Code Summarization: Generating Summary of Code Snippets

Shreya. R. Mehta[1*], Sneha. S. Patil[1], Nikita. S. Shirguppi[1,] Dr. Mrs. Vahida Attar[2]

[1]Department of Computer Engineering, College of Engineering, Pune India.

[2]Head of Computer and IT Department, College of Engineering, Pune India.

*Corresponding author

doi: https://doi.org/10.21467/proceedings.114.47

Abstract

Source Code Summarization refers to the task of creating understandable natural language summaries from a given code snippet. Good-quality and precise source code summaries are needed by numerous companies for a platitude of reasons - training for newly joined employees, understanding what a newly imported project does, in brief, maintaining precise summaries on the evolution of source code (using git history), categorizing the code, retrieving the code, automatically generating documents, etc. There is a considerable distinction between source code and natural language since source code is organized, has loops, conditions, structures, classes, and so on. Most of the models follow an encoder-decoder structure, we propose an alternative approach that uses UAST(Universal Abstract Syntax Tree) of the source code to generate tokens and then use the Transformer model for a self-attention mechanism which unlike the RNN method is helpful for capturing long-range dependencies. We have considered Java code snippets for generating code summaries.

Keywords: Transformer model; Universal Abstract Syntax Tree; Structural representation; Encoder; Decoder.

1 Introduction

Comprehending code snippets is an important pillar of software development and code maintenance. Developers' efforts are reduced considerably if the natural language summary of the code snippet is available. The task of creating readable and understandable natural language summaries that define the purpose of code is known as source code summarization. Due to the advancements in deep learning techniques, increasing use of neural networks, easy availability of high-volume data via numerous large warehouses, repositories, etc.; developers are devoting a lot of time and resources on automatic code summarization. Seq2seq is a preferred option by many while generating summaries using neural networks. Recurrent Neural Networks using attention mechanisms for combining solo code tokens which are represented with the help of an embedded matrix is one of the earlier methods to generate summaries of code snippets. Later works also include the seq2seq network which deploys recurrent neural networks via attention on different program snippets. As the Recurrent neural network-based models try to model the data sequentially by processing the tokens generated from source code, in a sequential manner; they prove to be inefficient in modeling non-sequential-structural code representation. Another drawback of using the RNN method is that the model is unable to detect long-range dependencies between program tokens which might arise if the length of the code snippet is very long. Unlike Recurrent Neural Networks, LSTM (long short term memory) is able to identify long-range dependencies between code tokens which is very useful in source codes which are very long. LSTM has shown to increase efficiency substantially in many NLP tasks like text generation, summary generation, concept making, story writing, etc. In order to model the relationship between code tokens, it is necessary to learn the order in which the code tokens appear in a sequence. In our work, we show that efficiency in generating source code summary is improved by an incredible degree if the positional encoding is infused utilizing the Transformer model. We show that

Proceedings DOI: 10.21467/proceedings.114; Series: AIJR Proceedings; ISSN: 2582-3922; ISBN: 978-81-947843-8-8

instead of absolute positioning, relative positioning is a better measure and is substantially more efficient for determining code summaries. Our proposed approach is easy, effective, and efficient and results show that it outperforms many state-of-art approaches that use deep neural networks (RNN, CNN, Code-NN, etc.) Our basis for evaluating our Transformer model, which uses pairwise relative positioning of code tokens are 3 matrices - BLEU, METEOR, and ROUGE-L.

2 Literature Review

We have studied many research papers on code summarization and the models used in them are discussed in brief below:

2.1 Automatic Source Code Summarization with Extended Tree-LSTM

Source code contains structural features like loop, branching, conditions etc. These structural features of the source code are captured by Abstract syntax trees (ASTs). As a result, ASTs play an important role in machine learning studies. This paper explains how Tree base LSTM which is a generalization of tree structured data isn't sufficient as it is unable to handle ASTs with nodes with arbitrary number of children. To handle this issue, the paper proposes an extension to tree LSTM – Multi way tree LSTM, which can handle arbitrary numbers of children to a node in ASTs. This approach uses 3 basic components – the encoder, the decoder and attention mechanism. The proposed approach achieved better results than several state-of-art techniques.

2.2 Summarizing Source Code using a Neural Attention Model

Code-NN uses Long Short Term Memory (LSTM) model with attention mechanism to produce natural language summaries of C sharp programs and MySQL queries. The corpus is generated by scrapping code from stack Overflow website, each time the model is trained.

Task Definitions: CODE-NN produces a Natural Language summary of C sharp code snippets (*GENERATION task*). CODE-NN also performs the inverse task to retrieve source code given a question in NL (*RETRIEVAL task*).

Consider UC to be the set of all code snippets and UN to be the set of all summaries in NL. For training the corpus with J number of code snippets and summary pairs (cj , nj), $1 \leq j \leq J$, $cj \in UC$, $nj \in UN$, it defines the following two tasks:

GENERATION: For a given program snippet 'c' $\in$ UC, the task is to generate an NL sentence $n* \in UN$ that maximizes the chosen score function *'s'* $\in (UC \times UN \rightarrow R)$: $n = argmax\ n\ s(c, n)$

RETRIEVAL: The approach also uses the same score function 's' to retrieve the highest-scoring code snippet $c * j$ from the training corpus, given an NL query $n \in UN$: $c * j = argmax\ cj\ s(cj, n)$, $1 \leq j \leq J$. Here, j lies between I and J, both including.

The accuracy of this approach is as follows:

Error	% cases
Correct	37
Redundant	26
Missing Information	24
Out of context	13

Loopholes:

1. Only the lexical approach was taken into consideration.
2. High Accuracy could not be achieved.

2.3 TASSAL: Automatic Source Code Summarization

This method aims to automatically create a snapshot of each source code snippet in the repository by wrapping its regions of code below. The latest code editors have code wrapping features to specifically select encoding code-blocks, it is not possible to use as wrap decisions to be made manually or according to simple rules. TASSAL Algorithm created by using tree tracing is based on building a similarity between the source code summary and the source code. This paper is the first content-based wrap to summarize the code. TASSAL Framework: It acts as a compilation of a set of source files and the required compression rate and the summary of the source code is given as output, where the source code itself is the input. Steps: 1. Sets the AST of the code to find a region that is suitable for wrapping. 2. A source code language model is used for each region that overlaps. 3. Identify the entire source file, which directly shows the code, as opposed to special projects or Java-generic tokens that have little to do with file comprehension. 4. Determines regions that cannot be changed to fold while achieving the required level of size reduction.

2.4 CODE2SEQ: Generate sentences from structured representations of code

Sequence-to-sequence (seq2seq) models, used in neural machine translation (NMT), utilize state-of-the-art functionality of the functions by considering source snippets as a string of tokens. This model uses an alternate method that provides the architecture of programming languages to improve source code. This model represents a code line as a set of overlapping paths in its abstract tree (AST) and utilizes attention mechanisms to select the appropriate paths during the opening process. Representing a source code in AST:

1. AST is a unique identifier for source code written in any language.
2. User-defined values that indicate identifiers and names from the source snippet are used to refer to the leaf edges of the AST.
3. Terminals (non-leaf nodes) represent a limited set of structures in a language, e.g. constraints, expressions, and declarations.

2.5 Improving Automatic Source Code Summarization via Deep Reinforcement Learning

Most of the models include the encoder-decoder framework in which the function of the encoder is to encode the code into a hidden space and then the decoder decodes into natural language space. However, it has two major drawbacks:

a) The encoder they use, majorly uses the sequence to sequence model and therefore only the sequential content of the code is under consideration and as a result, the tokens' relationship in the snippet which should be captured by the AST structure and which is important for the task of code summary generation is ignored here.

b) Their decoders are just trained to predict the next token with the previous token given. However, during test time it is expected to generate the entire sequence from scratch. In this paper, the BLEU metric is used to train both networks.

A more comprehensive approach is used in this paper which includes usage of two LSTMS; one AST-based LSTM for the structure of source code and another LSTM for the sequential content of the source code. These 2 representations are combined using a mixture of attention mechanisms (hybrid model). So the model works in lexical as well as syntactic level. Comments in the code are also considered as a part of syntactic analysis. For summarization, deep reinforcement learning is used. The restriction confronted is to structure a reward function to measure the value of action correctly which is as yet an open issue. The danger to legitimacy is generally on the metrics which we chose for evaluation. For tasks such as neural machine translation, image captioning, etc. it is a tough task to evaluate the resemblance between two

sentences. Also, the performance based on human evaluation gives more perspectives that have to be considered.

3 System Requirement Specification

Following software and hardware specifications are preferable for our model to run efficiently and without any errors.

A. Software Specifications

The following are the software specifications required:

- *Python 3+* - Python is an interpreted, high-level, regular-purpose programming language. Version 3+ is preferable.
- *Numpy* - NumPy is a python library used for programming. It has additional support for large, multi-dimensional arrays and matrices. Also, it has a huge collection of high precision mathematical methods used for operations on these arrays.
- *Torch 1.3+* **-** The torch is a python based computing package that provides many useful features. The features we use for our work are as follows:- 1. Computational tensor (like NumPy) with strong GPU acceleration and Deep Neural Networks on a tape-based autograd system which provides maximum flexibility and speed.
- *Tqdm* - Tqdm is a python library having progress bar features. It has good support for loops like 'for', 'while' etc which are nested. It can be used in Jupyter/IPython notebooks.
- *PrettyTable* - PrettyTable is another Python library that supports features like tabular data in visually alluring ASCII tables in a quick and simple manner.

B. Hardware Specifications

The following hardware system specifications are required:

- Ubuntu 12.04 or higher
- 16 Gb RAM system memory
- Disk space of at least 10 Gb

C. Assumptions and Dependencies

The following assumptions and dependencies need to be satisfied:

- Hardware is in proper working condition
- Latest and updated OS is used
- Python libraries are installed prehand

4 Proposed Approach

We propose to implement an LSTM (long short term memory) based model for generating summaries of input programs. Our dataset consists of java code snippets and we aim to expand our language scope to other languages like python, CPP, etc. The input source code is used to construct a universal abstract syntax tree, tokens are then extracted from the UAST and given to the Transformer LSTM model. The code summary generated is also a sequence of tokens. X = (x1, x2, ..., xn). A brief description of how to model the pairwise relative positioning between tokens to generate a code summary is described in the coming sections.

5 Architecture

The Transformer model consists of an encoder and a decoder, both having stacked multi-headed attention mechanisms. The encoder and the decoder are also equipped with linear transformation layers which are

Proceedings DOI: 10.21467/proceedings.114
ISBN: 978-81-947843-8-8

Series: AIJR Proceedings
ISSN: 2582-3922

parameterized. At each layer, the self-attention mechanism is carried out, and 'K' attention heads are utilized by the multi-head attention model. We have chosen the Encoder-Decoder transformer model as it can solve versatile problems of sequence to sequence models. As the number of input tokens, as well as the number of output tokens, vary from code to code, it is important to address the same.

6 Transformer

The transformer model converts a sequence of tokens into another sequence but does not use recurrent neural networks like RNN or CNN. It can be described using 2 blocks - the encoder block and the decoder block. Both the encoder and the decoder models are so designed that stacking them on top of each other is feasible. Multi headed attention and feedforward networks are the major components. As direct strings cannot be used, the input and output sequence of tokens are first embedded in an n-dimensional space. As the transformer model is unable to store the absolute positions of the input tokens, we have chosen to track the relative positions of the tokens and learn the relationship between them. Consider Q to be the query matrix, and K be the vector representations of all tokens in the sequence. Let V be the set of values which are vector representation and let 'a' be attention weights which are defined by:

$$a = softmax(QK^T/\sqrt{dk})$$

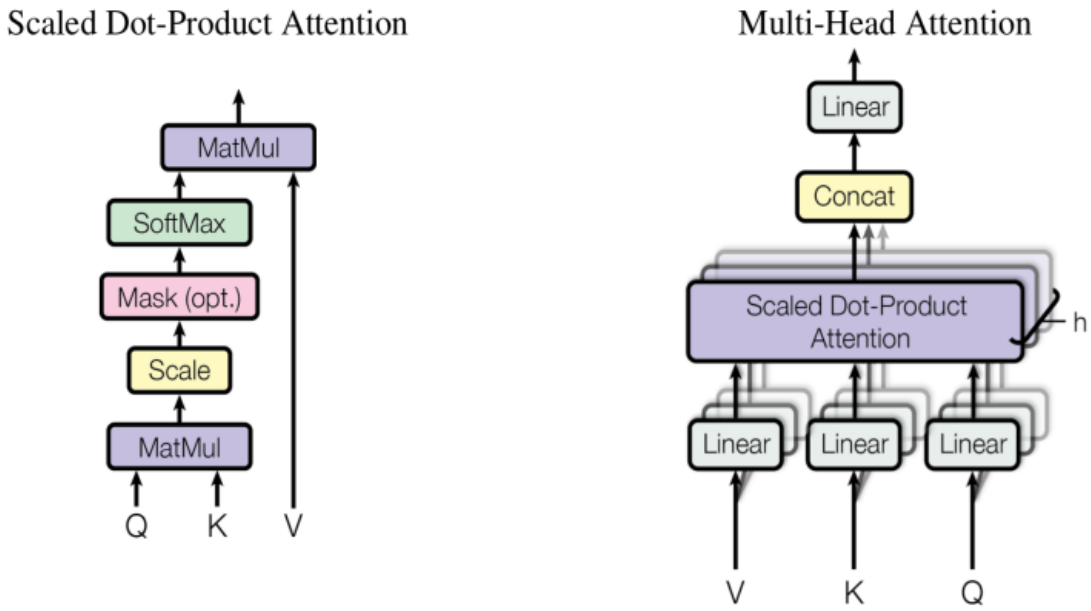

Figure 1. Attention Models

The softmax function applied to the weights 'a' varies in the range [0, 1] and also has distribution in the same range. All the tokens in the sequence have the weights then applied to them. Depending upon the positions of the above matrices in the attention model, their values change. The positions include at the encoder, at the decoder, and in between the encoder and the decoder ie them attention. The multi-head attention model attaches the encoder model to the decoder model. A feed-forward point-wise neural network follows the multi-head attention model. This feed-forward network is present in both, the encoder as well as the decoder and, it has similar parameters for each position. These params consist of a separate identical linear transformation of each sequential token.

7 Encoder

Reads a sequence of input tokens, and encodes them into a constant-length vector. The encoder apprehends both, the semantic and the syntactic structure of the snippet. The encoder level learns the relationship between the input set of tokens and develops an internal representation of these relationships. Let x = (x1, x2, ..xn) be the set of input paths. Let yi be the vector representation of each path - yi = (v1, v2, …,. vn). Each path is represented distinctly via a bidirectional transformer which is used for capturing the compositional nature of the end values. Any UAST path is characterized by terminals which are its first and last nodes. The values of these terminals are the tokens of the code snippet. Code tokens are further split into sub-code tokens, using camel-case split, snake case split, etc. The Long short term decoder can also be used for predicting sub tokens.

8 Decoder

For n paths, the average of the combined results of all n paths is considered for providing the decoder with an initial stage. Unlike the usual Encode-Decode models, the sequence of the input paths which are random is not considered. Hence, the code snippet is represented not as a list but as a set of random vectors.

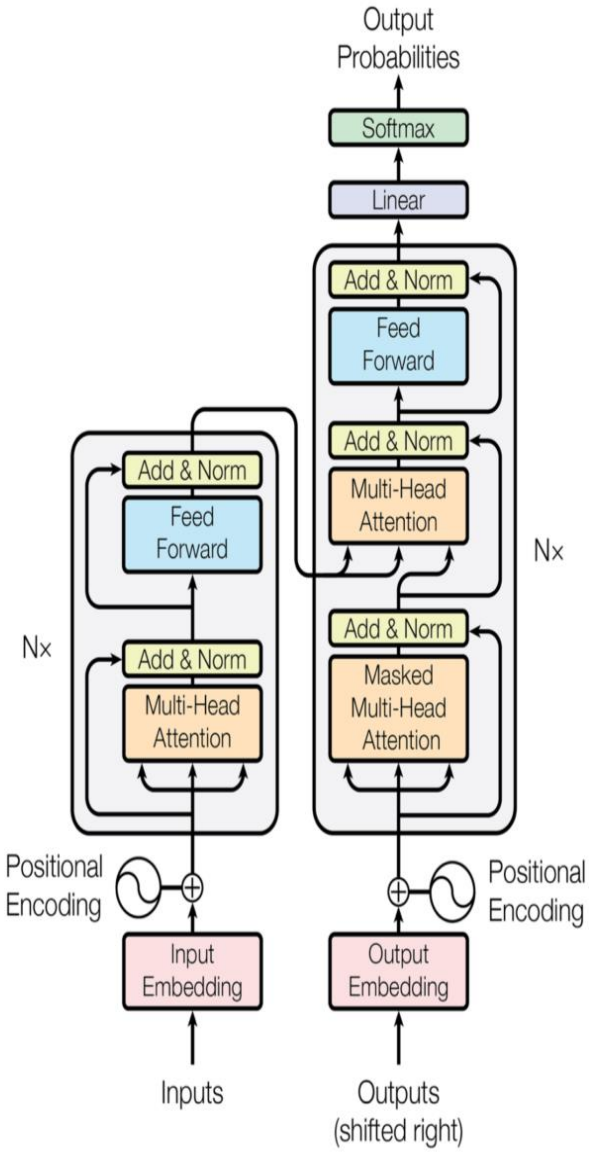

Figure 2. Transformer Model

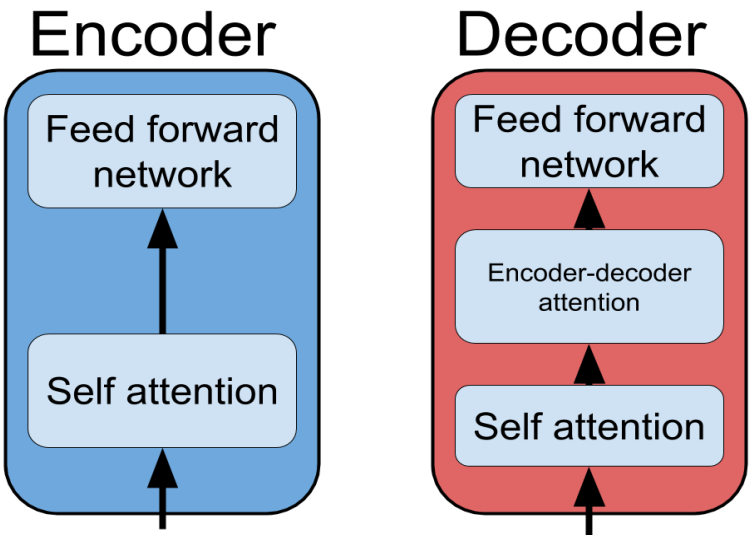

Figure 3. Encoder and Decoder

9 Self Attention

In this mechanism, every head computes sequential input tokens which are provided as input to the Transformer model which in turn outputs a sequential set of output tokens. Here x = (x1, x2, ..., xn) are the input code sequential tokens and O = (O1, O2, ..., On) are the output summary sequential tokens. Also, V^P, V^W, V^Q are unique parameters for each layer in the encoder, decoder, attention mechanism respectively. $Oi = \sum_{i=1}^{\infty} \beta ij\,(xj\,V^W)$ $eij = xi\,V^Q\,(xj\,V^P)^T / \sqrt{dp}$

All the output vectors are calculated using the above-given formula.

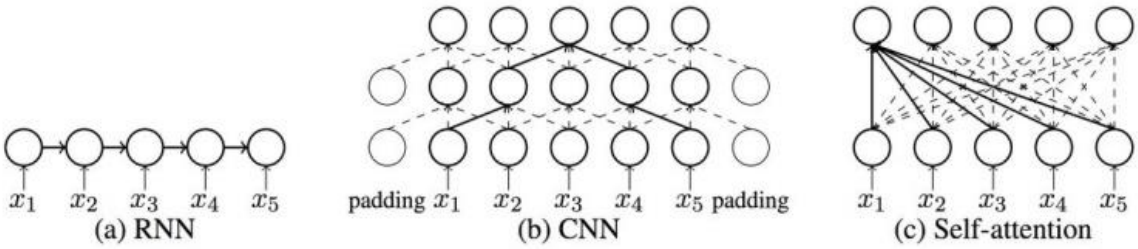

Figure 4. Comparison between Models

10 Copy attention

In our approach, the output sequence of vectors has tokens that are generated from both, the input set of tokens as well as extracting words from the vocabulary. We implement an extra layer of attention which is used for learning copy distribution on the stack head. This attention model allows the Transformer to copy tokens like method names, variable names, global variables from the code snippet and therefore enhance the summary generation considerably.

Dataset	Java
Train	68,238
Validation	9,231
Test	9,231
Unique tokens in code	66,560
Unique tokens in summary	45,998
Average tokens in code	123.24
Average tokens in summary	18.38

11 Coordinate Representation

In this section, the focus is on modeling source code vectors and modeling relationships between any pair of tokens.

11.1 Encoding Actual Distance

Our transformer model utilizes the ordering of the program snippets' tokens and an embedded matrix is trained using the same. This matrix is used for learning the absolute positions of the encoding vectors. We have shown that calculating absolute positions of the tokens have not proven useful and the accuracy is limited as well. We have also trained another embedded matrix, which is trained correspondingly to learn the actual/absolute distance between the summary vectors.

11.2 Encoding Relative Positions

We observed that the semantic representation of a program snippet does not only depend on the absolute positioning of vectors, but it more or less depends on the interaction between them. The interpretation of the source code gets influenced by these mutual/pairwise interactions. For example, x+y and y+x mean the same. It has been observed that, if the maximum relative position is clipped, then the positional information precision is limited. In our approach, we have not considered directional information obtained from tokens. So for 2 tokens A and B, the fact that A lies to the left/right of B in sequential order are not considered. The self-attention mechanism can be further expanded to encode the mutual relationship between the input tokens as follows:

$$oi = \sum_{j=1}^{n} (\beta ij\ (xj\ V^{W} + bij^{W}))$$

$$eij = xi\ V^{Q}\ (xj\ V^{K} + bij^{K})^{T} / \sqrt{dp}$$

12 Experimentation

Experimentation of our project is as follows:

12.1 Setup

12.1.1 Preprocessing the data

We have used a java dataset. The scope of our project can be expanded to other languages as well, as we are using the UASTs of the source code to generate summaries. Preprocessing is carried out on the generated code tokens. The preprocessing step includes camel case splitting and snake case splitting carried

out on the tokens obtained as the output of UAST. The accuracy and performance of the code summary increase significantly if preprocessing related to splitting is carried out.

Camel Case Splitting: Given a string with/without camel case letters, this split divides the string into substrings and splits it on camel case letters. Eg. AbstractClassWord is split into [Abstract, Class, word].

Snake Case Splitting: Snake casing is a method of writing a collection of words intended to be joined by space, to be joined by an underscore(_). A typical example of the same is Abstract_class_word. Snake case split splits the word on the underscore, (removes the underscore) and returns a list of sub tokens. Eg. Abstract.class.word is split into [Abstract, class, word].

12.1.2 Metrics

We use 3 different matrices to evaluate the accuracy of our summary generation model. These matrices are ROUGE-L, METEOR, and BLEU. We have also used tf-idf in initial stages.

Rouge-L: The full form of ROUGE is Recall Oriented Understudy for Gisting Evaluation. ROUGE itself is a predefined set of matrices used to measure the performance of automatic summary generation models. A dataset of reference summaries is compared with the automatically generated summaries (which are machine produced). Recall and precision are used for calculating the overlap of N-grams between both the summaries.

Recall: (number of overlapping words)/ (total number of words in the human-generated summary)

Precision: (number of overlapping words)/ (total number of words in the summary generated by Transformer model)

Meteor: Meteor is an automatic evaluating measure that measures the performance of system generated summaries with a set of human-generated reference summaries. System and segment generated values of the meteor are evaluated on the basis of alignments between hypothesis reference pairs. This matrice also includes paraphrasing tools and X-rays.

Bleu: BLEU stands for bilingual Evaluation Understudy. It is an algorithm used to measure the performance of text translation by machine from one natural language to another. The output of this matrix is always a floating-point between 0 and 1. Values for individually calculated segments of lines are compared against reference human-generated sentences, using precision as the major evaluator.

12.1.3 Inception

We have compared our transformer-based approach with existing models and have arrived at the conclusion that our model performs better in most of the scenarios in comparison to other state-of-art existing methods.

12.1.4 Hyper Parameters

Vocabulary size and the length of the program snippet, both are kept substantially large, and experiments are carried out. The transformer model is trained using Adam optimizer with a grasping rate of 0.0001 initially. The batch size is kept small (32) initially and the dropout rate is set to 0.2. The model is trained for a maximum of 150 epochs and if the validation performance does not enhance for 20 sequential iterations, then the training is stopped at an earlier stage.

13 Analysis of Results

It is observed that, if the model is trained without camel case splitting and snake case splitting, there is a decrease in overall performance by 0.6 (BLEU) and 0.72 (ROUGE-L). Implementation of the copy attention model enhanced the overall accuracy by 0.44 (BLEU) for data-set of java programs.

1. Representation of positions of the tokens also has a substantial difference when absolute positions are compared to relative positions. The model which learns the absolute position of vectors performs slightly badly when compared to the model which learns the relative positions of all pairs of tokens. Along with this, K- the clip distance is also varied in the experimentation and the results are noted. Directional information is proved to be valuable. Also, distances of the format 2^i where i greater than 1 are seen to have comparable performance.
2. Variation in the size of the model as well as changing the number of layers in attention, encoder and decoder, it is observed that a deeper model with more layers has greater performance than a model with lesser layer i.e. a wider model.
3. Another additional set of experiments was carried out by using/not using UASTs. Input to the UAST generator is the source code. The output is a sequence of tokens which is the input to the Transformer model. It was observed that performance increases when the UAST module is used instead of directly extracting tokens from the source code. $O(n * d^2)$ is the transformer complexity where n is the length of the input vector. This implies that using UAST comes with additional costs.
4. Analyzing the model qualitatively: Adam optimizer model is compared with the Vanilla optimizer model and the results of the experimentation are recorded. The self-attention model with copy feature enabled help in generating shorter and briefer summaries as the unique and rare words are taken into consideration. Also, tokens that occur frequently in the source code get a greater copy probability when relative positions are implemented as against absolute positioning. We conclude that this is due to the flexible learning of the relation between i/p vectors in accordance with only their relative positions.

We tried varying the size of the model and recorded the matrix values for each model size. The results are recorded in the table given below. It can be observed that for the model size 768, the values for all 3 matrices are relatively higher.

Model size	**BLUE**	**METEOR**	**ROUGE-L**
256	15.7	21.52	48.62
384	28.3	24.51	51.41
512	44.2	25.92	52.73
768	85.2	27.65	5.41

Table 3. Study on the hidden size for our model on the Java dataset

We also tried varying the number of layers of the model and noted the corresponding values of METEOR, ROUGE-L, and BLEU. We interpreted that as the number of layers goes on increasing, the summary generated by the model turns out to be more precise. Deeper the model, the better its performance.

Proceedings DOI: 10.21467/proceedings.114
ISBN: 978-81-947843-8-8

Series: AIJR Proceedings
ISSN: 2582-3922

Number of layers	BLUE	METEOR	ROUGE-L
3	22.2	41.27	51.38
6	44.1	3.42	52.73
9	66.3	45.30	54.04
12	88.4	45.65	54.88

Table 4. Study on the number of layers for our model on the Java dataset

Another experimentation involved varying the Relative distance (p). The ablation studies were performed taking both directional and nondirectional aspects. The table given below shows the values recorded. It was observed that, for p = 32 and the directional model, the scores recorded were highest.

Relative Distance (p)	Directional	BLUE	ROUGE-L	METEOR
8	Yes	44.21	26.36	53.87
8	No	42.61	24.66	51.12
16	Yes	44.15	26.35	53.94
16	No	44.07	26.32	53.52
32	Yes	44.54	26.67	54.32
32	No	43.96	26.29	53.26

Table 5. Study on relative positional representations (in encoding) for Transformer

14 Conclusion

We have demonstrated that the transformer model is better suited for source code summarization. We have proclaimed that in comparison to other existing state-of-art models, our model which consists of a self-attention mechanism along with a copy of the attention model has a substantial performance. We have also shown that the Transformer model provides comparable performance when compared to LSTM, RNN models. Comparing the ROUGE-L, METEOR, and BLEU matrices of different approaches, we have shown that the Transformer model can be used for parallel computing as well. Hence, our model can be trained on CPU, GPU as well as using parallel computing. Our model is generic for many programming languages like java and python as it computes a UAST of the source code before training the dataset with the Transformer. In our future scope, we consider expanding to our languages like python, CPP, etc. We also want to apply techniques in other s/w token generation tasks which will include git/bitbucket commits of source code. We calculated the matrice scores (ROUGE-L, BLEU, and METEOR) for other approaches and compared our model with them. As it can be observed in the given table, our model outperforms most other models in all the 3 measures. It has a score of 44.61 for BLEU, 26.42 for METEOR, and 54.77 for

ROUGE-L. Hence, we show that the transformer model, with self-attention and copy attention, provides substantially better source code summaries in comparison to other state of art methods.

Approach	BLUE	METEOR	ROUG-L
Code-NN	27.4	12.72	41.01
Tree2Seq	38.1	22.61	51.62
RL+Hyb2Seq	39.71	22.74	51.93
Deep Com	40.8	23.12	52.76
API+Code	41.2	23.81	52.23
Dual Model	42.41	25.73	53.62
Our Model	42.61	26.42	54.77

Table 6. Comparison of our proposed model with other baseline models

15 Future SCOPE

Nearly all the neural source code summarization perspectives follow the problem in a sequence to sequence model and hence results in sequence generation tasks and the building blocks of the model are mostly recurrent encoder-decoder networks with attention mechanisms. Recent works in code summarization have considered the structured and syntactic representation in the form of Abstract Syntax Trees (ASTs) through encoding techniques that use tree structure encoders like LST, transformer model, Graph Neural Network, etc. A structure-Based Traversal approach is also proposed. In our future work, we want to study the software engineering sequence generation tasks (like including commit messages from the git repositories, comments which are written in order to understand the code more efficiently, etc. in our summary) and apply them to our code structure in the Transformer.

References

[1] Shido, Yusuke, et al. ”Automatic Source Code Summarization with Extended Tree-LSTM.” 2019 International Joint Conference on Neural Networks (IJCNN). IEEE, 2019.

[2] Iyer, Srinivasan, et al. "Summarizing source code using a neural attention model." Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2016.

[3] Alon, Uri, et al. "code2seq: Generating sequences from structured representations of code." arXiv preprint arXiv:1808.01400 (2018).

[4] Cortés-Coy, Luis Fernando, et al. "On automatically generating commit messages via summarization of source code changes." 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. IEEE, 2014.

[5] Haiduc, Sonia, et al. "On the use of automated text summarization techniques for summarizing source code." 2010 17th Working Conference on Reverse Engineering. IEEE, 2010.

[6] Allamanis, Miltiadis, Hao Peng, and Charles Sutton. "A convolutional attention network for extreme summarization of source code." International conference on machine learning. 2016.

[7] LeClair, Alexander, et al. "Improved code summarization via a graph neural network." arXiv preprint arXiv:2004.02843 (2020).

[8] Ye, Wei, et al. "Leveraging Code Generation to Improve Code Retrieval and Summarization via Dual Learning." Proceedings of The Web Conference 2020. 2020.

[9] Wei, Bolin, et al. "Code Generation as a Dual Task of Code Summarization." Advances in Neural Information Processing Systems. 2019.

[10] LeClair, Alexander, and Collin McMillan. "Recommendations for Datasets for Source Code Summarization." arXiv preprint arXiv:1904.02660 (2019).

[11] Moore, Jessica, Ben Gelman, and David Slater. "A Convolutional Neural Network for Language-Agnostic Source Code Summarization." arXiv preprint arXiv:1904.00805 (2019).

[12] Wan, Yao, et al. "Improving automatic source code summarization via deep reinforcement learning." Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018.

[13] Cortés-Coy, Luis Fernando, et al. "On automatically generating commit messages via summarization of source code changes." 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. IEEE, 2014.14

[14] Sulír, Matúš, and Jaroslav Porubän. "Generating method documentation using concrete values from executions." 6th Symposium on Languages, Applications and Technologies (SLATE 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[15] Karampatsis, Rafael-Michael, and Charles Sutton. "Maybe deep neural networks are the best choice for modeling source code." arXiv preprint arXiv:1903.05734 (2019).

[16] Rodeghero, Paige, and Collin McMillan. "Detecting Important Terms in Source Code for Program Comprehension." Proceedings of the 52nd Hawaii International Conference on System Sciences. 2019.