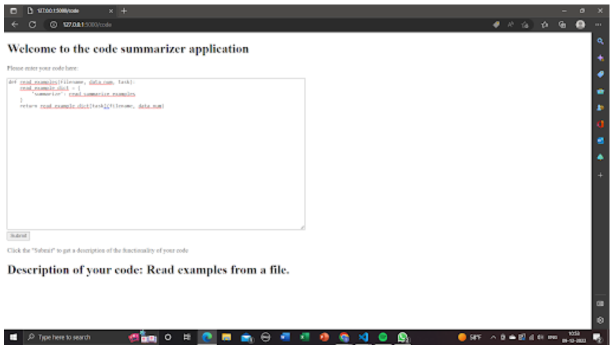
For our project the likely reasons for project failure were:

- Less available resources in the field of code summarization being a topic of Research.
- Less methodologies and approaches for carrying out summarization on snippets of code.

We were able to overcome the said challenges with extensive testing and training runs and successfully built our model.

XII. Quantifiable metrics to justify engineering tradeoffs

As per observations, we can conclude that training our model for more epochs would give better results. Accuracy and number of epochs would be fair predictors when it comes to comparing our model with existing pre-trained models. The


127.0.0.1:5000/code
Welcome to the code summarizer application
Please enter your code here:
Submit
Click the "Submit" to get a description of the functionality of your code
Type here to search

```
def read_examples(filename, data_num, task):
    read_example_dict = {
        'summarize': read_summarize_examples,
    }
    return read_example_dict[task](filename, data_num)
```

Description of your code: Read examples from a file.

Fig. 5. Application Interface for user: MLOps

pre-trained models produced better accuracies with very little training as opposed to those achieved on our model on training for the same number of epochs. Training a pre-trained model and finetuning it on CodeSearchNet dataset was more feasible in terms of resources and time taken to carry out fine-tuning. On the other hand, training a model from scratch on such a large corpus of code snippets was challenging with training times of over 15 hours.

XIII. Conclusion

We were successfully able to build our custom model for summarization. Three of us have equally contributed in building and implementing the code summarizer model and writing this report. After experiments, we conclude that the pre-trained models used gave us better results than our custom model.

One of the main question that was answered after this project was- can similar tokenization and embedding schemes be applied for different types of summarization, such as text based and code based? After implementation, we found the answer to be yes and realized that the only difference lies in what we choose to emphasize and pay more attention to.

After the completion of this project, we are still curious about how to build a model that would have the ability to understand the indentation of the code and the variables used in the code to generate docstrings. We believe that if a model is able to learn the indentation of the code and focus more on certain variables in the code it should perform better in generating text description.

Currently our model performs on obfuscated code (function and variable names don't reveal what they are actually for) and would love to work on implementing right tokenization and embedding mechanisms for the same. We have plans to work on extending the project work by replacing the current encoder model with a custom encoder model. Also, we plan to extend the work in MLOps by allowing the user to input a GitHub repository link and extracting the code part from it and generating summaries for each function present in the code.

Acknowledgment

We thank Prof. Brandon Franzke for his continuous support and guidance while working on this project.

References

[1] Iyer, Srinivasan, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. "Summarizing source code using a neural attention model." In Prceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 2073-2083. 2016.

[2] Wang, Yue, 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).

[3] Ahmad, Wasi Uddin, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. "A transformer-based approach for source code summarization." arXiv preprint arXiv:2005.00653 (2020).

[4] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, Marc Brockschmidt, "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search"(2019)

GitHub Codebases

[5] https://github.com/salesforce/CodeT5

[6] https://github.com/wasiahmad/PLBART

[7] https://github.com/wasiahmad/NeuralCodeSum

[8] https://github.com/github/CodeSearchNet#data-details