Automated Code Summarization of Program Subroutines Using Deep Learning Technologies

MSc Research Project
Data Analytics

Pramod Belur Ramesh
Student ID: 19211015

School of Computing
National College of Ireland

Supervisor: Dr. Athanasios Staikopoulos

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Pramod Belur Ramesh
Student ID:	19211015
Programme:	Data Analytics
Year:	2021
Module:	MSc Research Project
Supervisor:	Dr. Athanasios Staikopoulos
Submission Due Date:	31/01/2022
Project Title:	Automated Code Summarization of Program Subroutines Using Deep Learning Technologies
Word Count:	6881
Page Count:	20

Signature:	
Date:	31st January 2022

Contents

1 Introduction **1**

1.1 Scope and Assumptions . . . 3

1.2 Research Questions . . . 3

2 Related Work **3**

2.1 Source Code Summarization . . . 3

2.2 Graph Neural Network . . . 5

2.3 Evaluation Metrics . . . 6

3 Methodology **7**

3.1 Research Formulation . . . 7

3.2 Surveying relevant literature . . . 7

3.3 Data Analytics . . . 7

3.4 Evaluation . . . 10

4 Design Specification **12**

5 Implementation **13**

6 Model Evaluation **14**

6.1 BiLSTM . . . 14

6.2 BiLSTM + GNN . . . 15

6.2.1 Human Evaluation of context capturing . . . 15

7 Discussion **16**

8 Conclusion and Future Work **17**

Automated Code Summarization of Program Subroutines Using Deep Learning Technologies

Pramod Belur Ramesh
19211015

Abstract

Software Engineering field, now, is enjoying a burst of data-driven research as there is an interest in code generation, code optimization, code summarization, code understanding etc. Automated source code summarization is generating succint source code summaries without human intervention by learning code sequences. Automated way of generating source code summaries by leveraging data and deep learning techniques to reduce the burden on programmers and bring about faster and better knowledge management in the software engineering industry is purpose of this reaseach. The approach of building models that understand programming languages and summarizes them in English has far-reaching impacts which will bring about democratizing programming to even non-programmers.

In our approach source code of 2.1 million java code-comment pairs was used to train deep learning models to generate summaries, traditional encoder-decoder models were conflated with Graph Neural Network (GNN) and its impact was measured against a model with a traditional sequence to sequence model. The GNN model outperformed the sequence to sequence in all the evaluation metrics of BLEU and ROUGE by at least 1%, which means that the overlap between a human-written reference summary and a model generated summary is more with GNN than without GNN.

1 Introduction

Software Engineering (SE) and Information Technology (IT) have been used in almost all walks of life to solve much of the world's problems. The use of software solutions therefore has increased the amount of software code that is written. There are many repositories that are being maintained by hundreds of engineers, having tens of thousands of lines of code. Code commenting is one of the best practices that is prescribed by various stalwarts of the field but unfortunately programmers are notoriously infamous for pushing the task of code commenting and documentation to the last in their task list (de Souza et al.; 2005). This has proven to be a hindrance in maintaining software code and its especially a glaring problem with legacy codebases. Apart from maintaining an ill-commented codebase, it is also expensive in terms of man-hours that are spent by the companies in ramping-up new engineers that join these teams. A piece of code that goes through multiple changes as part of requirement changes end up having the original comment which no longer reflects what the code does. This consequently makes the task of making any further changes to the same snippet of code even more time-consuming as the developer responsible will have to read the code and figure out the program flow. Software applications that need a technological uphaul or a complete migration to cloud or container-based technologies must deal with the inevitable refactoring, and ill-commented codes will only make the whole process slower. All the above will eventually contribute to “technical debt”, an expression borrowed from Financial Debt, to signify the amount of technical work that is required to be done, quantified in monetary (or sometimes in time) terms, for a project to be operational as per best practices. The technical debt of all software projects was estimated to be a whopping 500 Billion US dollars in 2010 and was estimated to double in 5 years’ time (Szykarski; 2012). Lack of knowledge distribution and documentation debt is also known to contribute to technical debt (Slinker; 2013).

To obtain multi-faceted advantages that frees up a programmer’s time, makes his or her life easier, reduces technical debt, decreases the monetary and time resources for ramping up activities of new personnel for them to get acquainted with the codebase, an automated way of code summarization is required. Several heuristic based methods specific to programming languages were developed by Integrated Development Editors (IDEs) such as eclipse, with the advent of JavaDocs, that automatically added comments to accessors and modifiers, “getters and setters”, in java-speak which greatly reduced the burden. However, other types of business logic still required programmer intervention for it to be clearly documented. To this end, multiple approaches are being sought by computer scientists in the past decade.

Code Summarization is a term coined in 2010 (Haiduc et al.; 2010) for the automated human-like succinct code snippets generated automatically. Initially, the task of automatically generating code comments followed a heuristic rule-based approach, where in the comments would be generated based on a rule “look-up” that contained the appropriate comment (McBurney and McMillan; 2015). As the advancement in the field of Natural Language Processing (NLP) happened, it also fostered a similar growth in the field of Natural Machine Translation (NMT), where in 2016, a path breaking encoder-decoder based code summarization tool was developed by Iyer et al. (2016), like NLP. The encoder would take in the code sequence and the decoder would take in the summaries, and finally summaries would be predicted once the outputs of encoder-decoders are processed. The introduction to self-attention mechanism by Vaswani et al. (2017), where in attention was placed on important words while translating them, encouraged a slurry of research into this niche field, which by then also had its name in the world of Big Data called “Big Code”.

Although there are many research methodologies that have tackled this issue, there is still a dearth of research that explores the efficacy of graph-based models in generating summaries of code snippets. The Abstract Syntax Tree (AST) that gets generated during the compilation of the code takes the form of the graph, which intuitively makes one wonder if the graph representation of the code can be leveraged to learn more about the code summaries and establish more context and get closer to human-like summarization. Context is akin to the zooming out of a picture to get more information, in terms of code, it could be the class a method resides in which gives the summarization more meaning. For example a method named “book()” would be summarized as ”books the train” in ”Train” class, while it would be ”books the flight” in ”Flight” class.

Thus, this project aims to generate summaries of code using graph-based modelling techniques, and the generated summaries will be evaluated against universally agreed metrics.

The rest of the paper is structured as follows: Section 2 discusses the related work, Section 3 talks about the methodology used in this project, Section 4 specifies the design decisions undertaken, Section 5 describes the implementation details, Section 6 evaluates and interprets the results of the project, Section 7 discusses the impacts of the findings and Section 8 concludes the paper with mentions of potential future work.

1.1 Scope and Assumptions

Although the problem of technical debt is all encompassing, this project limits its research to Java, and java datasets only, this is because, being in its 17th version, Java continues to be one of the most popular programming languages with over 9 million active users in the world.

A method is a granular module in programming languages, it is sometimes referred to as "subroutine", it is situated inside a class, which hosts multiple other methods. Software engineering code dependencies extend to external libraries to plugins hosted on clouds, thus context can sometimes be between a method and an 3rd party obscure plugin that is used for some logic. Hence it is assumed that "context" is the rest of the methods that are situated along with the method for which the summary is being predicted.

1.2 Research Questions

Being aware of the dearth of research that is being made into graph based models in confluence with encoder-decoder with attention architecture, the following Research Questions (RQ) was specified and will be tackled:

RQ1*: To what extent does the usage of GNNs in code summarization increase the evaluation metric scores?*

RQ2*: To what extent can context be captured by GNN based models?*

2 Related Work

With the advancement of computing methodologies and plethora of open source libraries to aid complex computation problems, the field of code summarization consequently has seen an uptick in the number of research papers that have been published using data-driven approaches. Although heuristic and rule based methodologies were the building blocks in this research area, this paper solely focuses on data-driven research due to its relevance, in specific, this paper takes into account the relevant research that was conducted between 2016-2021. The related works are categorized into 3 subsections, Source Code Summarization 2.1, Graph Neural Networks 2.2, and Evaluation Metrics 2.3

2.1 Source Code Summarization

Iyer et al. (2016) were the first to pioneer the data-driven methodology by using the Natural Machine Translation (NMT) based model into code summarization task, they used encoder-decoder with attention mechanism directly from NMT tasks and utilized the verbose stackoverflow tags and based their model on predicting the next word by using code sequences directly as input. This model, although with its limitations in scalability and the idea of using a corpus word-by-word, and code into one input of an Recurrent

Neural Network (RNN) architecture, served as a good starting point and a baseline model for several models that followed. Alon et al. (2019) proposed the use of embedding space that is widely used in Natural Language Processing (NLP) tasks, they proposed the usage of "code embeddings" in their model called "code2vec", this approach involved the use of Abstract Syntax Tree (AST) of the code and breaking down the edges of the AST and create these code embeddings which was then used to predict code semantics and predict the name of the method given its embeddings. Hu, Li, Xia, Lo and Jin (2018) also suggested the use of AST in order to learn the hidden representations of a code snippet, the proposed a unique way to flatten the AST by developing a score-based traversal that gave importance to lexicons in the AST and called it Structure Based Traversal (SBT). These models all produced promising results and are considered baselines in this area. Apart from using AST to extract structural information, there have been researches that have delved into looking at API for information (Hu, Li, Xia, Lo, Lu and Jin; 2018).

Code and code dependencies by virtue have lots of interlinked dependencies, AST representation of codebases which were very well curated were also found to be extremely hard to train with lots of parameters to take into consideration, this problem was mentioned and tackled by Lin et al. (2021), a better approach was to do block-wise splitting of AST, called BASTS, which was then fed into the transformer, although this paper exceeds the baselines that were mentioned, the steps to reproduce the splitting of AST into block-wise ASTs were hard to follow, and in datasets that have large ASTs, the breaking down of individual block-wise ASTs could be hard in programming languages that are not verbose and type heavy.In addition to using ASTs, method name was mined for semantics and combined with its summary to learn more insights for summarization by research conducted by Haque et al. (2021). Their nomenclature for this mined relationship was 'action-word', this action-word was sent to encoder-decoder with attention, however, methods that are badly named will break-down the premise of this idea hence it may not generalize well. Liu and Wang (2020) built on top of Haque et al. (2021) by using method name to derive insights along with AST, but they also added a key-word enhancer that was used as a separate embedding layer and as an input to the encoder, similar to the work done by LeClair et al. (2019a), this methodology added attention and weights to each of the enhancers in form of a weighted vector, the use of separate embedded space for key-word enhancer increased their metrics by 4% but the training time was high due to the high dimensional spaces used. Research by Xie et al. (2021) also uses the same mechanism of combining the code method name but adds a mechanism for methods that do not have meaningful method names, this mechanism entails the generation of method names, which would subsequently help in the summarization. They added a novel prediction mechanism that was fed into the encoder. The cross-validation needed for this prediction is skipped which runs the risk of not being able to generalize hence this research defeats its purpose while predicting method names.

Zhang et al. (2020) developed rencos, a baseline model that is prevalent for its robustness. This research addresses the problem of encoder-decoder mechanism missing out on words that might be important but are ignored due to their low occurrence. Rencos combined the Information Retrieval (IR) and NMT ways of appraoching this problem by hiring Amazon Mechanical Turks to highlight important low frequency words, these words were then used in their model as a separate vector space in their encoder with seperate attention mechanism. This research was useful in summarizing code that did not contain meaningful summary in the first place. It increased BLEU (Papineni et al.; 2002) and ROUGE (Lin; 2005) by 3% and exceeded all the baseline models.

LeClair et al. (2019a) came up with an approach of separating the code and code semantics, learnt from AST as two separate inputs into two different encoders, the decoders were fed the summaries, and the ultimate output of the above were given to the attention mechanism which placed attention to the important words in the summaries. This work also curated the dataset, which is popularly known as ”Java 2.1m”, which contained 2.1 million java methods, tokenized as well as raw, with auto-generated comments removed, filtered and manually checked.

Semantics and insights drawn from the surroundings of the code snippet or subroutine termed as ’context’ was first taken up by Haque et al. (2020) pre-training a model that contained ’context’, comprising of all the methods. This pre-trained context is added as an extra parameter along with code, text and AST embeddings, this work was built on top of LeClair et al. (2019a). The context here was built by removing the code that was added as an input and the rest of the methods were built as a n x m matrix, where n is the number of methods in the file, m is the size of the sequences, these hyperparameters were tuned in their pre-training step. This approach increased the evaluation metrics of BLEU (Papineni et al.; 2002) and METEOR (Banerjee and Lavie; 2005). The use of bi-directional Long Short Term Memory (LSTM) along with the context as an input scored higher than all other models. The context creation and building, although was a novel approach, was static in nature and the number of methods in the nxm matrix contributed to the overall summarizing of the code only when n and m were high in number thereby increasing complexity. The visualization of attention gave an insight into the explain-ability of the model.

The use of ’global’ context and ’local’ context which is in tune with the variablemethod scope where in the variables also interact with their own value and scope along with the rest of the project. The rest of the code might reside either in the same file or in the same project or in a separate library. Bansal et al. (2021) used this idea and conflated this with the existing encoder-decoder architecture, they used project context and file context as part of their embeddings and the number of methods in a file and the number of files as hyper-parameters, this methodology also achieved promising results and it was significant in its usage of two real-world contexts, however as with all higher dimensional models, this model tends to be extremely hard to train and it might take higher training time as the number of files/methods increase. The research by Zügner et al. (2021) also came up with a language agnostic model by combining the tokenized code sequences and built a tree with code dependencies, their approach used the transformer approach using pre-trained ASTs. The paper scores well on all the evaluation metrics but the baselines do not consider the latest models that are established in this area. All of these methodologies that rely on context use NMT metrics to plot and publish their results, which might not be one-size-fits-all approach.

Thus, the use of valid dataset and the use of ASTs as part of the embedding into the input along with code sequences is used in this research as it is validated to be a valid methodology in the above research.

2.2 Graph Neural Network

Graph Neural Networks (GNN) are a key to multiple problems because the key relationships between things naturally take the form of a graph, there are multiple variations of GNNs that are described by the survey by Wu et al. (2020). Convolutional Graph Neural Networks (ConvGNNs) are of relevant to this research as it involves the ’convulsing’ of data across multiple nodes as it learns information from nodes that surround it, at the end of the propagation each node will have a vectorized information about the nodes that surround it to varying degree, the information is then aggregated and arbitrary relationships are learnt (Kipf and Welling; 2016).

GNNs have been used to discover relationships in multitude of problems such as parts of speech, parsing of dependencies between two sequences, or two sequence of token (Xu, Wu, Wang, Yu, Chen and Sheinin; 2018), molecular structure prediction (Duvenaud et al.; 2015), computational biology (Gao et al.; 2021) and natural question generation (Chen et al.; 2019). The natural dependency of code and its relationships innately takes the shape of a graph, and the compiled code is generally used to construct an AST which is naturally a tree, which is a special graph, hence the use of GNN for this research makes it part of the research objective.

The use of graph based structure was first proposed by Allamanis et al. (2017) their work was in the area of code generation. GNNs have also been used to tasks that are close to code summarization such as the ones in graph2seq model for parsing of semantics between code sequences (Xu, Wu, Wang, Feng, Witbrock and Sheinin; 2018). LeClair et al. (2020) in their study of the usage of GNNs and their effort to explain the reason behind the justification and usability of GNNs showed promising results. They used GNNs to learn AST representation in a better way and used GNN as an input embedding to the encoder-decoder architecture. GNNs have also used in the smart contracts to generated summaries from APIs (Yang et al.; 2021).

Thus, by looking into the research using GNNs in this area, this project proposes the use of ASTs as an input into GNNs.

2.3 Evaluation Metrics

This research project proposes to use both Bilingual Evaluation Understudy Score (Papineni et al.; 2002) and Recall-Oriented Understudy for Gisting Evaluation (Lin; 2005). These are largely used in almost all of the literature cited earlier (Iyer, Konstas, Cheung and Zettlemoyer; 2016; LeClair, Haque, Wu and McMillan; 2020). This section explores the literature and justifies the usage of these two metrics in this paper for evaluation.

As mentioned by Papineni et al. (2002) can be thought of as precision, where in how of the generated text can be seen in the reference text (labelled data). While ROUGE is in contrast a measure of how much reference (labelled) data can be seen in the generated text. ROUGE uses the longest common sub-sequence of sentence similarity while BLEU uses the n-gram similarity, thus BLEU 1 to BLEU n means n-grams of length 1-n. Combining BLEU with the overall sentence similarity metric like ROUGE will make the evaluation more rounded and automated. This coupled with the vast research that supports and uses these evaluation metrics has given sufficient confidence in these methods to be used to verify the results of this experiment.

Apart from automated metrics, Shi et al. (2021) and Mahmud et al. (2021) in their study, questioned the justification of using BLEU as a direct replacement of NMT tasks into code summarization tasks, they experimented on a number of popular datasets, including the one used in this research to check the validity of BLEU scores, they found that the scores varied on a number of reasons like code duplication, data splitting, version of the library used to compute the BLEU score etc., they advise further researches to be wary of high BLEU scores and have suggested to incorporate human evaluation in conflation with automated metrics.

Thus, this project justifiably uses BLEU, ROUGE and human evaluation validate the summaries generated by models.

3 Methodology

This section describes the research methodology that was undertaken that led to the formation of RQ1 and RQ2, and to find answers to the questions established in RQ1 and RQ2, it describes in detail the steps that were followed to bring this research from ideation to fruition.

3.1 Research Formulation

As mentioned in the Section 1 and section 2, this research focused on software engineering and in particular automated ways to generated summarization of code snippets. This research also put an extra interest in using GNN based models to capture the context of a subroutine.

3.2 Surveying relevant literature

As a crucial part of any research, the overreaching part of this paper was to conduct exhaustive survey into the existing literature in this field. The idea of this step was to gain insights into the best practices, establish standard evaluation metrics, choose appropriate data mining mechanisms. A detailed account of related work studied can be found in section 2.

3.3 Data Analytics

The next step of the research was to perform data analytics, this entailed looking into best practices in methodologies that are followed in data analytics projects. The famous data mining strategies of KDD and CRISP-DM were looked into and a hybrid version of each was formulated to fit the bill for this research. An addition to the KDD was the use of business understanding and data understanding, steps that were borrowed from the CRISP-DM methodology. The data mining steps that were followed in this research are represented in Figure 1.

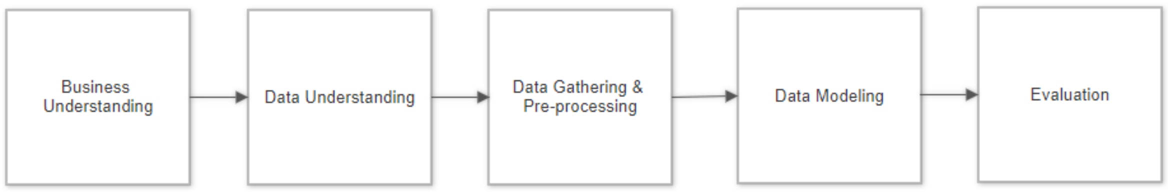


Figure 1: Overall data mining approach followed in this project

1. **Business Understanding :** This step involves the understanding of how the business is conducted, the business of software engineering generally involves monetary benefits against a software product that is shipped, meaning deployed to a server, there are contracts in place between the consumer of the software service and the software service provider, for the quality gates that are necessary to be maintained, there are also automated tools like Sonar [1], that measure the total quality of the project and report its technical debt. A general rule in software engineering is code maintainability and code understanding, this part requires a sound documentation which this research is trying to tackle. Being in the Information Technology industry for nearly a decade helped greatly in the understanding of the business.

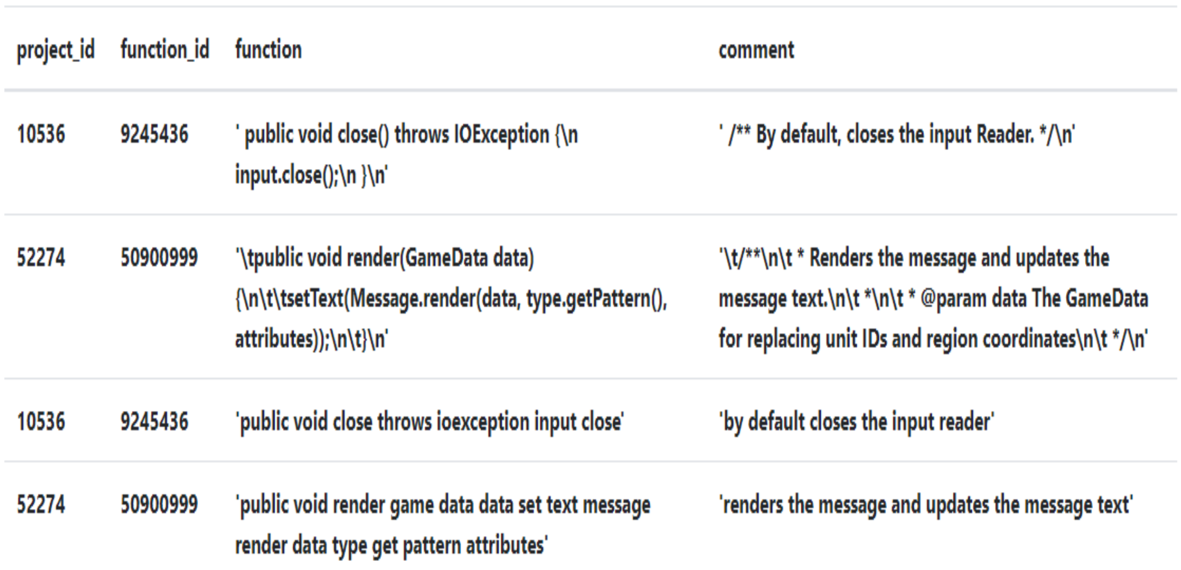
2. **Data Understanding :** The data that were required for this research was source code, code from a popular programming language like Java was sought. The data needed could be understood and being a software engineer and a seasoned Java programmer, understanding of the data that was needed and that fit the research was easily accomplished. Java contains an inbuilt library named JavaDocs that automatically summarizes auto-generated code such as getters and setters, the aim of this project was to predict summaries written for logic that were not auto-generated. Data needed to be rid of auto-generated comments so that the evaluation metrics do not report very high score on comments that are generated by tools. This research focused on code and summaries that are authentic and are written by actual engineers, sufficient literature was surveyed and data that were used and vouched by the research fraternity was therefore selected for this research.

3. **Data Gathering :** After sufficient literature survey, the Funcom[2] dataset provided by LeClair et al. (2019b), they curated the dataset from 56 million java code-comment pairs by removing auto-generated code, the code that did not contain comments (LeClair and McMillan; 2019). The resulting data was 2.1 million records of Java code-comment pairs, each method signature carries a project id and a function id, by using this project id and function id, a map can be created of the dependencies between method and its comment. Their raw, filtered and tokenized datasets were downloaded. The dataset is depicted in Figure 2.

4. **Data Pre-processing :** The raw dataset that was downloaded was used for AST embeddings, the project id and file id that was present was used to generate AST embeddings using srcML[3], which generated an XML representation of the source code which could be fed to the GNN models in the modelling phase. This AST representation was taken from LeClair et al. (2020)[4]. The tokenized data, containing code and comment corpus was explored. The tokenized dataset was then split into train, test and validation, while the AST data was split based on hyper-parameters as recommended by Haque et al. (2020).

5. **Data Modelling :** There are a number of modelling methods which were considered for this project by surveying the relevant literature, the following methods were explored,

[1]https://www.sonarqube.org/
[2]http://leclair.tech/data/funcom/
[3]https://www.srcml.org/
[4]https://icpc2020.s3.us-east-2.amazonaws.com/dataset.pkl



project_id	function_id	function	comment
10536	9245436	' public void close() throws IOException {\n input.close();\n }\n'	' /** By default, closes the input Reader. */\n'
52274	50900999	'\tpublic void render(GameData data) {\n\t\tsetText(Message.render(data, type.getPattern(), attributes));\n\t}\n'	'\t/**\n\t * Renders the message and updates the message text.\n\t *\n\t * @param data The GameData for replacing unit IDs and region coordinates\n\t */\n'
10536	9245436	'public void close throws ioexception input close'	'by default closes the input reader'
52274	50900999	'public void render game data data set text message render data type get pattern attributes'	'renders the message and updates the message text'

Figure 2: Funcom Java 2.1m dataset.

- Using Transformer based approach with multi-head attention was initially considered for, however, the work by Yang et al. (2021), showed that it is more suitable for data that involve APIs and smart contracts, the transformer based approach also needed a lot of computation time as it involves multi-head attention and feed-forward networks.
- Flattening of AST and using the flattened AST into the encoder instead of GNN was also considered, however the research by Iyer et al. (2016) shows that this approach does not capture the true nature of tree-like structure and its learning representation is heavily contrained.

The choice of modelling techniques was corroborated by plethora of research works that have been done in the field of NMT and NLP, using encoder-decoder model with attention mechanism and GNN is used for capturing context information from the AST embeddings of the srcML library. An overview of the model flow is shown in Figure 3 .The individual models are described below

- Encoder: An encoder is given a set of tokenized vectors as inputs, the vectors are padded by zeros so that all the inputs can be batched. Encoders usually learn the features of an input and update their internal state, this internal state is represented conventionally by h and c in code, encoders are usually Recursive Neural Networks(RNN), popular RNNs that are used are LSTM and BiDirectional LSTMs, this research chose BiDirectional LSTM due to the fact that the encoder can update its internal state and importance, known as *context vector* by traversing back and forth (Iyer et al.; 2016).Tokenized code sequences and the output from GNN are given as inputs to encoder.
- Attention Layer: An attention layer learns takes the context vector from the encoder and learns the importance of each of the state and assigns higher

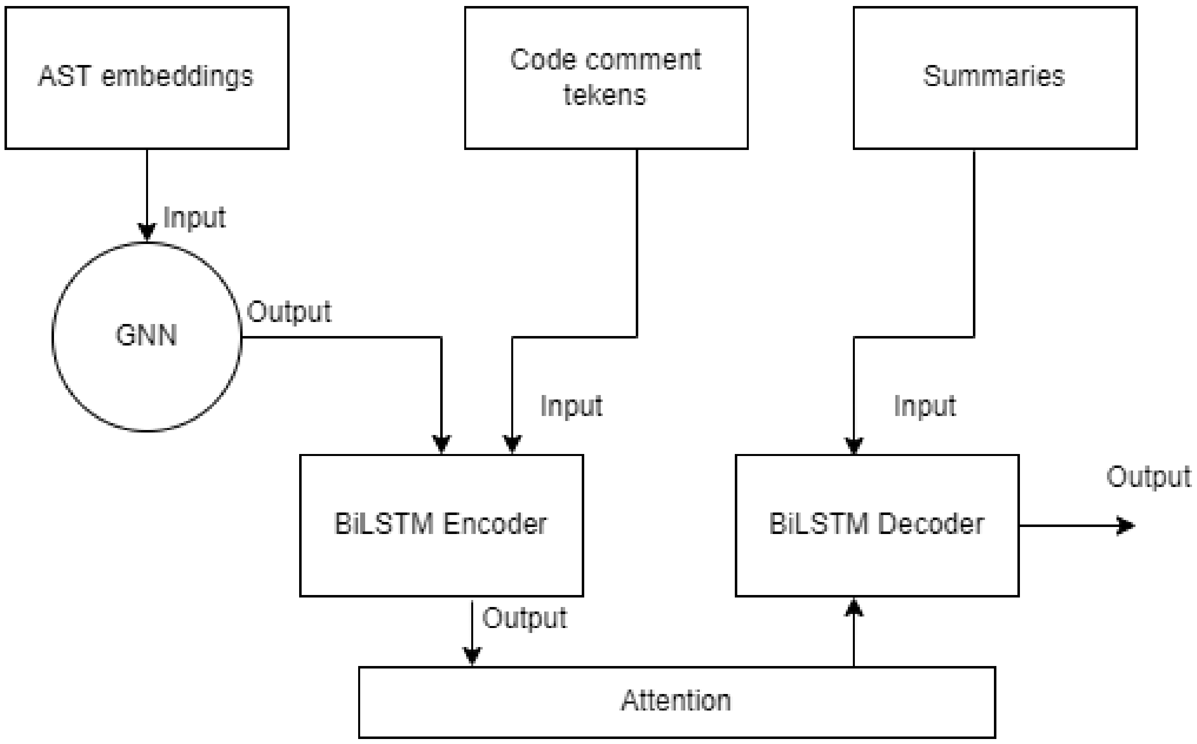


Figure 3: Model Overview

importance, this context vector with attentional weight is then passed on to the decoder. There are multiple types of attention implementations, the options are softmax, dot or the attentional mechanism proposed by Vaswani et al. (2017). This research makes use of both dot and the Vaswani et al. (2017) implementations.

- Decoder: A decoder is also an RNN that is given the attentional weights as the input from the attention layer, the decoder also takes in the subsequent summaries (comments) of code sequences as a parallel input, the attentional weights and the summary input will then be concatenated to predict the next word in the summaries. Bidirectional LSTMs are used for decoders as well (LeClair et al.; 2019b).
- Graph Neural Network: ConvGNNs are used to learn representations of the code to learn contextual information. The AST node and edges are passed as embeddings into the GNN which will convolve and learn important information about all the surrounding nodes, the output from the convGNN is passed as a seperate input to the encoder which will then learn features and place it in the context vector as described above. ConvGNNs were chosen as per the recommendation by LeClair et al. (2020).

3.4 Evaluation

This research uses two main metrics for evaluation, they are BLEU (Papineni et al.; 2002) and ROUGE (Lin; 2005). Research methodologies studied in section 2.3 corroborated the use of these metrics.

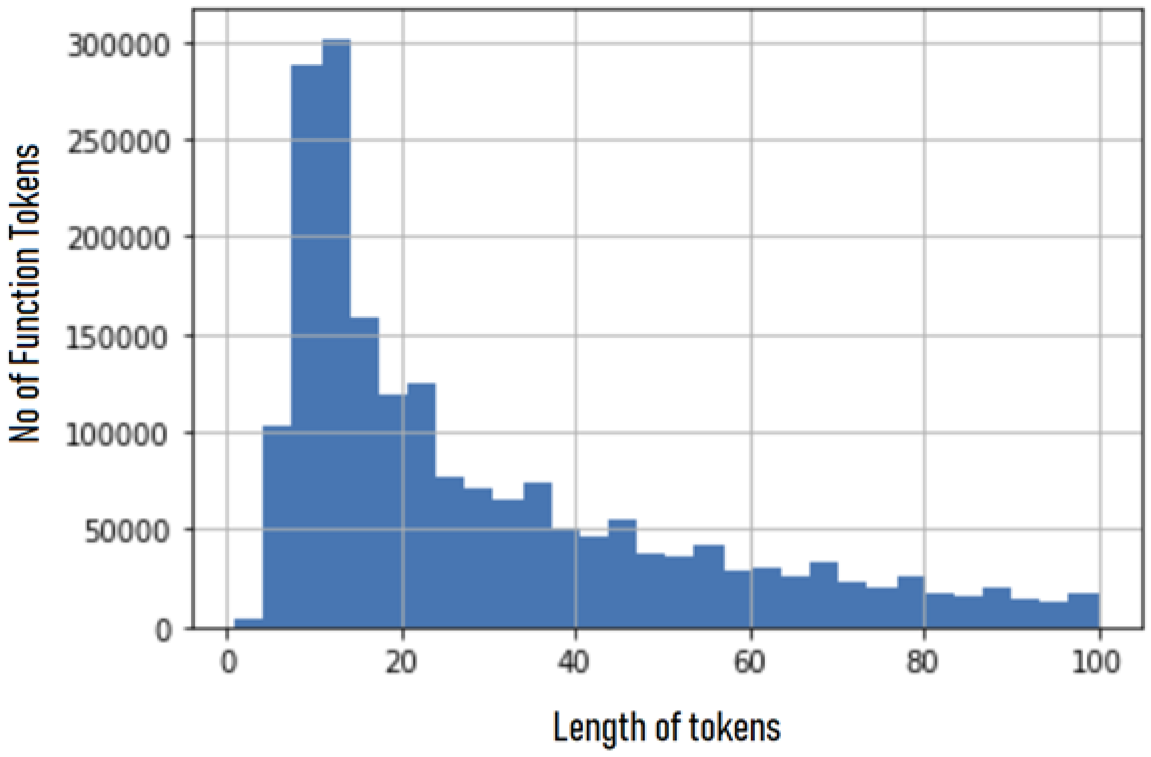


Figure 4: Length of function tokens

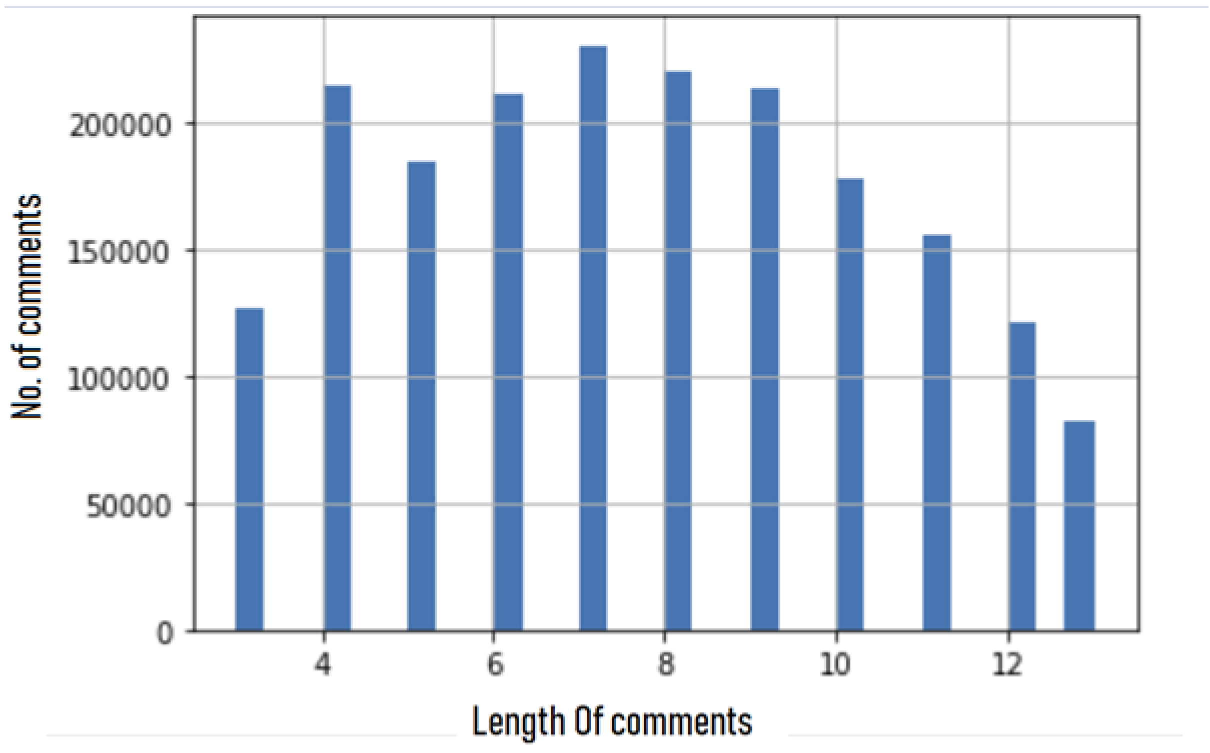


Figure 5: Length of Comment (summary) tokens

1. BLEU: BLEU was originally developed for NMT tasks, it takes into consideration the closeness between a generated sentence to a reference sentence, it is precision oriented. BLEU is used in almost all the papers that were studied in section 2.1. BLEU is denoted by BLEU-1, BLEU-2, BLEU-3, BLEU-4 to measure the precision. A score of 0 would mean that the generated summary has zero overlap with the reference summary. The original BLEU proposed by Papineni et al. (2002) measures sentence level and any sentence with geometric mean less than 4 would need smoothing as suggested by Chen and Cherry (2014). This paper uses the nltk package BLEU which uses corpus level similarity. The length of tokenized functions and comments are shown in Figures 4 and 5, the length of the sequences were carefully curated to less than 100 for function tokens in the filtered dataset by LeClair et al. (2019b) and majority of the function tokens contain less than 20 tokens, thus the usage of BLEU in this case at a corpus level is justified and is recommended by LeClair and McMillan (2019) in their work describing best practices for datasets for code summaries.

2. ROUGE: ROUGE is recall oriented automated metric (Lin; 2005), it also considers overlapping n-grams, pairs of words or sequences between generated and reference summaries. There are variants of ROUGE similar to BLEU, we consider ROUGE Longest Common Sub-sequence (ROUGE-LCS) in this paper. The reason for choosing ROUGE is that BLEU scores can yield higher scores if individual words overlap, for example "save file to database" and "save database to file" are two drastically different summaries but they generate high BLEU score due to its calculation of n-gram overlaps, to keep this limitation in check, ROUGE-LCS is used which checks the longest common sub-sequence between the reference summary and generated summary.

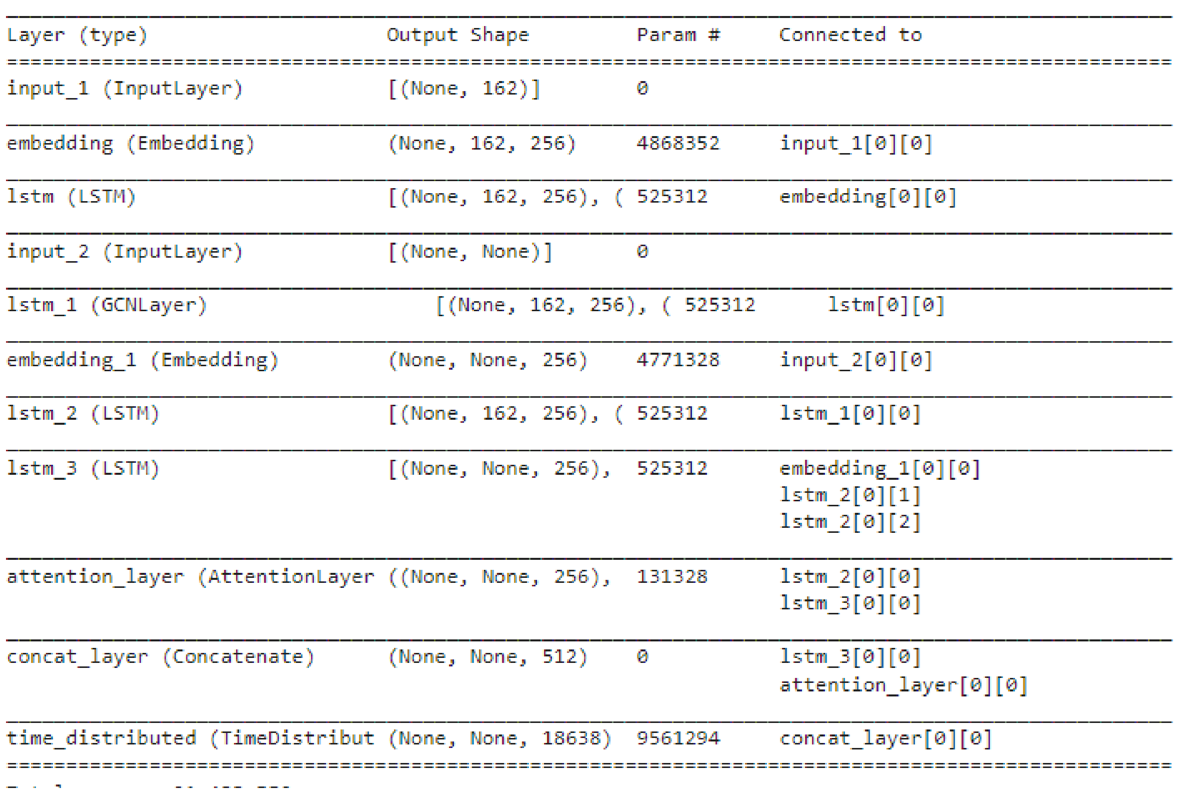
3. Human Evaluation: Although not at the large scale as using Amazon Mechanical Turks[5], which was followed by Zhang et al. (2020) in their approach, this research is cognizant of the shortcomings that have been recorded by recent researches such as Shi et al. (2021) and Mahmud et al. (2021), hence to evaluate context related information, this research proposes the usage of human evaluation of generated summaries along with their function and project ids, if the summaries contain words and learning from functions that contain the same project ids, it can be surmised to a considerable confidence that context information is learnt by GNN.

4 Design Specification

The model's architecture is depicted in Figure 6, it describes the overall model architecture that was designed to answer RQ1 and RQ2. The figure depicts the embedding layer, encoder layer, GNNs, dense and encoder layers. The code sequences are initially tokenized and padded so that training inputs are of the same size and could be batched respectively. The input consists of 256-dimension tokenized code sequence input to the encoder, the encoder is given an embedding layer, which helps in finding features in the input space. The embedding layer is also the same size as the input. AST embeddings is fed to the ConvGNN, the GNN implementation by LeClair et al. (2020) is chosen over the latest GNN implementation by tensorflow due to higher learning curve. The number of

[5]https://www.mturk.com/

hops that is passed as an input to the GNN is a hyper-parameter which is chosen to be 2, higher hop sizes and its detriments are discussed in section 6. The encoder is chosen to be bidirectional LSTM of 256 size, as per the works of Xie et al. (2021), it is the best RNN that could give the encoder propagation back and forth. The vectorized embeddings of the GNN is then passed as a separate input to the encoder. The hidden states of the encoder are captured and passed on to the attention layer, the attention layer is given the hidden state of the encoder as well as the encoder embedding, the encoder is then initialized with the initial state of the encoder. The decoder is also a bidirectional LSTM of 256 size, the decoder is also given the summary of the code sequences, the next word in the summary will get predicted as and when the input code sequences are sent by the encoder. The decoder is then sent to the TimeDistributed dense layer with softmax activation which contains the vectorized form of predictions.



```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 162)]        0
__________________________________________________________________________________________________
embedding (Embedding)           (None, 162, 256)     4868352     input_1[0][0]
__________________________________________________________________________________________________
lstm (LSTM)                     [(None, 162, 256), ( 525312      embedding[0][0]
__________________________________________________________________________________________________
input_2 (InputLayer)            [(None, None)]       0
__________________________________________________________________________________________________
lstm_1 (GCNLayer)                   [(None, 162, 256), ( 525312      lstm[0][0]
__________________________________________________________________________________________________
embedding_1 (Embedding)         (None, None, 256)    4771328     input_2[0][0]
__________________________________________________________________________________________________
lstm_2 (LSTM)                   [(None, 162, 256), ( 525312      lstm_1[0][0]
__________________________________________________________________________________________________
lstm_3 (LSTM)                   [(None, None, 256),  525312      embedding_1[0][0]
                                                                 lstm_2[0][1]
                                                                 lstm_2[0][2]
__________________________________________________________________________________________________
attention_layer (AttentionLayer ((None, None, 256),  131328      lstm_2[0][0]
                                                                 lstm_3[0][0]
__________________________________________________________________________________________________
concat_layer (Concatenate)      (None, None, 512)    0           lstm_3[0][0]
                                                                 attention_layer[0][0]
__________________________________________________________________________________________________
time_distributed (TimeDistribut (None, None, 18638)  9561294     concat_layer[0][0]
==================================================================================================
```

Figure 6: Model architecture.

5 Implementation

For the realization of the research questions set out in this research. Jupyter Notebook was used to fetch and explore the dataset. The dataset was in tokenized format and AST format that was used as input by BiLSTM and GNN respectively. Python 3.7 was chosen as the principle programming language of the whole project owing to its popularity and rich user guides and online help present.Google colaboratory was used to develop the project as it gives plugins to connect with google drive as well as github. The creation of environment and installations become easier as compared to setting everything up in the local environment. Pandas and numpy were used to explore and extract the tokenized and filtered data, and to measure the token length, this enabled the decision on what evaluation metrics that need to be used.

AST representation of the code was taken from LeClair et al. (2020)[6], their implementation of graphlayer was also borrowed as opposed to using the latest tensorflow GNN, this was done due to higher learning curve. The tokenizer that was implemented to read this pkl file was also borrowed and the AST edges and nodes were embedded as keras input and embedded layer into the GNN, the output of the GNN was passed to a bidirectional LSTM with 256 embedded space.

Model layers were built using the APIs provided by keras and tensorflow, each input layer was constructed using the input API given by keras, and an embedding layer was constructed to store the hidden state and higher dimensions, the latent and input dimensions were set to be 256.

Attention layer was added by passing the hidden state (h and c), the standard implementation of attention layer that is proposed by Vaswani et al. (2017) was taken, the decoder was also constructed using the input API by keras and an embedding layer consisting the hidden states and the initial encoder states. LSTM and BiDirectional packages of tensorflow were used to construct all the encoder-decoder models in this project. Early stopping was used to make sure that the model does not overfit. Methods were added to reverse engineer the output from the model back to words for it to be comprehensible. Model evaluation was performed programmatically using nltk packages 'bleu' and 'rouge-l'.

6 Model Evaluation

The results of the experiments are shown in this section and discussed in detail. To better capture the effect of GNNs, an ablation study is performed, where in the model is developed without GNN, and another with GNN. Each experiment also mentions the hyper-parameters that were used for the experiment.

6.1 BiLSTM

The BiLSTM model was a tradition sequence to sequence model which only leveraged the sequential token data that was passed to it as input. The study was done with epoch size of 50 and batch size of 64, it contained BiLSTMs for encoder and decoder with attention layer. The Code sequences were fed to the input layer and summaries to the decoder. AST embeddings were not added to this experiment for comparison to answer RQ1.

From Table 1, it can be interpreted that when comparing the generated summaries and reference summaries, when 1-gram overlapping is calculated (BLEU-1), 35.6% of 1-grams present in the generated summaries were also present in the reference summaries, when 2-gram overlapping is calculated (BLEU-2), it is 21.2%, 12.6% for BLEU-3 and 9.3% for BLEU - 4.

Similarly, the ROUGE-LCS scores can be interpreted as that 55.2% of the longest common sub-sequence of generated text is overlapping with the longest common sub-sequence of reference text, while recall shows that 46.4% of longest common sub-sequence present in the reference text, is also present in the generated text.

[6]https://icpc2020.s3.us-east-2.amazonaws.com/dataset.pkl

Table 1: Comparison of BLEU Metrics.

Model	**BLEU** 1	**BLEU** 2	**BLEU** 3	**BLEU** 4
BiLSTM	35.6	21.2	12.6	9.3
BiLSTM+GNN	**36.4**	**21.4**	**14.3**	**11.2**

Table 2: Comparison of ROUGE-LCS Metrics.

Model	**Precision**	**Recall**	**F1-score**
BiLSTM	54.6	46.4	48.0
BiLSTM+GNN	**55.2**	**47**	**49**

6.2 BiLSTM + GNN

An extension to the earlier experiment was done to answer the questions in RQ1, the experiment was run with an epoch size of 20. The hop size of GNN was set to 4, the experiment was run with TPU configurations in Google Collab, however due to the enormity of the number of variables that each node had to learn due to the hop size, which is the number of nodes, a given node needs to learn, this model consistently ran out of memory despite multiple trials and had to be abandoned. Hyper-parameter for hop size of GNN was set to 2 as per the recommendations by LeClair et al. (2020), they set their hop size to 2 in their studies, and the results of it are shown in Table 1.

As opposed to the evaluation scores of BiLSTM in 6.1, it is seen that the BLEU scores in n-gram overlapping are higher and are consistently higher even when more n-grams are overlapped as indicated by BLEU scores.

The overlap of longest common sub-sequences between generated summary and reference summary in precision and reference summary and generated summary in recall is also evident from the Table 2.

6.2.1 Human Evaluation of context capturing

While automated metrics are a great way of measuring the performance of models quantitatively, there is no direct metric that can measure or do justice if the context was learnt by the GNNs. Just reporting a higher BLEU score as shown by studies in Haque et al. (2020),Zügner et al. (2021) and Haque et al. (2020) can be misguiding as the direct usage of NMT metrics sometimes do not translate well when applied to code summarization tasks (Shi et al.; 2021). Hence a human evaluation of 5% summaries generated by biLSTM+GNN were compared against the BiLSTM model and the overall observation of random summaries suggest that the GNNs, although contributed in the increase in the metrics score in Tables 1 and 2, the context information was not captured. The possible reasoning and implication is discussed in detail in section 7

Table 3: Example of generated summaries of function id 8867852 and project id 9728.

Model	**Summary**
Reference	set the value of fq
BiLSTM	sets the ¡UNK¿ of this filter
BiLSTM+GNN	sets the filter query

7 Discussion

This section discusses in detail the results presented in the evaluation (section 6). The discussion is divided into the two RQs presented in 1.2.

RQ1: RQ1 sought to see the quantitative improvement from GNNs. From the results presented in Tables 1 and 2, it is clearly seen that GNNs have a qualitative effect on the code summarization tasks. GNNs have brought, on an average, about a point increase in the BLEU scores in the generated summaries. This increase can be attributed to the nodes of the GNN learning about the AST embeddings. Another advantage of using GNN is that it can learn the words that appear rarely in the summaries by learning the AST embeddings. The BiLSTM based models generally outperform all other models as noted by LeClair et al. (2020) in their research, even when AST embeddings are flattened, tokenized and fed as an input either along with code sequences or as separate output. The GNN based models however give an increased performance at the cost of computation and resources. The GNN models at any time need to hold many variables as its hidden state and given the innate depth of graphs that are present in program ASTs, this could lead to significant computation costs. Another significant advantage of GNNs over other models is that they can employ 'copy mechanism', wherein new words are directly copied from the source tokens (Gu et al.; 2016).

RQ2: The context capture of GNN was evaluated by human evaluation rather than metrics as metrics such as BLEU can sometimes give a very high score (Shi et al.; 2021). There are multitude of reasons as to why the model could not capture context, it could be that the hop size of GNN was only to two levels where just the method name and its immediate operation in the AST were embedded into nodes and edges, a higher hop-size with a bigger computation power and time might yield the contextual information that are present in the AST embeddings. Another reason could be the attentional mechanism, where in the AST embedding and the code sequences must be 'aligned' for the GNN to learn the code context that is around the particular code sequence that is fed to the BiLSTM.

As a limitation, it is observed that although the BLEU scores are better with GNN model, it can be seen that when the n-gram overlapping increases, the precision score is decreasing, this could be because of the brevity penalty that gets applied when the generated summary is too small to calculate recall. The assumption of two functions having the same project id to have context needs more corroboration and the dataset must be curated especially to capture context. Another limitation of this approach is the computing time that it takes to train graph related models, the general evaluation metric improvement entails higher computation power which would mean tough trade-off decisions, it could be overcome by higher hop size hyperparam setting but that would again mean further computation power. The usage of GNN along with BiLSTM also hinders the interpretability of the model, although this could be overcome by using history and checkpoints. This project uses Java dataset and is not generalized to inputs from other programming languages.

8 Conclusion and Future Work

Code summarization is a niche field and exciting field. Through *RQ1* and *RQ2*, this research set out to explore the ambitious task of generating human-like automated summaries of Java code. It also explored and identified the advantages graph-based models have over the normal RNN based models while attending to learning code sequences that are embedded into ASTs. Through experimentation it was found that GNN models were better capable of generating summaries which was corroborated by the metric scores. The contextual information that was sought in RQ2 was evaluated using the project id, file id of the data and the generated summaries did not have the information of the project, which was expected if the context was captured. GNN based models significantly outperformed the RNN based model in all of the BLEUs, and ROUGE-LCS metrics.

A great arena for future work would be to generate language agnostic models that would be capable of generating summaries irrespective of the programming language that is given as input, the use of method signature or API signature could be leveraged to create such a model. Another area is where summaries could be reverse engineered to generate code i.e code generation.

Usage of a project's dependency tree that is generated by build tools such as maven [7] could be another great arena where in the entire library and the project level dependencies could be captured, this could aid in curating datasets that could be used to train models that also learn dependencies between projects and files.

Acknowledgement

Sincere and heartfelt thanks to Dr. Athanasios Staikopoulos for his support, insights, reviews and guidance throughout the duration of this project.

References

Allamanis, M., Brockschmidt, M. and Khademi, M. (2017). Learning to represent programs with graphs, *arXiv preprint arXiv:1711.00740* .

Alon, U., Zilberstein, M., Levy, O. and Yahav, E. (2019). Code2vec: Learning distributed representations of code, *Proc. ACM Program. Lang.* **3**(POPL).
URL: *https://doi.org/10.1145/3290353*

Banerjee, S. and Lavie, A. (2005). Meteor: An automatic metric for mt evaluation with improved correlation with human judgments, *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pp. 65–72.

Bansal, A., Haque, S. and McMillan, C. (2021). Project-level encoding for neural source code summarization of subroutines, *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 253–264.

[7]https://maven.apache.org/

Chen, B. and Cherry, C. (2014). A systematic comparison of smoothing techniques for sentence-level bleu, *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pp. 362–367.

Chen, Y., Wu, L. and Zaki, M. J. (2019). Reinforcement learning based graph-to-sequence model for natural question generation, *arXiv preprint arXiv:1908.04942* .

de Souza, S. C. B., Anquetil, N. and de Oliveira, K. M. (2005). A study of the documentation essential to software maintenance, *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pp. 68–75.

Duvenaud, D., Maclaurin, D., Aguilera-Iparraguirre, J., Gómez-Bombarelli, R., Hirzel, T., Aspuru-Guzik, A. and Adams, R. P. (2015). Convolutional networks on graphs for learning molecular fingerprints, *arXiv preprint arXiv:1509.09292* .

Gao, J., Lyu, T., Xiong, F., Wang, J., Ke, W. and Li, Z. (2021). Predicting the survival of cancer patients with multimodal graph neural network, *IEEE/ACM Transactions on Computational Biology and Bioinformatics* .

Gu, J., Lu, Z., Li, H. and Li, V. O. (2016). Incorporating copying mechanism in sequence-to-sequence learning, *arXiv preprint arXiv:1603.06393* .

Haiduc, S., Aponte, J., Moreno, L. and Marcus, A. (2010). On the use of automated text summarization techniques for summarizing source code, *2010 17th Working Conference on Reverse Engineering*, IEEE, pp. 35–44.

Haque, S., Bansal, A., Wu, L. and McMillan, C. (2021). Action word prediction for neural source code summarization, *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 330–341.

Haque, S., LeClair, A., Wu, L. and McMillan, C. (2020). Improved automatic summarization of subroutines via attention to file context, *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, Association for Computing Machinery, New York, NY, USA, p. 300–310.
URL: *https://doi.org/10.1145/3379597.3387449*

Hu, X., Li, G., Xia, X., Lo, D. and Jin, Z. (2018). Deep code comment generation, *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pp. 200–20010.

Hu, X., Li, G., Xia, X., Lo, D., Lu, S. and Jin, Z. (2018). Summarizing source code with transferred api knowledge.

Iyer, S., Konstas, I., Cheung, A. and Zettlemoyer, L. (2016). Summarizing source code using a neural attention model, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2073–2083.

Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks, *arXiv preprint arXiv:1609.02907* .

LeClair, A., Haque, S., Wu, L. and McMillan, C. (2020). Improved code summarization via a graph neural network, *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, Association for Computing Machinery, New York, NY, USA, p. 184–195.
URL: *https://doi.org/10.1145/3387904.3389268*

LeClair, A., Jiang, S. and McMillan, C. (2019a). A neural model for generating natural language summaries of program subroutines, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 795–806.

LeClair, A., Jiang, S. and McMillan, C. (2019b). A neural model for generating natural language summaries of program subroutines, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, pp. 795–806.

LeClair, A. and McMillan, C. (2019). Recommendations for datasets for source code summarization, *arXiv preprint arXiv:1904.02660* .

Lin, C. (2005). Recall-oriented understudy for gisting evaluation (rouge), *Retrieved August* **20**: 2005.

Lin, C., Ouyang, Z., Zhuang, J., Chen, J., Li, H. and Wu, R. (2021). Improving code summarization with block-wise abstract syntax tree splitting, *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 184–195.

Liu, P.-f. and Wang, X.-m. (2020). Utilizing keywords in source code to improve code summarization, *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, pp. 664–668.

Mahmud, J., Faisal, F., Arnob, R. I., Anastasopoulos, A. and Moran, K. (2021). Code to comment translation: A comparative study on model effectiveness & errors, *arXiv preprint arXiv:2106.08415* .

McBurney, P. W. and McMillan, C. (2015). Automatic source code summarization of context for java methods, *IEEE Transactions on Software Engineering* **42**(2): 103–119.

Papineni, K., Roukos, S., Ward, T. and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation, *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318.

Shi, E., Wang, Y., Du, L., Chen, J., Han, S., Zhang, H., Zhang, D. and Sun, H. (2021). Neural code summarization: How far are we?, *arXiv preprint arXiv:2107.07112* .

Slinker, G. (2013). More on code debt, *Capturado em: http://digeratiilluminatus. blogspot. com/2008/03/more-on-code-debt. html, Setembro* .

Szykarski, A. (2012). Ted theodorpoulos on managing technical debt successfully.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. and Polosukhin, I. (2017). Attention is all you need, *Advances in neural information processing systems*, pp. 5998–6008.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C. and Philip, S. Y. (2020). A comprehensive survey on graph neural networks, *IEEE transactions on neural networks and learning systems* **32**(1): 4–24.

Xie, R., Ye, W., Sun, J. and Zhang, S. (2021). Exploiting method names to improve code summarization: A deliberation multi-task learning approach, *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 138–148.

Xu, K., Wu, L., Wang, Z., Feng, Y., Witbrock, M. and Sheinin, V. (2018). Graph2seq: Graph to sequence learning with attention-based neural networks, *arXiv preprint arXiv:1804.00823* .

Xu, K., Wu, L., Wang, Z., Yu, M., Chen, L. and Sheinin, V. (2018). Exploiting rich syntactic information for semantic parsing with graph-to-sequence model, *arXiv preprint arXiv:1808.07624* .

Yang, Z., Keung, J., Yu, X., Gu, X., Wei, Z., Ma, X. and Zhang, M. (2021). A multi-modal transformer-based code summarization approach for smart contracts, *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 1–12.

Zhang, J., Wang, X., Zhang, H., Sun, H. and Liu, X. (2020). Retrieval-based neural source code summarization, *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 1385–1397.

Zügner, D., Kirschstein, T., Catasta, M., Leskovec, J. and Günnemann, S. (2021). Language-agnostic representation learning of source code from structure and context, *arXiv preprint arXiv:2103.11318* .