Evaluation of Transformers for Code Summarization

Ashwin Shenolikar
ash07@vt.edu
Virginia Polytechnic Institute and State University
Falls Church, VA, USA

ABSTRACT

The core part of software development and maintenance is the program code used to build it. In a software development lifecycle in actual use, it is typical for multiple people to work on it. As such, high quality comments help developers understand each others' code better. Consequently, a lack of commentary or a poor quality of code summarization can lead to significant delays in development of the product. With the advancements in Natural Language Processing, we can use Deep Learning to generate a brief description of a given code. The objective of this project is to build upon existing state-of-the-art Natural Language Understanding models, namely Transformers, and use them for the task of code summarization and evaluate their performance.

KEYWORDS

Code Summarization, Neural Networks, Transformers, Encoder-Decoder

ACM Reference Format:
Ashwin Shenolikar. 2018. Evaluation of Transformers for Code Summarization. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

With the rapid scale at which software development as a field is growing, and with new technologies constantly being created to make more functionalities possible, it is vital for teams working on tasks to have complete knowledge of all aspects of the task, even if the team members are assigned to different subtasks. Also, new members being added to the team need to be introduced to all the existing codebase. Due to these and other circumstances, it becomes necessary for all code to be comprehensible. If that is not the case, varying degrees of damage in terms of time or resources can be caused to the project development. One of the best ways to ensure that code is legible is with the use of comments.Summarising one's code in the form of a comment in the source file itself will make it very accessible to future users of the codebase, and will also give a general idea to individuals who simply need to know the function of the code. Here, another question rises. Namely, the quality of the code comment left by users is variable, and over a large number of developers, consistently good code summaries by everyone is difficult to achieve.

Code Summarization is a task which can be achieved by state-of-the-art Natural Language Processing Models, and can produce reliably good summaries for input codes. This task has gained traction in recent years due to advances in the field of Natural Language Generation. Sequential to Sequential Models used with LSTMS, Encoder-Decoders, and Transformer models are consistently performing better than their predecessors with the integration of Deep Learning Models which are able to learn context and remember it over longer inputs. This mechanism has been improved upon with attention models and now, we can have a variety of natural language tasks being solved in a convincing manner.

In this project, we will build upon state-of-the-art NLP models and fine-tune them to understand java and/or python code, and give a brief description of the function for that code. We use the amazing CodeSearchNet's java and/or python datasets for this task. We will describe the data ahead in detail, but in essence, CodeSearchNet is a collection of about 2 million sample pairs of code and their corresponding comments called 'docstrings', which act as the label for our model. While the complete Dataset has examples from the programming languages Java, Python, JavaScript, Ruby, Go and PHP; we will be focusing on java code summarization, and if time permits, we will also work to generate comments for python code. The general structure of the dataset is a code block which performs a particular task, and the comment for that block. These have been collected from various open source libraries. Our model is built upon the RoBERTA model which is obtained from the huggingface transformers library, and

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

then fine-tuned on the data. We also employ the help of the CodeXGlue repository to define various functions to help process the data. Finally, we use the CodeBERT's auto-tokenizer to generate tokens for exploring and visualizing the data.In this project, we are using the Python language with Google Colaboratory Pro to process java code snippets, which are basically java methods, and try to output their function. Python is a high-level scripting language released in 1991. It is created with readability and convenience in mind. It is the most popular language used for machine learning since there are many libraries such as numpy and pandas which do a lot of heavy lifting behind the scenes. We will use python to work with java.

Java is the globally most used programming language and is integrated into the software departments of companies belonging to mostly all domains. Java is an object oriented programming language designed with the aim of having least dependencies. Developed in 1991, it is now the most famous programming language in the world, with its development being guided by a few major goals.

- Simplicity, Object-oriented, Familiar.
- Robust, Secure
- Portability, Architecture-neutral
- High Performance capability
- Interpretability, Threading Compatibility, Dynamic.

Due to the omnipresence of Java in the software development and overall computing spheres, it was decided to focus on Java as the target language.

2 RELATED WORK

The field of Deep Learning in Software Engineering is currently very active as far as research work goes, and we have investigated related work to better inform us with our project. We will also cover the most important bases for our project in this section.

In 2017, Vaswani et al [11] presented the Transformer model to the world. Until then, the latest models used for natural language tasks were based on the Recurrent Neural network, and were extremely complex and slow to execute due to their sequential nature. However, this paper revolutionized the field by introducing the attention model. The attention model was capable of eschewing recurrence for calculations by employing the use of encoder-decoder models which could be used in parallel. This paper is the foundation for many other successors and transformers are now widely used for almost all deep learning natural language related tasks. The newest thing about the paper wasn't the attention mechanism itself, which had been previously established, but instead the way it was executed in the transformer model. A self-attention

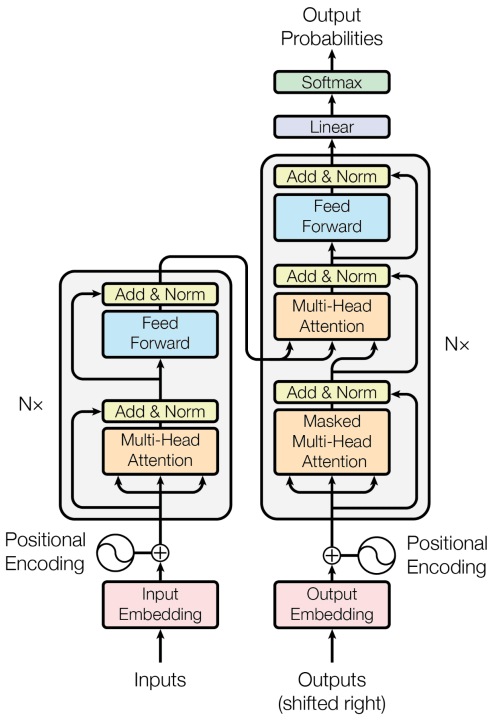

Figure 1: Transformer Self-Attention

mechanism was used for every encoder-decoder cell. The long running problem of recognizing context for machine learning models was solved previously by using memory cells to remember important words over time, however, with self-attention heads, the model was capable of tracking related words in the same example. This could be done with parallelization integration and so sped up tasks significantly.

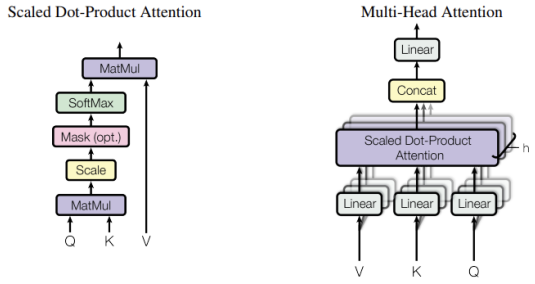

Figure 2: Transformer Architecture

The CodeBERT paper [4] used the CodeSearchNet dataset as well, albeit with the aim of teaching the model to represent general purpose representations of text which can

be comprehended in a useful way. CodeBERT builds upon the BERT paper [3], (Bidirectional Encoder Representations from Transformers) which was also originally created as a language representation model. The original BERT model was constructed to make it very easy to fine-tune it by adding an additional layer for specific tasks. It incorporates a technique called mask language modelling which is used in the CodeBERT paper, along with a Replaced Token Detection technique which was first introduced in the Electra paper [2].

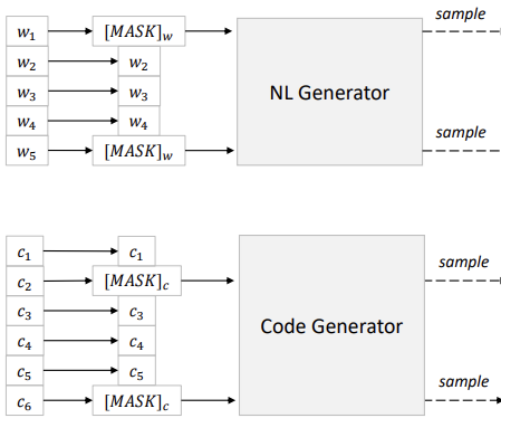

Figure 3: CodeBERT Architecture-1

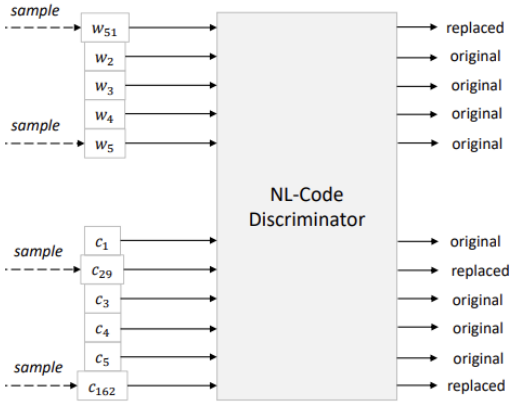

Figure 4: CodeBERT Architecture-2

In this paper [6], the authors have used a LSTM architecture with the attention mechanism in order to summarize code snippets in C as well as SQL Queries. Their model 'CODE-NN' has been created using data from StackOverflow, which is a popular open website for programming related questions. While it is widely used, the data from the website is also noisy, and a portion of the work is dedicated to data cleaning, which is not as much of an issue for our dataset. Wasi Uddin Ahmad et al.[1] in their paper, have worked to create a result similar to what we are aiming to accomplish, wherein they used a similar model with some changes to describe code snippets and achieved results comparable to the then state-of-the-art techniques to perform the same task.
LeClair et al.[7] have used incorporated a graph with neural networks to improve upon the then existing models for code summarization. They use the classification of Graph Neural Networks as follows:

- Recurrent Graph Neural Networks (RecGNNs)
- Convolutional Graph Neural Networks (ConvGNNs)
- Graph Autoencoders (GAEs)
- Spatial-temporal Graph Neural Networks (STGNNs)

However, the main focus of the paper is on the ConvGNNs, with which they used 'aggregation' techniques to learn representations.
Shido et al.[10] also focuses on source code summarization as a part of document generation from a given code. They have used an Extended Tree-LSTM architecture to perform code summarization in an attempt to solve a problem with using Abstract syntax trees (ASTs) by using a Multi-way Tree-LSTM.

3 DATA AND MODEL DESCRIPTION

3.1 Data

In this section we will review the dataset that we used to train and fine-tune our model on that dataset. The three main components of the project are:

- CodeSearchNet Dataset [5]
- CodexGLUE repository [9]
- HuggingFace [12] Models:CodeBERT [4] and RoBERTa [8]

We will describe each of these parts of the project now.

3.1.1 CodeSearchNet. The CodeSearchNet Corpus is a large collection of multiple datasets gathered from various open source projects on GitHub. This dataset is one of the biggest collections of code available, and it's format is especially useful for our task of Source Code Summarization. This is due to the fact that not only does the dataset contain code, it is segmented into code snippets and corresponding comments which work perfectly for the purpose of fine-tuning or training from scratch deep neural networks for our task. As mentioned before, the CodeSearchNet Corpus has two million preprocessed code functions with associated documentation. The distribution of the examples across languages

is given here:

	Number of Functions	
	w/ documentation	All
Go	347 789	726 768
Java	542 991	1 569 889
JavaScript	157 988	1 857 835
PHP	717 313	977 821
Python	503 502	1 156 085
Ruby	57 393	164 048
All	2 326 976	6 452 446

Figure 5: CodeSearchNet

We are going to use the java dataset from this for our project. As said before, the CodeSearchNet corpus and its datasets have been compiled from open source github repositories. As such, while we have a huge dataset to work with, we can not guarantee that the quality of this data will always be great as well. Consequently, to make a proper model based on this dataset, we need to perform a lot of data preprocessing before we feed it to our model.
First, let us look at some of the statistics we can glean from the data itself. This is possible due to tokenization using the Roberta AutoTokenizer

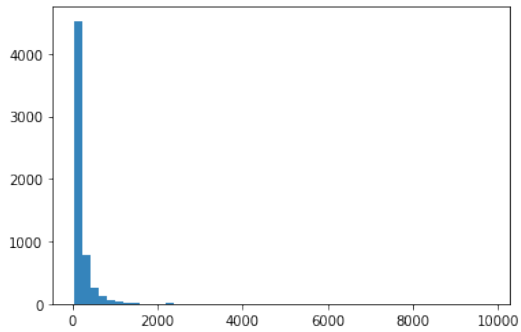

Figure 6: Histogram for Code

Also, we can look at the most commonly occuring tokens for code and comments.

Now, Comment Plot

For example, even a cursory glance over the examples given

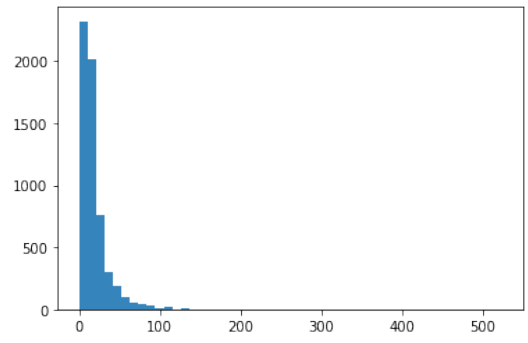

Figure 7: Histogram for comments

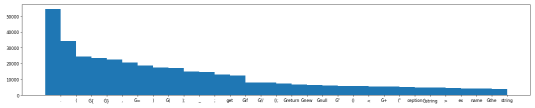

Figure 8: Code Token Frequency

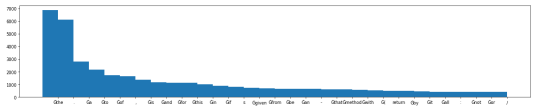

Figure 9: Comment Token Frequency

in the dataset make this problem apparent. There are many arbitrarily named function with comments that are simply placeholders and do not actually lend any meaning to the code they are supposed to describe. Apart from that, while we are selecting the java dataset, there are many examples in this dataset which include other languages, such as HTML. In such cases, we do not want to include these examples since the model will simply be thrown off and learn HTML parameters as part of java, most likely in a detrimental way. Another problem faced with this dataset is that there are pairs of codes and comments which do not actually correlate. This usually happens when a developer updates either the code or the comment, mostly the former, but does not update the other. This leads to discrepancies which are not ideal for the task.

And so, the steps for our preprocessing of the data are:

- Retrieve the java language corpus from CodeSearchNet
- Parse the dataset to determine data preprocessing steps
- Due to the inclusion of many non-ASCII characters in the dataset, where people have commented or otherwise included non-English languages as part of the data.

- Drop code-comment pairs when either one has not been updated.

These are the data processing steps that we have implemented in the model. However, there are still many examples which are simply not ideal for a model to learn from. However, there begin to be many cases which are not easily removable by code. Provided more time and more efficient code, it would be possible to produce a much better dataset for model learning. In any case, the model generated was still able to learn basic code interpretations, even if more complex code was not easily interpreted.

3.1.2 CodeXGLUE. General Language Understanding Evaluation benchmark for CODE, codeXGLUE, is a banchmark as described which consists of multiple datasets. CodeXGLUE is a benchmark with datasets which also serves to guide for various sequence to sequence programming language tasks across the categories of code-code,text-code,code-text and text-text scenarios. Namely, these tasks are:

- Clone Detection
- Defect Detection
- Clone test
- Code Completion
- Code refinement
- Code translation
- Natural Language Code search
- text-code generation
- code summarization
- documentation translation

While this is a great dataset, it also provides information about the various benchmarks for the task and helps in describing setup and preprocessing stages for anyone attempting to use codeXGLUE for one the aforementioned tasks. We have also borrowed some code from the official codeXGLUE github repository to fine-tune codeBERT model using the roberta-base model.

3.1.3 HuggingFace Transformers. Hugging Face is an open source hub of ready-to-use datasets for Machine Learning models. It also has access to very optimal and easy-to-use data manipulation tools. It allows for students to working professionals to work on aavailable datasets for various tasks and is supported and integrated by many organizations and universities and constantly updated with guides and implementations for state-of-the-art machine learning models, with guides on how to work with them and preprocess data to suit various language tasks.

In particular, hugging face transformers are especially popular due to their ease-of-use and comprehensive guides for usage. The hugging face hub provides us with pretrained latest models which can be used as is or can be fine-tuned for other tasks as well in an easy way. We use the codeBERT and RoBERTa models from the hugging face's transformers library, which has access to all or most models.

The model that we use and fine-tune is described by the following code:

```
from model import Seq2Seq
from transformers import RobertaConfig, RobertaModel

config = RobertaConfig.from_pretrained(model)
encoder = RobertaModel.from_pretrained(model,config=config)
decoder_layer = nn.TransformerDecoderLayer(d_model = config.hidden_size,nhead = config.num_attention_heads)
decoder = nn.TransformerDecoder(decoder_layer,num_layers=6)
model = Seq2Seq(encoder=encoder,decoder=decoder,config=config,beam_size=beam_size,max_length=target_l,sos_id=tokenizer.cls_token_id,eos_id=tokenizer.sep_token_id)
model.load_state_dict(torch.load(Path(output_dir)/'checkpoint-last/pytorch_model.bin'))
model.to('cuda')
```

Figure 10: Roberta Definition with fine-tuned weights

3.2 Model Description

The base of our model is the Transformer. The transformer works with the attention mechanism as opposed to classical recurrent neural network (RNN). At a very high level understanding, the transformer is a stack of six encoders followed by a stack of six decoders. The number six is not particularly special, and is open to experimentation. The encoders have the same structure: a self-attention later which then outputs into a feed forward neural network. This is a chain of operations through each encoder, after which the decoder receives the output of the final encoder. The decoders are similar to the encoders in structure, with one exception. There exists an encoder-decoder attention layer between the self-attention layer and the feed forward layer.
Now let us look at the multiple times repeated self-attention layer. At a high level, they perform the function of hidden layers in Recurrent Neural Networks. In self-attention, the first step is to create three vectors: a query vector, a key vector and a value vector. These are created by the embedding multiplication of training process. The second step is to actually calculate the self attention. This is done with the dot product of query vector and key vector, which is then followed by gradient stabilization, then passing the result through softmax. Next, we multiply with a softmax score then sum up the weighted value. The final product is the self-attention value. This is calculated at every encoder and decoder.
Another concept used is 'multi-head attention'. This is basically using multiple vectors at the same encoder to capture different representation subspaces, so that we can average the representations to find the best one.

Finally, the rest of the calculations are similar to other neural networks and easy to understand provided prior knowledge.Now that we have understood the working of a transformer, we can much better understand how these are expanded to produce CodeBERT, RoBERTa models which we

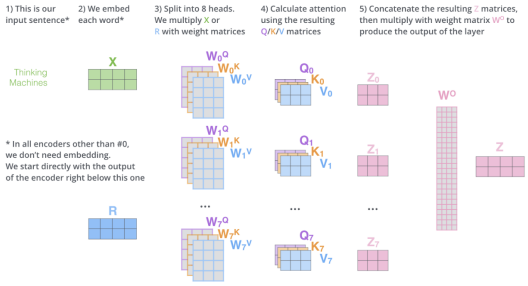

Figure 11: Self-Attention Calculation

have used in our project.

4 EVALUATION

For the purpose of Code-to-text tasks, we usually use special evaluation metrics which work better for natural language generation specifically. The classical calculations of accuracy, precision, recall and f-score do not work well on natural language, especially text generation tasks, since they fail to capture context and semantics, and are purely statistical in their calculations. On the other hand, for the case of natural language tasks, there are usually more acceptable answers than just the one provided as the label. In these cases, context, semantics and word definitions also matter.
So, for Natural Language task, there have been new definitions of evaluation metrics which are better at capturing the actual output of a model in terms of human acceptability. These models work by taking pairs of tokens in the outputs and evaluating multiple words with the output to see how much of the intended meaning is captured by these evaluation metrics. Let us now take a look at these evaluation metrics and see how they work.

4.1 BLEU Score

The Bilingual Evaluation Understudy Score(BLEU) Score is a precision focused metric that uses n-grams to do its calculations. This is done by calculating n-grams for a certain number upto n, decided by the complexity of the task. Upon calculation of all the n-grams for the reference and output texts, we check for the overlap between corresponding n-pairs of n-grams. The strength of the BLEU Score evaluation is that it quite easy to calculate, since n-gram calculations are not that expensive. Also, due to n-grams being impervious to the positional encodings or otherwise, just the positioning of words in a sentence, it can better focus on the contextual meaning of the sentence rather than being reliant on the model to do preprocessing to compensate for that.
For the task of text generation from code, BLEU Score does not always work well since it's best application is translating from text to text. However, it is a computationally inexpensive task and is still capable of capturing basic representations from code and relate it to the output string, even though it is not perfect for the task. While it is easy to write a function to calculate the BLEU score for an input, we also have various predefined data models which include their own implementation of BLEU score calculation, which we will also use for our implementation.

4.2 Rouge

The Recall Oriented Understudy for Gisting Evaluation is a widely-used metric and is commonly used alongside the BLEU Score. It is similar to the BLEU function calculation, meaning it also works on n-gram overlap. However, in terms of comparisons to the classic metrics for evaluating standard machine learning models, it can be said that while the BLEU score is more comparable to the precision of a model, the ROUGE score is more akin to the recall of a model. Unfortunately, the ROUGE score is not considered a good metric for the problem of code summarization and so we do not use it.

4.3 Perplexity

Perplexity is yet another commonly used metric for sequence to sequence model evaluation. The calculation of the perplexity of a model is done by using probabilities. Essentially, perplexity is the power of a probability distribution to assign probabilities to a sample. Naturally, the lower the perplexity of a model, the better its performance. For a completely dumb model, the perplexity is equal to the size of the vocabulary used.We have used the perplexity metric for our model evaluation.

4.4 METEOR

METEOR is a less-used metric since it works on word alignments. It computes F-score based on one-to-one mappings of output and label strings. Again, we do not use this metric since it did not give a good estimation of model performance.

5 RESULTS

At the end of our experiments, we have a model which can generate summaries for a given input code block. It is an impressive result considering that the data cleaning process was not as comprehensive, and since Java is a language that is not as easily interpretable by a machine without being impeded by factors such as data quality and pre-processing, it is evident that to produce a model that is exceptionally good at this task will require more work to be done.
All of this said, we still managed to accomplish a lot for

the scope of this task. We have described various hyperparameters and tested the model on them to varying results. Surprisingly, while the model loss does decrease over time in all hyperparameter settings, it is found that only in some cases are the evaluation metrics like the BiLingual Evaluation Understudy scores and the Perplexity scores are improving over time. The reason we do not have training and validation loss plots or BLEU score and Perplexity score plots over epochs is because the training is done via CodeXGLUE's run.py file and it displays these values but does not allow easy storage of them to let us perform plots. However, the .ipynb file does have these values for each epoch.

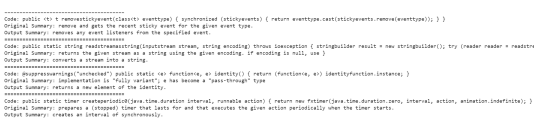

Figure 12: Model Outputs with 100 epochs

Parameters:

```
lang = 'java'
lr = 5e-5
batch_size = 16
beam_size = 20
source_l = 256
target_l = max_comment_len
data_dir = '.'
output_dir = f'model/{lang}'
train_file = f'{data_dir}/{lang}/train.jsonl'
dev_file = f'{data_dir}/{lang}/valid.jsonl'
epochs = 100
model = 'microsoft/codebert-base'
```

Figure 13: Parameters with 100 epochs

However, unfortunately, even with those settings, the most that the BLEU score reaches on a scale of 0-80, where a score of above 50 is usually considered human level or better, the best score that we are able to achieve is 21.263. And upon further study of the output, we can find that the model has mostly relied on the method name itself to try and predict the summary. While not a bad strategy, it certainly leaves a lot to be desired since the model is ideally something that should be able to read and interpret actual code.

6 BROADER IMPACTS

The Software Development sector has a lot of small inconsistencies and factors which make the overall Software Development LifeCycle of product delivery slower. There is a lot of scope to automate relatively tedious tasks in the Software Industry so that organizations can motivate employers to focus on the actual logic, instead of having to worry a lot about making the code follow some guidelines which make it readable. This can hinder the efficiency for programmers and result in either or both, inefficient code and subpar summaries. On the other hand, not having to worry about the peripheral tasks would be a great advantage as it would let programmers focus on actual programming, instead of doing documentation work. Not only will this be helpful in the software development industry, the education industry can benefit a lot from this application as educators can give well-commented code to students which will still make sense after reviewing it after a long time.

7 CONCLUSION

I would like to conclude by saying that this was a great learning experience and gave deep knowledge about deep learning applications in the software engineering domain. We successfully created a model which inputs code snippets of data and outputs a string describing the code.
The implementation was able to provide some very basic summaries for code inputs, but did not make sense for other inputs. On average, the name of the function was the primary way for the model to determine its function. As such, well written functions tended to have well output summaries.
Overall, the implementation of model was according to the scope which was modified from the project proposal keeping in mind the reviews regarding evaluation metrics, as well as the fact that the group size was reduced to one around two weeks before the project execution.

8 AUTHOR CONTRIBUTION

The author of this paper is "Ashwin Shenolikar". The author confirms sole responsibility for the following: Project report including model description and evaluation, results and conclusions, and Project slides. Being the single author, entire project contribution belongs to him.
To avoid discrepencies, it is noted that the project proposal was submitted by two authors: 'Ashwin Shenolikar' and 'Prachi Pundir'. Later, the course was dropped by the second author, and hence this report is a solo project report.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. https://doi.org/10.48550/ARXIV.2005.00653

[2] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. https://doi.org/10.48550/ARXIV.2003.10555

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://doi.org/10.48550/ARXIV.1810.04805

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. https://doi.org/10.48550/ARXIV.2002.08155

[5] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. https://doi.org/10.48550/ARXIV.1909.09436

[6] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 2073–2083. https://doi.org/10.18653/v1/P16-1195

[7] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. https://doi.org/10.48550/ARXIV.2004.02843

[8] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. https://doi.org/10.48550/ARXIV.1907.11692

[9] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. https://doi.org/10.48550/ARXIV.2102.04664

[10] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic Source Code Summarization with Extended Tree-LSTM. https://doi.org/10.48550/ARXIV.1906.08094

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. https://doi.org/10.48550/ARXIV.1706.03762

[12] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. https://doi.org/10.48550/ARXIV.1910.03771