REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifiq
Université Mohamed Khider – BISKRA
Faculté des Sciences Exactes, des Sciences de la Nature et de la vie

Département d'informatique

N° d'ordre:SIOD34/M2/2022

Mémoire

Présenté pour obtenir le diplôme de master académique en

Informatique

Systeme d'Information Optimisation Et Decision (SIOD)

Autocomment Generation from Source Code using ML Techniques

Par :

DORSAF KIHAL

Dédicace

I dedicate this work to myself.

Remerciement

First of all, I would like to offer thanks to Allah for having granted us all the determination, and strength to carry out this modest work. I would also like to be infinitely grateful to my supervisor Dr.TIBERMACINE Okba for his advice, his patience, his availability, and his support throughout this period.

I would like to express my gratitude to my parents Abd ELmadjid Rania ,my sister and my brothers Abd Raouf and Fares Seif Dine, who gave me their moral support and lively discussions throughout my effort, and my grandmother Rachida and my grandfatherAhmed Bouchouareb who always accompanied me with his prayers.

We would like to express our deepest gratitude and sincere thanks to the members of the jury for having accepted to judge our work and to have enriched it.

Abstract

During software maintenance, developers spend a lot of time understanding the source code. Existing studies show that code comments help developers comprehend programs and reduce additional time spent on reading and navigating source code. Unfortunately, these comments are often mismatched, missing or outdated in software projects. Developers have to infer the functionality from the source code. This paper proposes a new approach to automatically generate code comments for the functional units of Java language, namely, Java methods. The generated comments aim to help developers understand the functionality of Java methods.

Keywords: comment generation, deep learning, machine learning, LSTM, Code2Seq, Software Development, Java

General Introduction

Contexte

Source code comments play a crucial role in software development by enhancing code readability, maintainability, and collaboration among developers. These comments provide explanations and descriptive language that is not executed by the program itself. They help both the original author and future developers understand the purpose, functionality, and logic of the code by providing additional details and insights.

Generating comments within source code is a fundamental practice in software development. It involves adding explanatory text to clarify the purpose, functionality, and logic of the code. Such comments serve as documentation, benefiting both the original author and future developers who work on the codebase. Autocomment creation improves code readability, documentation, and relieves developers from the manual task of writing and maintaining comments. It also ensures consistent and informative code documentation.

However, due to factors such as tight project schedules and other constraints, code comments are often mismatched, missing, or outdated in many software projects. Automatic generation of code comments can address these issues, saving developers time and aiding in source code understanding. Several approaches have been proposed to generate comments for methods and classes in Java, which is the most popular programming language in the past decade. These techniques range from manual crafting to leveraging Information Retrieval. In this thesis, we aim to develop a novel model for generating descriptive comments based on Java source code and Abstract Syntax Tree (AST) using NMT based techniques. A special attention is given to Code2seq and Open NMT architecture to develop automatic comment generator from source code.

Objective and Research Questions:

The objective of this master thesis is to explore the generation of comments from source code using Neural Machine Translation (NMT). NMT has shown significant advancements in natural language processing tasks and has the potential to create high-quality translations. By applying NMT approaches to generate comments from source code, we can investigate the feasibility of leveraging these models to automate the process and enhance the clarity and effectiveness of comment generation.

The primary research question addressed in this study is as follows:

- Research Question: How can Neural Machine Translation be leveraged to generate comments from Java code snippets (Java methods)?

Manuscript organization

This manuscript is organized into four chapters as follows:

- Chapter 1: Neural Machine Translation: This chapter provides an overview of the foundational concepts and techniques of neural machine translation.

- Chapter 2: Source Code Analysis and Documentation: In this chapter, we describe source code analysis, documentation practices, and explain the concept of abstract syntax tree (AST) and structure-based traversal.

- Chapter 3: NMT-based Comment Generation from Source Code: The third chapter presents our proposed models for generating comments from source code and outlines the generation process. It also includes a discussion of related work in the field.

- Chapter 4: Implementation, Testing, and Validation: In this chapter, we showcase the frameworks, tools, and libraries employed in this project. Additionally, we present the results of our work.

- Conclusion: The manuscript concludes with a general summary and some perspectives.

Contents

List of Figures

Chapter 1

Neural Machine Translation

1.1 Introduction

Machine Translation (MT) is an important task that aims to translate natural language sentences using computers. The early approach to machine translation relies heavily on handcrafted translation rules and linguistic knowledge. As natural languages are inherently complex, it is difficult to cover all language irregularities with manual translation rules. In this chapter, we present an overview about Neural Machine translation (NMT), especially in the context of the translation of source code as a sequence of words to comment generation cases. Therefore, we will discuss in general the Recurrent Neural Networks (RNNs) which are sequence-based deep learning architectures. With a focus on long-short-term memory (LSTM) networks which deal with sequential data and time series, RNNs are ideal for mimicking source code's sequential structure.

1.2 Machine Learning

Artificial intelligence (AI) has a subject called machine learning that focuses on developing algorithms and models that let computers learn from data and get better at a particular activity. Machine learning, in other terms, is the act of teaching a computer system to recognize patterns and make choices without being expressly programmed to do so.

supervised learning, unsupervised learning, and reinforcement learning are the three primary categories of machine learning.

In supervised learning, a model is trained on labeled data with the appropriate result being given for each sample of the input. By reducing the discrepancy between its predictions and the accurate labels in the training data, the model learns to map inputs to outputs.

In unsupervised learning, a model is trained on unlabeled data and is required to discover patterns and structure on its own, without being given specific objectives. Tasks like grouping, dimensionality reduction, and anomaly detection require this form of learning.1.1

A model is trained through reinforcement learning to take decisions based on rewards or consequences. The model develops the ability to act in a way that maximizes a success indicator known as a reward signal. In tasks like playing video games, building robots, and solving optimization issues, this kind of learning is employed.

Deep learning, decision trees, random forests, support vector machines, and a number of other subfields are examples of machine learning algorithms that can be further divided into subcategories. [1]

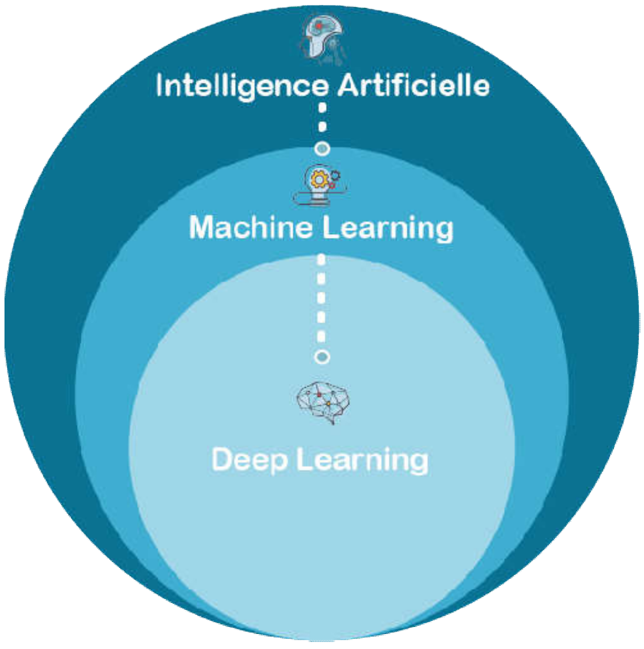


Figure 1.1: Relation between AI and ML and DL

1.3 Neural Machine Translation

Neural machine translation (NMT) is a type of machine translation that uses artificial neural networks to translate text from one language to another.

NMT models train on a huge quantity of bilingual data, in contrast to conventional rule-based or statistical machine translation techniques.

Typically, NMT models have an encoder and a decoder, each of which is a neural network. The decoder receives the input sentence from the encoder and decodes it into a fixed-length representation. Based on the input representation and the previous tokens it has created, the decoder creates the output sentence one token at a time.

NMT models can be trained using a variety of techniques, including supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, the model is trained on pairs of source and target language sentences, with the goal of minimizing the difference between the predicted translation and the correct translation. In unsupervised learning, the model is trained on monolingual data in both the source and target languages, with the goal of learning to translate without explicit supervision. In reinforcement learning, the model is trained to maximize a reward signal, which can be thought of as a measure of translation quality.

NMT models have been shown to be highly effective at translating between a wide range of language pairs, and have achieved state-of-the-art results on several machine translation benchmarks. Some of the advantages of NMT models over traditional machine translation methods include their ability to handle long-range dependencies between words, their ability to learn representations that generalize across different languages, and their ability to produce more fluent and naturalsounding translations.??

NMT models do have significant drawbacks, though, such as their high processing needs, propensity to create translations that are excessively impacted by the training data, and their inability to handle uncommon or uncommon terms. Addressing these drawbacks and enhancing the performance of NMT models across a range of translation tasks are the main goals of ongoing research. The process followed for NMT is depicted in figure 3.1.

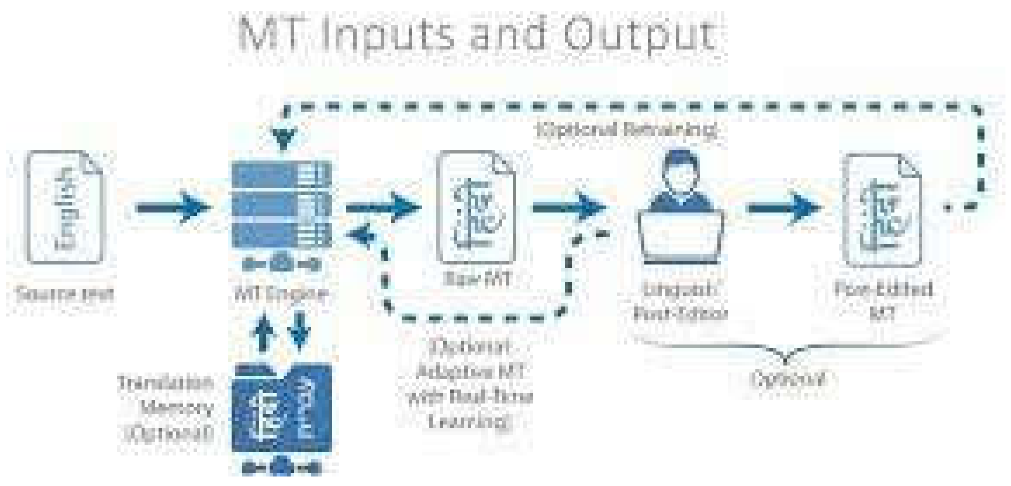


Figure 1.2: Process of Neural Machine Translation

1.4 Neural Networks

Neural networks, often known as artificial neural networks (ANNs) is a machine learning algorithm . The idea of these networks has been inspired from the neurons in human brains that communicate with each other using electrical and chemical signals Which is later translated into an actions , the same way in neural networks . The output from the previous layer is the input to the next layer . In the end, the final value arrives at the output layer and it is taken as a prediction . In deep neural network, The more hidden layers, the more efficient and complex the network.[2] [3]

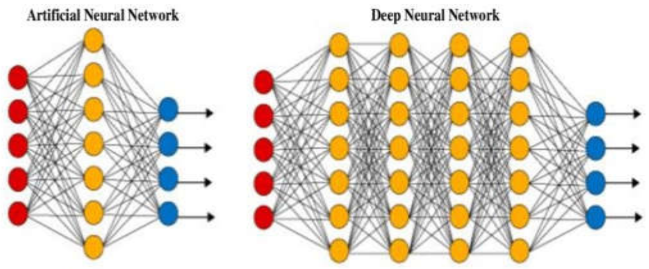


Figure 1.3: Visualization of a neural network and deep neural network

1.4.1 Architecture of Neural Network

In the 1980s, most neuron networks formed only one layer due to the cost of computing and the availability of data. Nowadays we can This allows us to have more layers hidden in our neural networks, hence the nickname deep learning. The different types of neuronal networks available for use They are also known as
Convolutionary Neural Networks (CNNs) and Recurrent neural networks (RNNs).

1.5 Type of Neural Network

1.5.1 Linear Models

Linear models are a core element of statistical machine translation.Possible translations x from A sentence is represented by a set of features hi(x).Each feature is weighted by the parameter i to get the total score.Ignore the exponential function we used to rotate earlier To convert a linear model to a log-linear model, the following formula summarizes the model.

$$\text{score}(\lambda, x) = \sum_j \lambda_j \, h_j(x)$$

Figure 1.4: linear model scoring formula

Graphically, a linear model can be represented by a network of input eigenvalues Nodes, arrows are weights, scores are output nodes (see Figure 2)

Most prominently, we use linear models to combine different components of a machine translation system, such as the language model, the phrase translation model, the reordering model, and properties such as the length of the sentence, or the accumulated jump distance between phrase translations. Training methods assign a weight value i to each such feature hi(x), related to their importance in contributing to scoring better translations higher. In statis- tical machine translation, this is called tuning.

1.5.2 Multiple Layers

Neural systems adjust straight models in two critical ways. The primary is the utilize of multiple layers.Instead of computing the output value directly from the input values, a hidden layer

The network is processed in two steps. First, a linear combination of weighted input node is computed to produce each hidden node value. Then a linear combination of weighted hidden nodes is computed to produce each output node value. At this point, let us present numerical documentations

- a vector of input nodes with values x = (x1, x2, x3, ...xn) T
- a vector of hidden nodes with values h = (h1, h2, h3, ...hm) T
- a vector of output nodes with values y = (y1, y2, y3, ...yl) T
- a matrix of weights connecting input nodes with hidden nodes W = wij
- a matrix of weights connecting hidden nodes with output nodes U = uij

1.6 Neural Languge Models

In order to lessen the effects of the curse of dimensionality, neural network language models, which are based on neural networks, can develop distributed representations.

A neural language model surpasses n-gram language models in terms of dimensionality reduction by using distributed word representations. Several models based on feedforward networks, log-bilinear models, skip-gram models, and recurrent neural networks have been developed. Each word in the vocabulary is represented as a real-valued feature vector, and these vectors are linked together so that the cosine of the angles between these vectors is high for words that are semantically related. [4]

1.6.1 Feed-Forward Network

A feedforward network is a multilayer network in which the units are connected with no cycles; the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.

A feed forward neural network is a type of artificial neural network in which there is no cycle in the connections between the nodes. A recurrent neural network, in which particular paths are cycled, is the reverse of a feed forward neural network. Since information is only processed in one direction, the feed forward model is the simplest type of neural network. Although the data may move through several hidden nodes, it always proceeds forward and never backward.

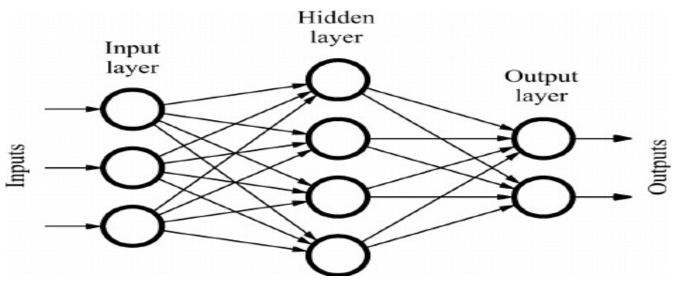


Figure 1.5: feed forward neural network

1.6.2 Word Embedding

Word embeddings, also known as word immersion, are the mapping of words to real-number vectors in a condensed space dimension [34], which may be one of the key advancements for the impressive results of deep learning methods on challenging natural language processing problems. The vectors of word inclusion represent words and their contexts; as a result, words with similar meanings (synonyms) or close semantic relationships will have more similar dives. Additionally, word inclusions must reflect the relationships between individual words. For instance, the word "man" should be written "king" just as "women" is written "queen."

1.6.3 Reccurent Neural Network RNN

Recurrent neural networks (RNNs) are a form of neural network in which the results of one step are fed into the current stage as input. Traditional neural networks have inputs and outputs that are independent of one another, but there is a requirement to remember the previous words in situations when it is necessary to anticipate the next word in a phrase. As a result, RNN was developed, which utilized a Hidden Layer to resolve this problem. The Hidden state, which retains some information about a sequence, is the primary and most significant characteristic of RNNs.

RNNs have a "memory" that retains all data related to calculations. It executes the same action on all of the inputs or hidden layers to generate the output, using the same settings for each input. In contrast to other neural networks, this minimizes the complexity of the parameter set.

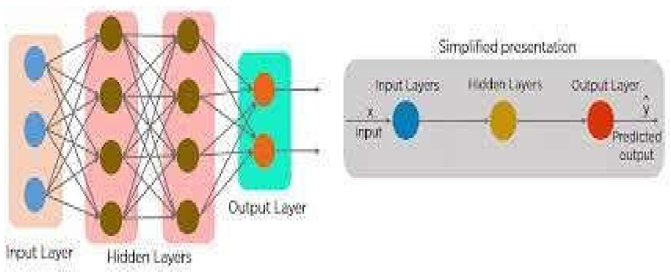


Figure 1.6: Recurrent neural networks (RNNs)

1.6.4 Long Short-Term Memory Models (LSTM)

An RNN model is a version known as long short-term memory (LSTM). A LSTM can handle lengthy time-series data, but an RNN can only memorize short-term information. In addition, lengthy sequence data present an RNN model with the vanishing gradient problem, which LSTM can avoid during training. An LSTM model has automated control over whether to keep important properties in the cell state or toss out unimportant ones, and it can recall past long-term time-series data. The input gate, forget gate, and output gate are the three gates that an LSTM model uses to regulate its characteristics. The input gate regulates how new information enters the cell state.

The main components of the LSTM are its gates. There are three gates in an LSTM: the input gate, the forget gate, and the output gate. The input gate will control the inflow of new information into the cell. The forget gate will control the content of the memory, that is, the forget gate will decide if we want to forget a piece of information so we can store new information.

The output gate will control when the information is used in the output from the cell. A gated recurrent unit (GRU) is a simplification of an LSTM. Unlike an LSTM, a GRU has just two gates. This kind of setup ultimately makes the execution faster as there is a lower number of weights to learn

1.6.5 Gated Reccurent Units

When compared to long short term memory (LSTM), the Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN). GRU is quicker and requires less memory than LSTM, however LSTM is more accurate when working with datasets that contain longer sequences.

Additionally, GRUs address the issue with vanishing gradients that affects regular

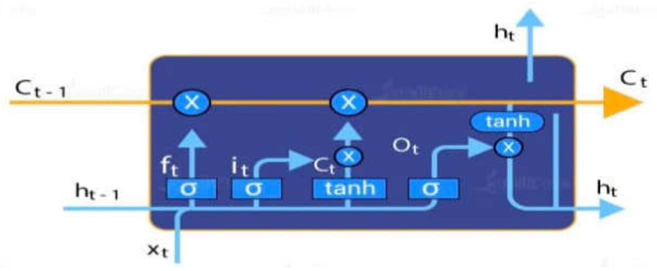
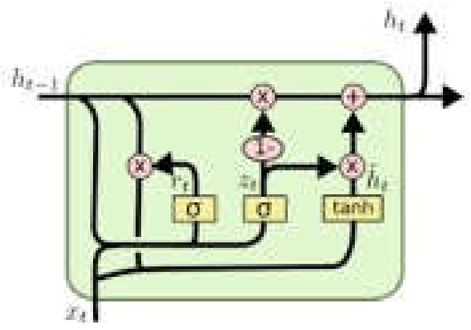


Figure 1.7: Long Short-Term Memory Models

recurrent neural networks (values used to update network weights). Grading may become too little to have an impact on learning if it shrinks over time as it back propagates, rendering the neural network untrainable.

RNNs can basically "forget" lengthier sequences if a layer in a neural net is unable to learn.
GRUs solve this problem through the use of two gates, The update gate and reset gate are two gates that GRUs utilize to address this issue. These gates may be taught to retain information from further back and determine what information is let through to the output. As a result, it can transfer important information along an event chain to improve its forecasts.



$$r_t = \sigma\left(W_r \mathbf{x}_t + U_r \mathbf{h}_{t-1}\right)$$
$$\tilde{h}_t = \tanh\left(W \mathbf{x}_t + U\left(\mathbf{r}_t \odot \mathbf{h}_{t-1}\right)\right)$$
$$z_t = \sigma\left(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1}\right)$$
$$h_t = (1 - z_t) h_{t-1} + z_t \tilde{h}_t$$

Figure 1.8: Gated Recurrent Units

1.6.6 Deep Model

An artificial neural network (ANN) with numerous layers in between the input and output layers is known as a deep model. In a deep model, information is transformed via a number of non-linear layers in a hierarchical fashion. The model may learn increasingly sophisticated representations of the input data by employing several layers, which can increase the predictability of the model.

Deep models have grown in prominence in recent years as a result of their propensity to handle challenging problems including audio, picture, and natural language processing. Convolutional neural networks (CNNs) for image recognition and recurrent neural networks (RNNs) for sequence modeling are two examples of deep models.

However, because of problems like vanishing gradients and overfitting, training deep models can be difficult. To assist with these issues, academics have created approaches including regularization, dropout, and batch normalization.

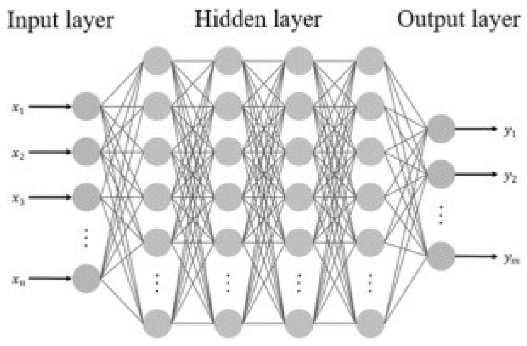


Figure 1.9: deep neural network

1.7 Neural Translation Models(Tronsformer)

The Transformer is a deep learning model that was first proposed in 2017. It uses a "self-attention" method that enhances neural machine translation (NMT) applications' performance in comparison to the conventional recurrent neural network (RNN) model. As a result, training for natural language processing (NLP) tasks is accelerated.

A neural network called a transformer model follows relationships in sequential input, such as the words in this phrase, to acquire context and subsequently meaning.

Transformers use positional encoders to tag data elements coming in and out of the network.These tags are followed by attention units, which calculate a sort of algebraic map showing how each element connects to the others.

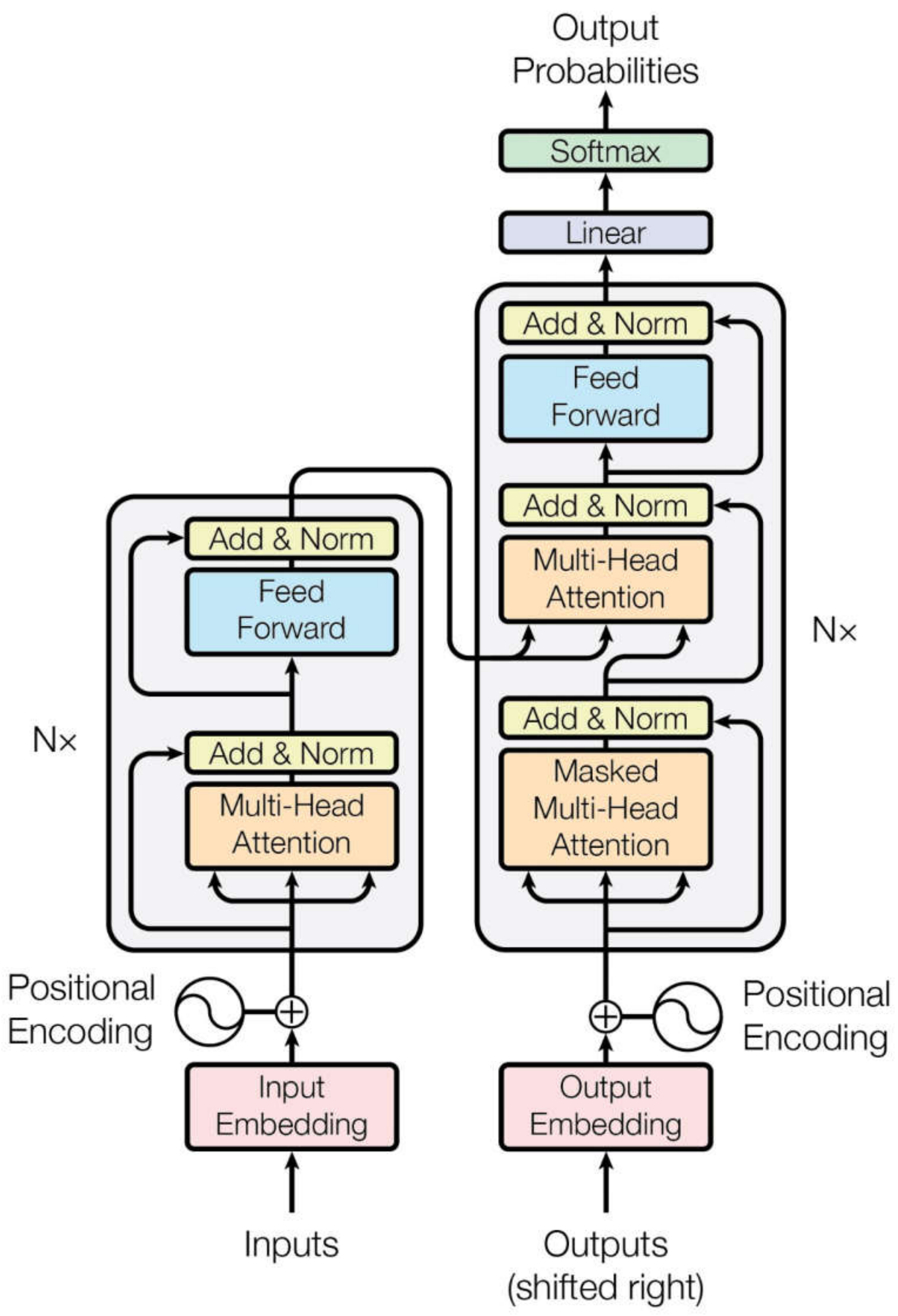


Figure 1.10: The Transformer architecture The Transformer model, like the Sequence-to-Sequence (seq2seq) machine translation paradigm, is built on an encoder-decoder architecture.However, the Transformer differs from the seq2seq model in three ways:

- Transformer Block: The recurrent layer in seq2seq is replaced by a Transformer Block. This block contains a multi-head attention layer and a network with two Position-Wise Feed-Forward network layers for the encoder. Another multi-head attention layer is used to compute the encoder state for the decoder.

- Add Norm: The inputs and outputs of both the multi-head attention layer and the Position-Wise Feed-Forward network are processed by two Add Norm layers which contain a residual structure and a layer normalization layer.

- Position Encoding: Since the self-attention layer does not distinguish the order of items in a given sequence, a positional encoding layer is used to add sequential information into each sequence item.

1.8 Transformer Functioning

Data preparation, model training, and model prediction are the Transformer’s three main tasks; Data preprocessing, model training and model prediction.

1.8.1 Data Preprocessing

The data is preprocessed using tokenizers before being fed into the Transformer model.Tokenization of the inputs is followed by the conversion of the created tokens into the token IDs required by the model.Tokenizers, for instance, are created for PyTorch by using the "AutoTokenizer from pretrained" function to:

- Get tokenizers that correspond to pretrained models in a one-to-one mapping.

- Download the token vocabulary that the model needs when using the model’s specific tokenizer.

1.8.2 Model Training

A well-liked training technique for neural machine translation is teacher forcing. It cuts down on training time by using the actual output as inputs rather than the anticipated output from the prior timestamp.

1.8.3 Model Prediction

- The encoder encodes the input sentence of the source language.

- The decoder uses the code generated by the encoder and the start token () of the sentence to predict the model.

- At each decoder time step, the predicted token from the previous time step is fed into the decoder as an input, in order to predict the output sequence token by token. When the end-of-sequence token () is predicted, the prediction of the output sequence is complete.

1.9 Sequence To Sequence Model (Seq2Seq)

Sequence-to-Sequence (or Seq2Seq) is a neural net that transforms a given sequence of elements, such as the sequence of words in a sentence, into another sequence.

Sequence transformation is the process used by Seq2seq to change one sequence into another. To get over the vanishing gradient issue, it uses a recurrent neural network (RNN), or more frequently an LSTM or GRU. The output from the preceding stage serves as the context for each item. One encoder and one decoder network make up the main parts. Each item is converted by the encoder into a matching hidden vector that includes both the object and its context. Using the previous output as the input context, the decoder reverses the process and produces the vector as an output item.

Seq2Seq models are particularly good at translation, where the sequence of words from one language is transformed into a sequence of different words in another language. A popular choice for this type of model is Long-Short-TermMemory (LSTM)-based models. With sequence-dependent data, the LSTM modules can give meaning to the sequence while remembering (or forgetting) the parts it finds important (or unimportant). Sentences, for example, are sequencedependent since the order of the words is crucial for understanding the sentence. LSTM are a natural choice for this type of data.

Encoder and Decoder are the main components of Seq2Seq models. The input sequence is translated by the encoder into an n-dimensional vector in a higher level space. The Decoder receives this abstract vector and converts it into an output sequence. The output sequence might consist of symbols, another language, a duplicate of the input, etc.

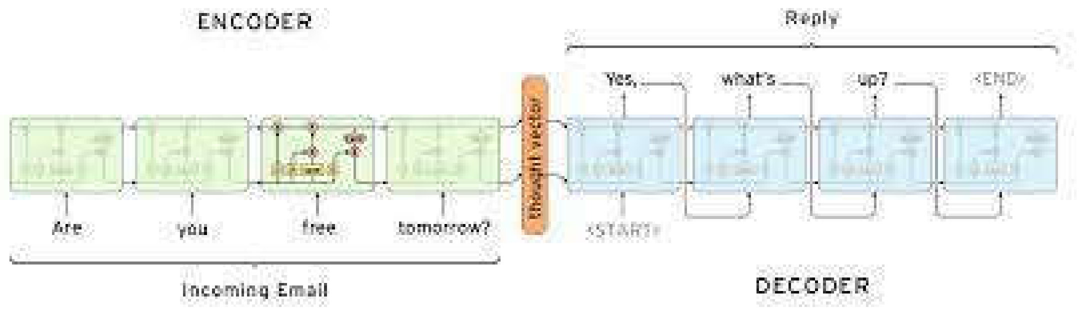


Figure 1.11: sequence to sequence architecture, Blue and green boxes are LSTM
cells

1.9.1 Encoder - Decoder Architecture

Encoder-decoder architecture is a type of neural network architecture that is best suited for use cases like machine translation where the input is a series of data and the output is another sequence of data. In other words, sequence-to-sequence modeling is best suited to encoder-decoder architecture. The initial purpose of the encoder-decoder design was to address the issue of machine translation, which entails translating text from one language to another. The major difficulty in machine translation is that it is challenging to directly map the input to the output since the input and output sequences have distinct lengths and structures.

The input data is initially passed through an encoder network in this configuration. The encoder network converts the input data into a numerical form that extracts the crucial information. The hidden state is another name for the numerical representation of the input data. The decoder network is subsequently given with the numerical representation (hidden state). One output sequence element is produced by the decoder network at a time to produce the output. The encoder decoder architecture is shown in the illustration below. As seen in the image below, data sequences for both the input and output might have different lengths.

RNN/LSTM used as an Encoder and a Decoder: For applications like machine translation, where the input and output are both collections of words with different lengths, this architecture can be employed. While the RNN/LSTM in the decoder may produce the appropriate output sequence of words in a different language, the RNN/LSTM in the encoder can encode the input sequence of words into a hidden state or numerical representation. The encoder-decoder architecture shown below uses RNN in both the encoder and the decoder networks. English words are used as the input sequence, and German is translated automatically.

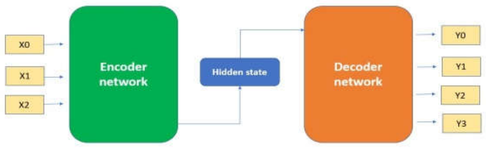


Figure 1.12: encoder-decoder- architecture

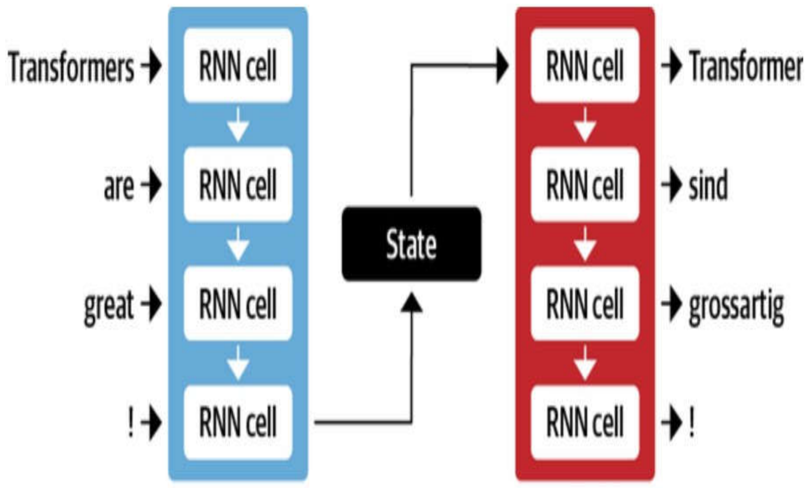


Figure 1.13: encoder-decoder- architecture RNN

1.10 Conclusion

The foundational information required to understand the ideas of neural machine translation, recurrent neural networks (RNNs), long short-term memory (LSTM) networks, and sequence-to-sequence (seq2seq) models has been covered in this chapter. For the upcoming investigation of comment creation from source code using these methodologies, it is essential to have a firm grasp of these ideas.

Chapter 2

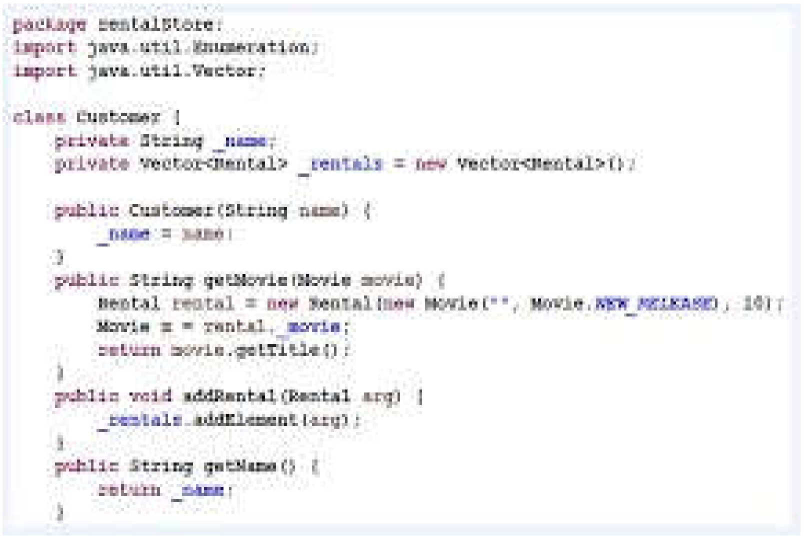
Code Source Analysis and Documentation

2.1 Introduction

AST tools and code source documentation both aid in the comprehension, evaluation, and advancement of software projects. Developers can understand and edit code with the help of documentation, while AST tools provide insights and capabilities for sophisticated program analysis and manipulation. In this chapter, we focus on Source code and especially on documentation source code, analysis source code, and the categories of comments in Java code. we talk also about the Abstract Syntax tree, AST tools, and AST application, we describe traversed algorithms (structure-based traversal and graph traversal algorithm).

2.2 Code Source

Source code is a text that represents the instructions that must be executed by a microprocessor. The source code is often materialized in the form of a set of text files. The source code is usually written in a programming language, thus enabling better human understanding. Once the source code is written, it generates a binary representation of a sequence of instructions executable by a microprocessor. for example of source code 3.1



```
package rentalStore;
import java.util.Enumeration;
import java.util.Vector;

class Customer {
    private String _name;
    private Vector<Rental> _rentals = new Vector<Rental>();

    public Customer(String name) {
        _name = name;
    }
    public String getMovie(Movie movie) {
        Rental rental = new Rental(new Movie("", Movie.NEW_RELEASE), 10);
        Movie m = rental._movie;
        return movie.getTitle();
    }
    public void addRental(Rental arg) {
        _rentals.addElement(arg);
    }
    public String getName() {
        return _name;
    }
```

Figure 2.1: java code source example

2.2.1 Code Statement Level

Software is a collection of programs with documentation. A program is a sequence of code statements. Many researchers have proposed comment-generation methods at the code block level. We can classify it further into two categories. The first is the single code statement level (pseudo code generation), and the second, the multiple code statement level.[5]

2.3 Code Source Documentation

Source code documentation is a type of technical document that is embedded within the source code itself. Software developers write source code documentation at various levels of abstraction. Developers require source code documents to understand the code during the maintenance phase.

Code Documentation By using comments, annotations, and detailed justifications, the code may be documented. Future developers will benefit from having a better understanding of the functions and purposes of various code parts.

in this section describes in a very synthesized manner the rules concerning the documentation of the code Programming in C++ and Java. This document also provides guidelines for Formatting the source code .

2.3.1 General Principles

- The documentation is intended for readers and users of a program, not not to its author (except 6 months after writing it; the author then no longer remembers Good for what he did.

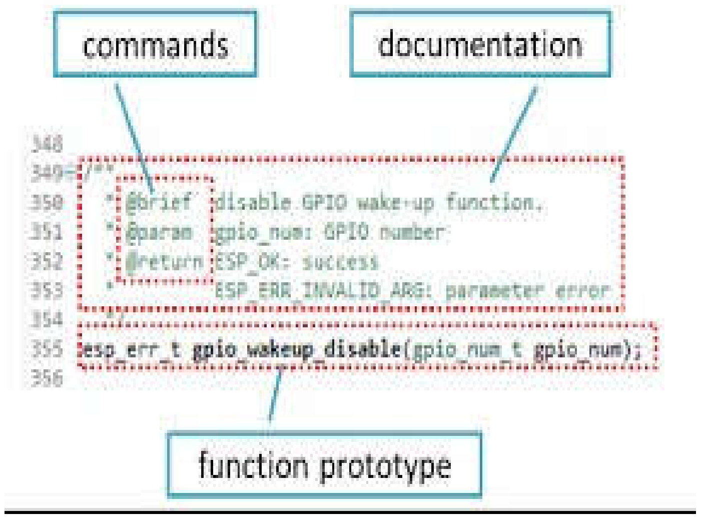
- It helps the reader understand the source code and use it; it does not duplicate information already explicitly contained in the source code.

- It must be clear, concise, accurate, complete, uniform and simple to maintain.

- It must be done as the program is developed, otherwise it It will never be done (correctement).

- It facilitates the maintenance of programs in an organization. programming style, which allows a person to It’s easier to get another’s job done.

Keep in mind that documentation is a continuous operation. It’s crucial to update the documentation when the code changes and new features are introduced in order to keep it accurate and helpful.



```
/**
 * @brief  disable GPIO wake-up function.
 * @param  gpio_num: GPIO number
 * @return ESP_OK: success
 *         ESP_ERR_INVALID_ARG: parameter error
 */
esp_err_t gpio_wakeup_disable(gpio_num_t gpio_num);
```

Figure 2.2: Code Source Documentation

Software development must include code source documentation because it promotes better knowledge, maintainability, and developer cooperation. Here are some important considerations for source code [6]documentation:

1. Readme Files:Include a readme file that gives a summary of the codebase, installation guidelines, a getting started guide, and links to pertinent documentation or resources at the project level.

2. Code Dependencies:Identify the external frameworks, libraries, or modules that the code uses. To guarantee correct configuration and setup while deploying or working on the codebase, note the versions and dependencies.

3. Code Style and Standards:Make that the code adheres to accepted best practices and coding standards. Record any departures from the norms and offer suggestions for bringing the code into line with the desired aesthetic.

4. Documentation Format:Establish the documentation for the code’s format. Inline comments, a separate documentation file, or an outside documentation tool like Doxygen or Javadoc are all possible forms of documentation.

5. Purpose and Overview:Give a brief explanation of the functionality and goal of the code. Describe its key characteristics, planned applications, and integration with the system’s general design.

6. Module/Class/Function Level Documentation: Clearly describe the roles, inputs, outputs, and any pertinent side effects for each module, class, and function. Specify the desired actions and usage instructions, along with the parameters' descriptions and return values.

2.4 Comments

2.4.1 Comments and Inline Documentation

Use inline comments within the code to explain complex or non-obvious sections, algorithms, or decisions. Comment key variables, constants, or logic blocks to provide additional context. However, strive for self-explanatory code and avoid excessive or redundant comments.

2.4.2 Comments Categories

For the Java and C/C++ programming languages, we differentiate between seven different types of comments:

- Copyright Comments: include information about the copyright or the license of the source code file. They are usually found at the beginning of each file.

- Header Comments: give an overview about the functionality of the class and provide information about, e. g.,the class author, the revision number, or the peer review status. In Java, headers are found after the imports but before the class declaration.

- Member Comments :describe the functionality of amethod/field, being located either before or in the same line as the member definition. They provide information for the developer and for a project's API.

- Inline Comments :describe implementation decisions within a method body.

- Section Comments : address several methods/fields together belonging to the same functional aspect. A fictitious example looks like// —- Getter and Setter Methods —and is followed by numerous getter and setter methods.

- Code Comments: contain commented out code which is source code ignored by the compiler. Often code is temporarily commented out for debugging purposes or for potential later reuse.

- Task Comments : are a developer note containing a remaining todo, a note a about a bug that needs to be fixed,or a remark about an implementation hack.

2.5 Code Source Analysis

Static code analysis, in which the source code is examined just as code and the program is not in use, is the same as source code analysis. This eliminates the requirement for developing and utilizing test cases and may help it avoid featurespecific issues like buttons that aren't the color that the specs specify. It focuses on identifying programming errors such lines of code that might cause crashes that could be harmful to the program's ability to run properly.[7]

Static code analysis, sometimes referred to as code source analysis or source code review, is the act of looking at the source code of a software program to find potential problems, weaknesses, or enhancements. It entails examining the code's structure, grammar, and logic without actually running it.

Coding standards compliance, bug detection, and security vulnerability identification are all goals of code source analysis. It aids in the early identification of possible problems by developers, enabling them to address them before they become difficulties during runtime.

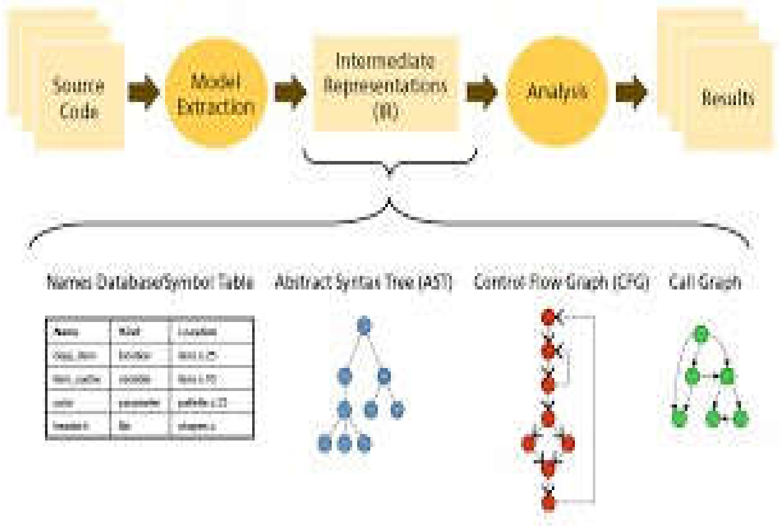

Source Code
Model Extraction
Intermediate Representations (IR)
Analysis
Results
Names Database/Symbol Table
Abstract Syntax Tree (AST)
Control Flow Graph (CFG)
Call Graph

Figure 2.3: how static analysis work

2.5.1 Analysis of Source code

Various methods and instruments can be used to study the code during source code analysis[8], including:

- Code Review: To begin, read through the source code to become familiar with its organization, structure, and features. Identify the key classes, modules, and components that make up the codebase.

- Code Flow:Trace the execution route to analyze the code flow and find any potential bottlenecks, dependencies, or performance problems. Record the high-level overview of the code's operation and the relationships between its many components.

- Error Handling::Check the code's handling of exceptions, errors, and edge situations. Any error-handling procedures that are in place should be documented, along with how they affect the code's overall behavior.

- Security Analysis: Examine the system for potential security flaws like SQL injections, cross-site scripting (XSS), or weak authentication and authorisation systems. Keep a record of any hazards found and make suggestions for improvements or mitigations.

- Performance Analysis:Check the code for any performance issues or potential improvements. Record any areas where performance, memory consumption, or resource utilization might be improved.

- Testing and Test Coverage:Check the code to see whether it has any unit tests, integration tests, or other automated tests. Examine the code coverage and note any testing strategy shortcomings.

- Automated tools: specialized software tools made to check source code for compliance with coding standards, common programming errors, and security flaws. These tools use pre-established rules or patterns to examine the code.

- Metrics analysis:To evaluate the quality and maintainability of the code, code metrics including complexity, cyclomatic complexity, duplication, and other software metrics are analyzed.

Developers can find possible problems, minimize defects, increase security, optimize speed, and improve code quality by doing code source analysis. For software developers to create trustworthy and durable applications, this technique is crucial.

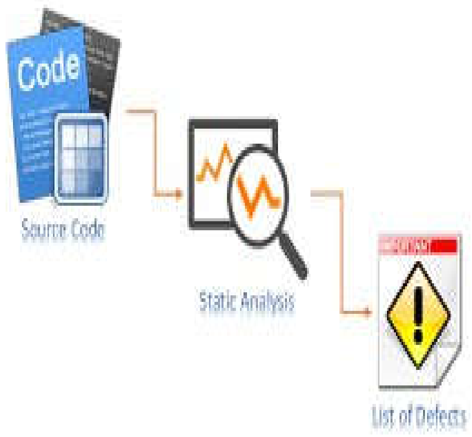


Figure 2.4: analysis code source

2.6 Structure and Necessary Elements for source code

2.6.1 Source File

Each source file contains an entity consisting of the elements[6] The following:

- Name of the program
- A brief description
- the authors and their registrations;
- the date of completion of the first implementation
- the end date of each version with authors and authors;
- the course for which the file is developed;
- a longer description containing:

  a- program entries and their preconditions; b- the

  outputs of the program and their post-conditions.
- A history of changes for each version.

2.6.2 Method and Function

Each method and function:

- includes an entrance describing the processing carried out; ideally, describe the Pre-entry conditions and post-exit conditions;

- includes, if necessary, comments in the body that explain the sections the complexity of the treatment;

2.6.3 Identifying

- Use of Significant Identificators a- The names of classes, variables and constants must be accurate. b- A function or method name describes the treatment performed. c- Indices in an iteration can be simple letters.
- Construction of identifiers

2.7 Abstract Syntax Tree

The structure of a program or a piece of code is represented by an abstract syntax tree (AST), a data structure used in computer science and programming language theory. It is a generic representation of the syntax of the code, unrelated to any particular computer language. When a program is parsed, its syntactic components, such as statements, expressions, operators, and control flow structures, are separated from the source code and organized into a hierarchical structure. The standard visual representation of this hierarchical structure is a tree, where each node corresponds to a syntactic element and the edges show the connections between these components.

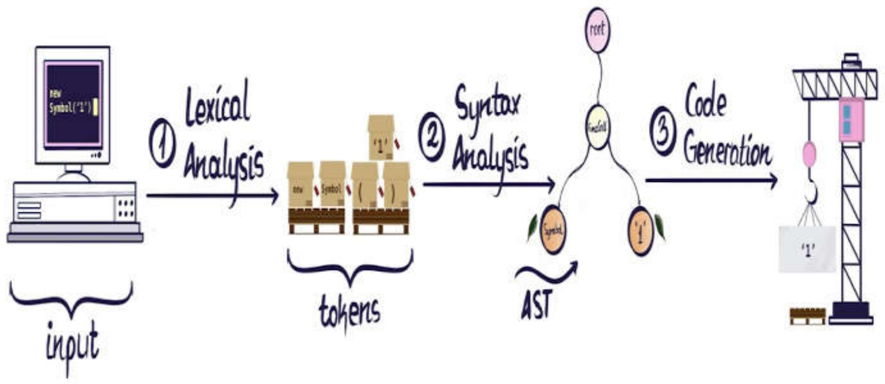


Figure 2.5: Introduction to Abstract Syntax Trees

While abstracting away elements like style, whitespace, and other non-essential information, the AST preserves the code’s core structure and meaning. Compared to the raw source code, it offers a higher-level representation that is simpler to understand and manipulate. ASTs are frequently used in tools for code restructuring, interpreters, compilers, and other stages of program analysis and transformation. By navigating and modifying the tree nodes, they make it possible to perform actions like syntax checking, semantic analysis, optimization, code creation, and program modifications.[9]

Tools and frameworks may support several programming languages without having to worry about the nuances of each language's particular syntax by working with code in a language-agnostic way utilizing an AST.

An Abstract Syntax Tree (AST) is a hierarchical representation of the syntactic structure of a program written in a programming language. It captures the structure and relationships among the various components of the code,

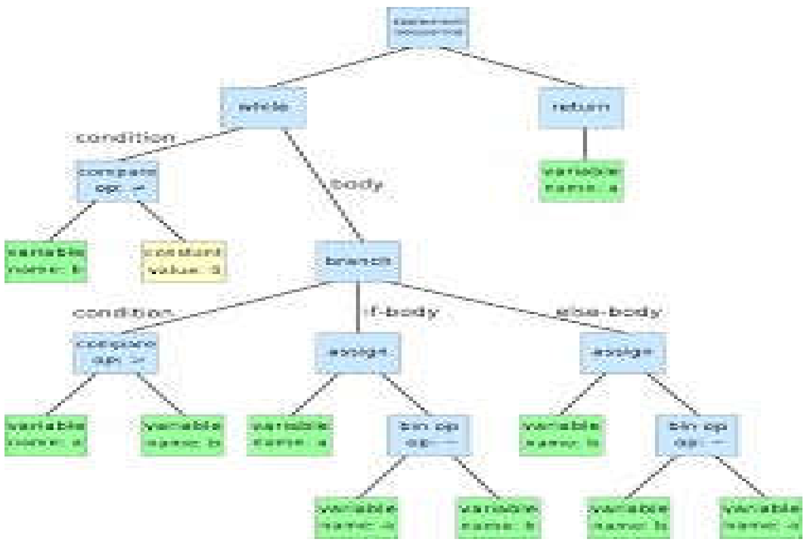


Figure 2.6: Abstract Syntex Tree Example

When a compiler or interpreter processes source code, it typically goes through several stages, including lexical analysis (breaking code into tokens), parsing (building a parse tree), and finally constructing an AST. The AST represents the parsed code in a more abstract and structured form, which makes it easier for subsequent stages of the compilation process, such as type checking, optimization, or code generation.

Here are some key points about Abstract Syntax Trees:

1. Hierarchical Structure:An AST is a structure that resembles a tree, with each node standing in for a particular piece of code, such as a function declaration, a loop statement, or an arithmetic expression. The layering and composition of code components are represented by the parent-child connections connecting the nodes.

2. Abstracted Representation: The AST concentrates on the underlying structure and meaning by abstracting away some of the source code's

specifics. For instance, the AST may represent a loop construct without explicitly describing how it is implemented or collapse many levels of parentheses in an arithmetic statement.

3. Language-Dependent: Because the organization of code components and linguistic requirements differs, every programming language has its own AST representation. The exact syntax and semantics of the language it represents are captured by the AST.

4. Loss of Certain Details:An AST may lose certain syntactic characteristics, such as formatting, comments, or original variable names, but it still retains the fundamental structure and meaning of the code. These specifics can be skipped over during parsing because they are frequently not necessary for later compilation steps.

5. Analysis and Transformation:ASTs are frequently used for a variety of compiler-related activities, including code creation, optimization, and analysis. Compilers may do static analysis using the AST, spot possible mistakes, apply optimizations, and produce effective machine code.

6. Tooling and Language Tools:Development tools like IDEs, linters, and refactoring tools also use ASTs. These tools make use of the AST to offer capabilities like automatic refactorings, code navigation, code recommendations, and static code analysis.

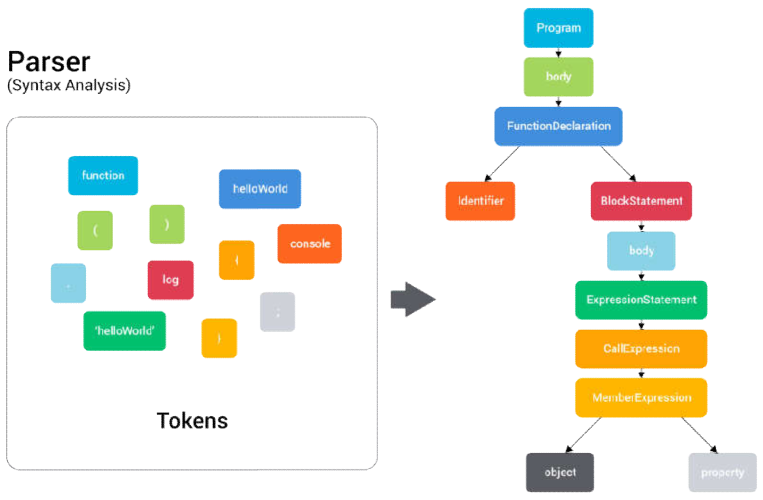


Figure 2.7: Abstract Syntex Tree Tooling

There are several tools available for working with Abstract Syntax Trees (ASTs) in various programming languages. Here are some popular AST tools:

2.7.1 AST Tools

Some industrial and academic AST tools are listed bellow.

- Tree-sitter: A parsing package called Tree-sitter produces ASTs for several programming languages. It offers a single API and facilitates effective incremental parsing. Tree-sitter is a flexible option because it offers bindings for several languages.

- Esprima and Babel: Esprima is a parser for JavaScript code that creates ASTs. Esprima is an internal tool used by the JavaScript compiler Babel to create ASTs. For the analysis and manipulation of JavaScript code, these tools are frequently used.

- Roslyn: Microsoft's Roslyn platform is an open-source compiler for C and VB.NET. It has APIs for AST interaction, code analysis, and transformation. Roslyn offers comprehensive assistance for working with and analyzing C and VB.NET code.

- Eclipse JDT:Java Development Tools (Eclipse JDT) are a collection of libraries and APIs for working with Java programming. It offers assistance in constructing ASTs and parsing Java source code. JDT offers a number of tools for examining and modifying Java code.

- Python ast module:The 'ast' module of Python's standard library offers tools for interacting with Python ASTs. It enables the parsing of Python source code, the creation of ASTs, and the exploration and analysis of those ASTs.

- Clang and LibTooling: The Clang frontend for the C/C++ compiler offers a robust framework for dealing with C/C++ code. It contains LibTooling, which enables the creation of unique tools for the use of ASTs in the analysis and transformation of C/C++ code.

- TypeScript Compiler API: TypeScript, a superset of JavaScript, has a Compiler API that enables parsing TypeScript code and generating ASTs. The API allows working with TypeScript ASTs to perform various code analysis and transformation tasks.

These are only a few illustrations of AST tools that are accessible for various programming languages. There may be more libraries, frameworks, or languagespecific tools that offer AST capabilities, depending on the language and particular needs.

2.7.2 AST Design and Construction

• AST Nodes must hold sufficient information to recall the essential elements of the program fragments they represent – For example, want to know that node X of the AST corresponds directly to procedure Foo of the program • Implementation of ASTs should be decoupled from what is represented – Accessors are used to hide a node's internal representation • No single class hierarchy – Each phase may view the nodes differently

2.7.3 Stages of Creating an AST

To transfer the code to an AST there are different stages that need to be done before creating an AST which are Lexical Analysis and Syntax Analysis.

1. Lexical Analysis : In this stage, we will firstly determine each of the token types that make up the source code. It is also called Tokenization, which tokenizes the source code into smaller units.

2. Syntax Analysis : The next step is syntax analysis, also known as a parser. This process creates the tree structure using the tokens produced by the lexical analyzer.

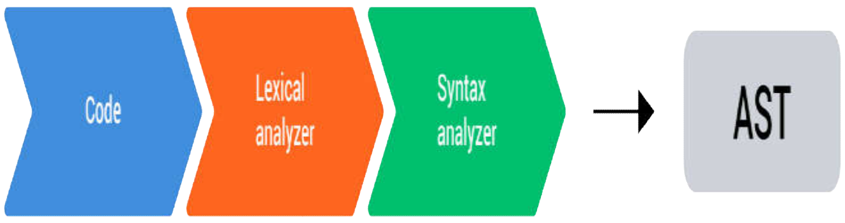


Figure 2.8: The Different stages of creating an AST.

2.7.4 Application of Abstract Syntax Trees

Abstract Syntax Trees (ASTs) are commonly used in programming language theory, compilers, interpreters, and various static analysis tools. Here are a few application areas in which ASTs are utilized:

- Compilation :A computer language's source code is converted into an executable format during the compilation phase. At the heart of this shift are ASTs. An AST, which encapsulates the structure and semantics of the code, is created by parsing the source code. Then, further steps like optimization, code creation, and linking make use of the AST.

- Interpretation : In interpreted languages, an interpreter immediately processes the AST to carry out the program rather than converting source code into machine code. The program may be run step-by-step because the interpreter moves through the AST and carries out the tasks listed by each node.

- Program analysis and optimization : ASTs give the code a structured form that enables different analysis and optimizations. The AST may be examined by static analysis tools like linters and static analyzers to look for possible mistakes, security holes, coding style infractions, and other problems. Based on the knowledge obtained from the AST, optimizations such as dead code reduction, constant propagation, and loop modifications can be done.

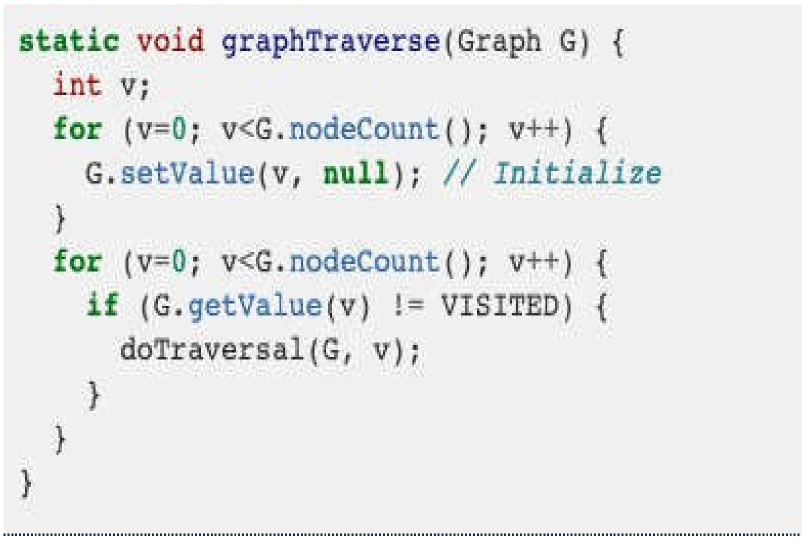
- Code refactoring and transformation: Code may be automatically refactored or transformed using ASTs. Integrated development environments (IDEs) and similar tools frequently use ASTs to carry out automatic refactorings such renaming variables, removing methods, or rearranging code structure. Developers can enhance the readability, maintainability, and general quality of their code by using AST-based modifications.

- Domain-specific language implementation : ASTs offer a practical form for capturing the domain-specific semantics while developing domain-specific languages. Developers can design customized languages targeted to certain problem areas by specifying the syntactic rules of the DSL, parsing the code into an AST, then interpreting or translating it accordingly.

These are only a handful of applications for ASTs in programming. A variety of operations including analysis, transformation, and interpretation are made possible by ASTs, which act as an intermediary representation of code.

2.8 Graph Traversal Algorithms

A graph's nodes or vertices can be explored or visited using graph traversal methods. By starting from one node and visiting every accessible node, they provide systematic traversal or navigation over the graph. Because they make it easier to comprehend the structure and interactions inside the network, traversal algorithms are crucial for understanding and modifying graphs. Graph traversal algorithms major objective is to efficiently cover



```
static void graphTraverse(Graph G) {
  int v;
  for (v=0; v<G.nodeCount(); v++) {
    G.setValue(v, null); // Initialize
  }
  for (v=0; v<G.nodeCount(); v++) {
    if (G.getValue(v) != VISITED) {
      doTraversal(G, v);
    }
  }
}
```

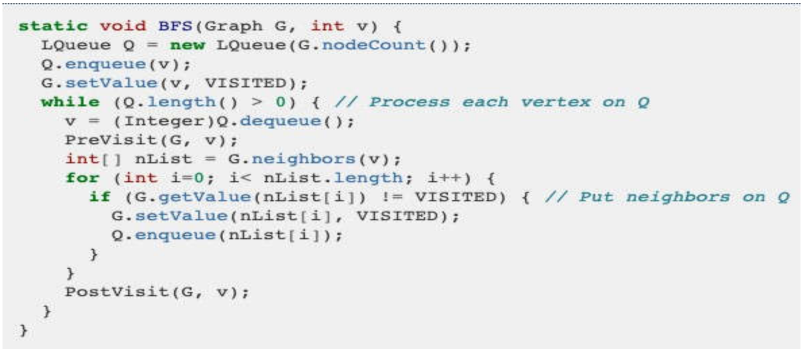
Figure 2.9: Graph Traversal Algorithm

all of the nodes in the graph by ensuring that each node is visited precisely once, avoiding endless loops, and completing the graph. BreadthFirst Search (BFS) and Depth-First Search (DFS) are two popular graph traversal techniques.

2.8.1 Breadth-First Search (BFS)

BFS begins at a particular source node and traverses the graph level by level. Prior to going on to their neighbors, it first visits all of the neighbors of the source node. To keep track of the nodes that need to be visited,

this method employs a queue data structure. BFS ensures that nodes that are closer to the source node are accessed before nodes that are farther away. It is helpful for examining a graph breadth-first and for determining the shortest path between two nodes.



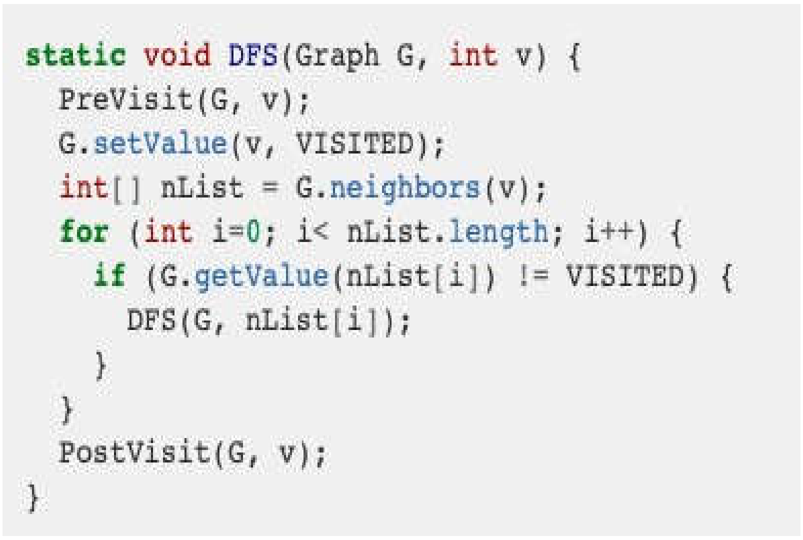
```
static void BFS(Graph G, int v) {
  LQueue Q = new LQueue(G.nodeCount());
  Q.enqueue(v);
  G.setValue(v, VISITED);
  while (Q.length() > 0) { // Process each vertex on Q
    v = (Integer)Q.dequeue();
    PreVisit(G, v);
    int[] nList = G.neighbors(v);
    for (int i=0; i< nList.length; i++) {
      if (G.getValue(nList[i]) != VISITED) { // Put neighbors on Q
        G.setValue(nList[i], VISITED);
        Q.enqueue(nList[i]);
      }
    }
    PostVisit(G, v);
  }
}
```

Figure 2.10: Breadth-First Search algorithm

2.8.2 Depth-First Search (DFS)

:

DFS investigates the graph by delving as far as feasible down each branch and then turning around. It travels to one of its neighbors after starting from a certain source node. It then moves on to that neighbor's subsequent unexplored neighbor and so forth until it comes to a dead end. Then it circles back to the first node and investigates a different unexplored neighbor. Up till all nodes have been visited, this procedure keeps on. DFS often keeps track of the nodes to be visited using a stack data structure (or recursion). It is helpful for discovering cycles in a graph and for exploring all potential pathways. Other specialized graph traversal methods exist in addition to BFS and DFS, such Dijkstra's algorithm for locating the shortest path in weighted graphs and A* search algorithm for locating the shortest path with heuristic information. Based on particular issue requirements, these algorithms use extra factors and heuristics to enhance the traversal process.

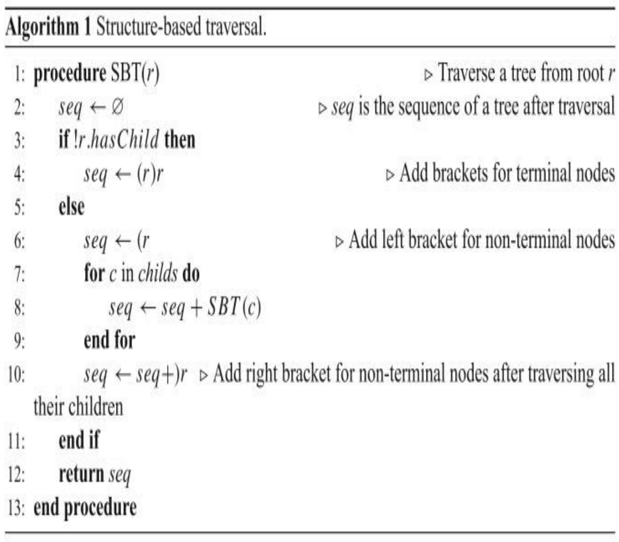


```
static void DFS(Graph G, int v) {
  PreVisit(G, v);
  G.setValue(v, VISITED);
  int[] nList = G.neighbors(v);
  for (int i=0; i< nList.length; i++) {
    if (G.getValue(nList[i]) != VISITED) {
      DFS(G, nList[i]);
    }
  }
  PostVisit(G, v);
}
```

Figure 2.11: Depth-First Search algorithm

2.9 Structure Based Traversal

The analysis and processing of tree-like structures, such as abstract syntax trees (ASTs) in programming languages, are done using the Structure-Based Traversal (SBT) technique. To carry out numerous tasks, like program analysis, code modification, or code generation, SBT tries to capture the structural relationships and attributes of nodes within a tree.[10]



Algorithm 1 Structure-based traversal.

```
procedure SBT($r$)                        ▷ Traverse a tree from root $r$
    $seq \leftarrow \varnothing$          ▷ $seq$ is the sequence of a tree after traversal
    if !$r.hasChild$ then
        $seq \leftarrow (r)r$             ▷ Add brackets for terminal nodes
    else
        $seq \leftarrow (r$               ▷ Add left bracket for non-terminal nodes
        for $c$ in $childs$ do
            $seq \leftarrow seq + SBT(c)$
        end for
        $seq \leftarrow seq + )r$         ▷ Add right bracket for non-terminal nodes after traversing all their children
    end if
    return $seq$
end procedure
```

Figure 2.12: SBT Algorithm

To visit and process nodes in the AST, traversal rules or patterns must be established as part of the SBT approach. These guidelines are based on the node types, parent-child relationships, sibling relationships, and other structural elements of the tree. SBT algorithms explore the AST in an organized manner by abiding by these principles, visiting nodes in accordance with their locations and connections within the tree.

SBT is essential in code generation tasks, where an AST is translated into executable code or another representation.Code generation methods can either produce code based on predetermined templates or map the tree nodes to the relevant code constructs in the target programming language by traversing the AST according to predefined patterns.

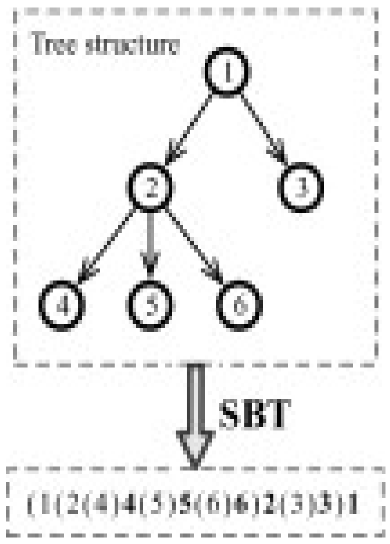


Figure 2.13: Example an AST to SBT

2.10 Conclusion

In Conclusion, the purpose of comprehending and maintaining software projects, code source documentation is crucial. It gives details about the function, usage, and intent behind certain code elements. However, program analysis and modification greatly benefit from the use of an Abstract Syntax Tree (AST) tool. The grammar and semantics of the code are represented in an organized manner by AST tools, which improve the productivity and efficiency of software development.

Chapter 3

NMT-based Comment Generation from Source Code

3.1 Introduction

For big software systems, the maintenance phase typically lasts longer than all the earlier life-cycle stages combined [2]. The maintenance cycle includes understanding software as a major component. In fact, knowing the code base takes up around 59% of the total time spent during the software development life cycle. This is due to project deadlines and time restrictions, which causes mismatched code comments, full omissions of comments, or failures to update comments of patched code in the database [11]. As a result, there has been extensive research on the automatic generation of comments given uncommented source code. Some of the early research on this topic conducted by [12] [13] [14]

In this chapter, we present the essence of our work. First, we give an overview of related work on generating comments from source codes. Then we introduce our proposition by explaining, and after that, we will present the process we followed and provide a comprehensive and detailed description of our approach.

3.2 Related work

in the literature, many notable approaches have been proposed to generate code comments automatically from source code. Different methods and techniques have been used, from clone detection to copy comments [6] to deep learning to forecast when new comments will be generated for uncommented code [11].

Code summarization research is a related area of study. Code summary aims to condense lengthy chunks of code into a manageable number of words. Using bidirectional LSTMs, such as in code2seq [15], which uses a bidirectional LSTM in conjunction with an attention mechanism to highlight the most crucial nodes in an abstract syntax tree (AST), significant progress has been made in this area. The function is then summarized using these ideas.

The function is then summarized using these ideas. As a result, the code's actions are briefly summarized. Of course, these summaries differ significantly from good comments.

An encoder-decoder architecture often employs 2 related models. who initially enters an encoded state from the input sequence. The decoder network, which has been trained to translate its input into a desired target, is then given this state. Because it enables the learning of translations across many types of sequences, this technique is frequently employed in neural machine translation (NMT).

Previous studies have demonstrated that using deep learning models to automatically generate code comments is a practical application [11] [16]. These studies train a sequence2sequence model (encoder-decoder architecture) using syntactical information stored in an AST. The comment that would describe a line of code is then predicted from a group of comments using this model. The BLEU score is used to compare the performance of these models. [17] This score serves as a gauge for machine translation accuracy. The models in papers [11] [16] received BLEU-4 scores of 38% and 39 %, respectively. Because the use of neural networks in software engineering is a rapidly progressing field, the authors of the paper (code2vec) upon which deepcomm is based have released a new paper boasting higher accuracy (code2seq). To improve upon the deepcomm paper we suggest a similar architecture, while using code2seq.

3.3 An approach for Comment generation from java source code

in this work, we propose a comment generator model using a new state-ofthe-art technique that is based on code2seq model for comment generation from Java source code. Our proposition is based on the DeepCom [11] architecture. Instead of directly generate comments from the traversed AST sequences. we propose to combine the source code and the modified AST sequences together to generate the comments. We focus on replicating the code2seq model with added capabilities such as, predicting natural language and modified ASTs.

3.3.1 Model design

We will employ a code to sequence model to automatically produce comments. It has been demonstrated that this is a successful method for NMT[18], code summarization[15] and comment generation [11] [16]. In this part, the network's overall architecture will be covered, starting with a dataset analysis, followed by AST processing, input code encoding, and finally decoding.

The propsoed model, shown in Figure 3.1 adopts an attention-based Code2Seq model to generate high-level description of code snippets. Compared to deepcomm, the BLEU score of Code2Seq model increases to 35.5on the Java corpus. In order to capture the structural information, AST sequences traversed by SBT are input into our model.

Fig 3.1. provides an illustration of the general structure of the proposed work. Three phases are required: data processing, model training, and testing. We analyze and preprocess the source code we downloaded from GitHub into a parallel corpus of Java methods and the comments that go with them. The Java methods are turned into AST sequences via a particular traversal methodology in order to understand the structural information.

The main propositions in this method are :

– Our method uses AST sequences as an input to obtain structural data. We suggest a new method for analyzing ASTs called SBT, which provides unambiguity while expressing semantics with structural infor-

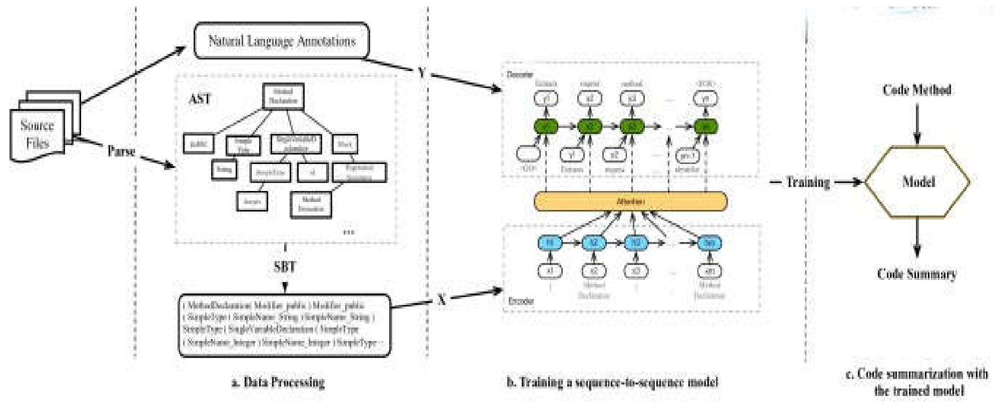

Natural Language Annotations
Source Files
Parse
AST
SBT
Y
X
Decoder
Attention
Encoder
Training
Code Method
Model
Code Summary
a. Data Processing
b. Training a sequence-to-sequence model
c. Code summarization with the trained model

Figure 3.1: General Process of the proposed approach

mation. By better aligning source code to the natural language description, our model can generate comment that are more accurate.

3.3.2 Out of Vocabulary problem

The Out of Vocabulary issue in NMT is well-known, especially when invented terms are employed, like in the name of variables in source code. This is not an issue when translating from one language to another if the majority of terms have been defined and trained upon by the model. However, because a (Bi)LSTM in a sequence2sequence model can only predict words it has been trained on, this issue is particularly common in comment creation. The way that different programmers name variables in their code might vary significantly, even within the same project [2]. A major obstacle for producing quality comments is presented by this. In this study, we use the AST to address this issue.

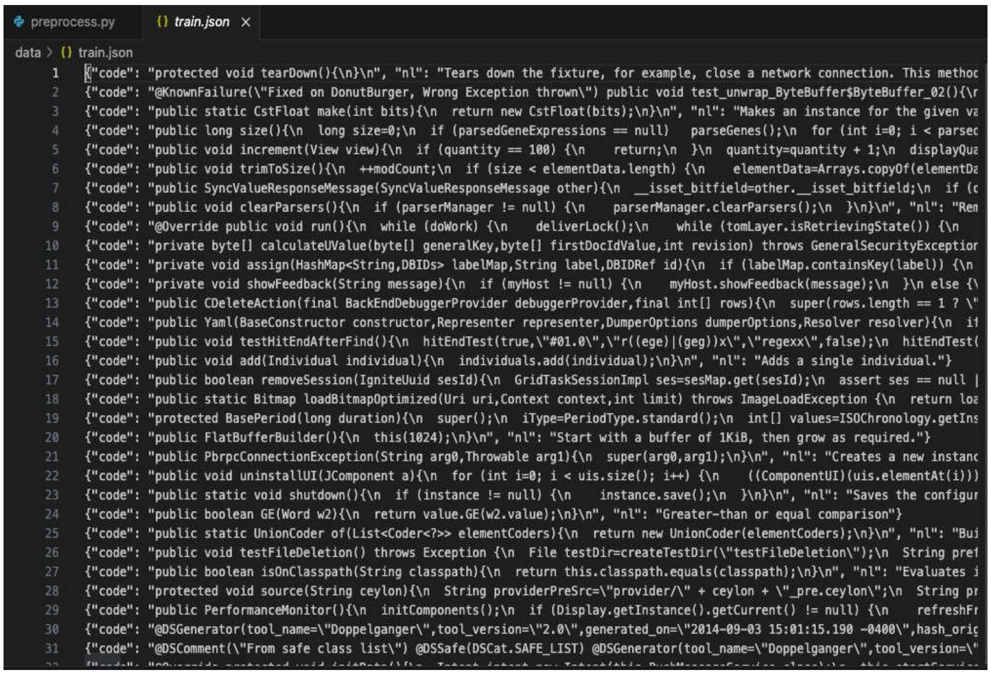
3.4 Dataset

The comment generator model is trained on the same dataset as Hu et al.'s deepcom network [11] in order to compare its capabilities with the stateof-the-art methods. To maintain a fair test, the same data split as in [11] will be utilized, which is 80% train, 10% test, and 10% validation. The dataset contains 588,108 code-comment pairs that can be used for training and evaluation. Details can be found in table 3.1

# Methods	# All tokens	# All identifiers	#Unique tokens	#Unique identifiers
588,108	44,378,497	13,779,297	794,711	794,621

Table 3.1: Statistics for code snippets in DeepComm dataset

The 9,714 Java projects that were scraped from GitHub with the [11] form the basis of the dataset.

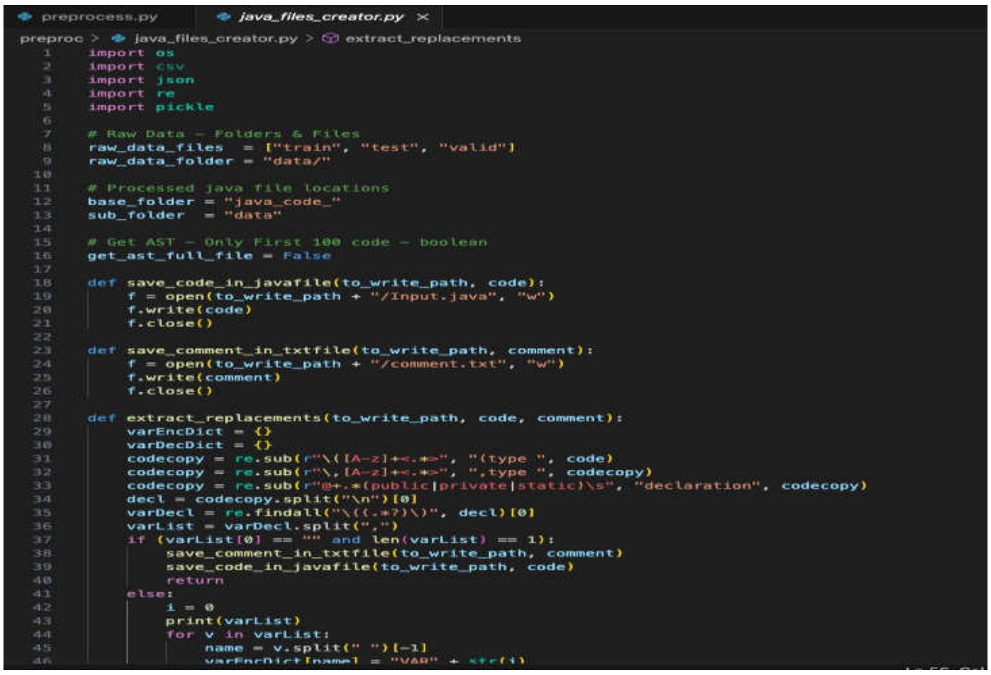


```
{"code": "protected void tearDown(){\n}\n", "nl": "Tears down the fixture, for example, close a network connection. This methoc
{"code": "@KnownFailure(\"Fixed on DonutBurger, Wrong Exception thrown\") public void test_unwrap_ByteBuffer$ByteBuffer_02(){\r
{"code": "public static CstFloat make(int bits){\n  return new CstFloat(bits);\n}\n", "nl": "Makes an instance for the given va
{"code": "public long size(){\n  long size=0;\n  if (parsedGeneExpressions == null)   parseGenes();\n  for (int i=0; i < parsec
{"code": "public void increment(View view){\n  if (quantity == 100) {\n    return;\n  }\n  quantity=quantity + 1;\n  displayQua
{"code": "public void trimToSize(){\n  ++modCount;\n  if (size < elementData.length) {\n    elementData=Arrays.copyOf(elementDa
{"code": "public SyncValueResponseMessage(SyncValueResponseMessage other){\n  __isset_bitfield=other.__isset_bitfield;\n  if (c
{"code": "public void clearParsers(){\n  if (parserManager != null) {\n    parserManager.clearParsers();\n  }\n}\n", "nl": "Rem
{"code": "@Override public void run(){\n  while (doWork) {\n    deliverLock();\n    while (tomLayer.isRetrievingState()) {\n
{"code": "private byte[] calculateUValue(byte[] generalKey,byte[] firstDocIdValue,int revision) throws GeneralSecurityExceptior
{"code": "private void assign(HashMap<String,DBIDs> labelMap,String label,DBIDRef id){\n  if (labelMap.containsKey(label)) {\n
{"code": "private void showFeedback(String message){\n  if (myHost != null) {\n    myHost.showFeedback(message);\n  }\n else {\
{"code": "public CDeleteAction(final BackEndDebuggerProvider debuggerProvider,final int[] rows){\n  super(rows.length == 1 ? \"
{"code": "public Yaml(BaseConstructor constructor,Representer representer,DumperOptions dumperOptions,Resolver resolver){\n  if
{"code": "public void testHitEndAfterFind(){\n  hitEndTest(true,\"#01.0\",\"r((ege)|(geg))x\",\"regexx\",false);\n  hitEndTest(
{"code": "public void add(Individual individual){\n  individuals.add(individual);\n}\n", "nl": "Adds a single individual."}
{"code": "public boolean removeSession(IgniteUuid sesId){\n  GridTaskSessionImpl ses=sesMap.get(sesId);\n  assert ses == null |
{"code": "public static Bitmap loadBitmapOptimized(Uri uri,Context context,int limit) throws ImageLoadException {\n  return loa
{"code": "protected BasePeriod(long duration){\n  super();\n  iType=PeriodType.standard();\n  int[] values=ISOChronology.getIns
{"code": "public FlatBufferBuilder(){\n  this(1024);\n}\n", "nl": "Start with a buffer of 1KiB, then grow as required."}
{"code": "public PbrpcConnectionException(String arg0,Throwable arg1){\n  super(arg0,arg1);\n}\n", "nl": "Creates a new instanc
{"code": "public void uninstallUI(JComponent a){\n  for (int i=0; i < uis.size(); i++) {\n    ((ComponentUI)(uis.elementAt(i)))
{"code": "public static void shutdown(){\n  if (instance != null) {\n    instance.save();\n  }\n}\n", "nl": "Saves the configur
{"code": "public boolean GE(Word w2){\n  return value.GE(w2.value);\n}\n", "nl": "Greater-than or equal comparison"}
{"code": "public static UnionCoder of(List<Coder<?>> elementCoders){\n  return new UnionCoder(elementCoders);\n}\n", "nl": "Bui
{"code": "public void testFileDeletion() throws Exception {\n  File testDir=createTestDir(\"testFileDeletion\");\n  String pref
{"code": "public boolean isOnClasspath(String classpath){\n  return this.classpath.equals(classpath);\n}\n", "nl": "Evaluates i
{"code": "protected void source(String ceylon){\n  String providerPreSrc=\"provider/\" + ceylon + \"_pre.ceylon\";\n  String pr
{"code": "public PerformanceMonitor(){\n  initComponents();\n  if (Display.getInstance().getCurrent() != null) {\n    refreshFr
{"code": "@DSGenerator(tool_name=\"Doppelganger\",tool_version=\"2.0\",generated_on=\"2014-09-03 15:01:15.190 -0400\",hash_oric
{"code": "@DSComment(\"From safe class list\") @DSSafe(DSCat.SAFE_LIST) @DSGenerator(tool_name=\"Doppelganger\",tool_version=\"
```

Figure 3.2: A Sample from the used data

3.4.1 Preprocessing

From these Java projects, we first extract the Java methods and their related Javadoc. Because they frequently describe the functionality of Java methods in accordance with Javadoc recommendations, the initial phrases of the Javadoc are used as the target comments. This refers to the person who also utilized the first sentence of a method's Javadoc as a concise synopsis. The methods without Javadoc have been removed. Second, we remove some of the Method and Comment pairs gathered in the preceding stage. In this work, pairs with one-word Javadoc descriptions are filtered out since they cannot adequately describe the Java method functionality.



```python
import os
import csv
import json
import re
import pickle

# Raw Data - Folders & Files
raw_data_files  = ["train", "test", "valid"]
raw_data_folder = "data/"

# Processed java file locations
base_folder = "java_code_"
sub_folder  = "data"

# Get AST - Only First 100 code - boolean
get_ast_full_file = False

def save_code_in_javafile(to_write_path, code):
    f = open(to_write_path + "/Input.java", "w")
    f.write(code)
    f.close()

def save_comment_in_txtfile(to_write_path, comment):
    f = open(to_write_path + "/comment.txt", "w")
    f.write(comment)
    f.close()

def extract_replacements(to_write_path, code, comment):
    varEncDict = {}
    varDecDict = {}
    codecopy = re.sub(r"\([A-z]+<.*>", "(type ", code)
    codecopy = re.sub(r"\,[A-z]+<.*>", ",type ", codecopy)
    codecopy = re.sub(r"@+.*(public|private|static)\s", "declaration", codecopy)
    decl = codecopy.split("\n")[0]
    varDecl = re.findall("\((.*?)\)", decl)[0]
    varList = varDecl.split(",")
    if (varList[0] == "" and len(varList) == 1):
        save_comment_in_txtfile(to_write_path, comment)
        save_code_in_javafile(to_write_path, code)
        return
    else:
        i = 0
        print(varList)
        for v in varList:
            name = v.split(" ")[-1]
            varEncDict[name] = "VAR" + str(i)
```

Figure 3.3: Preprocessing function

Figure3.4 shows the distribution of just the comments that are less than 40 words in length, which will help you better understand how the data is organized.

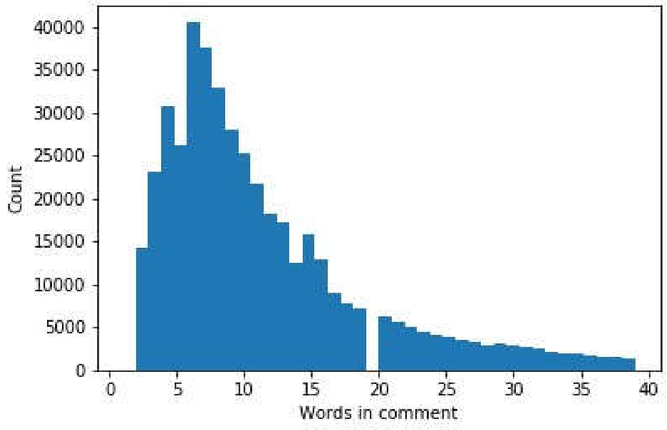
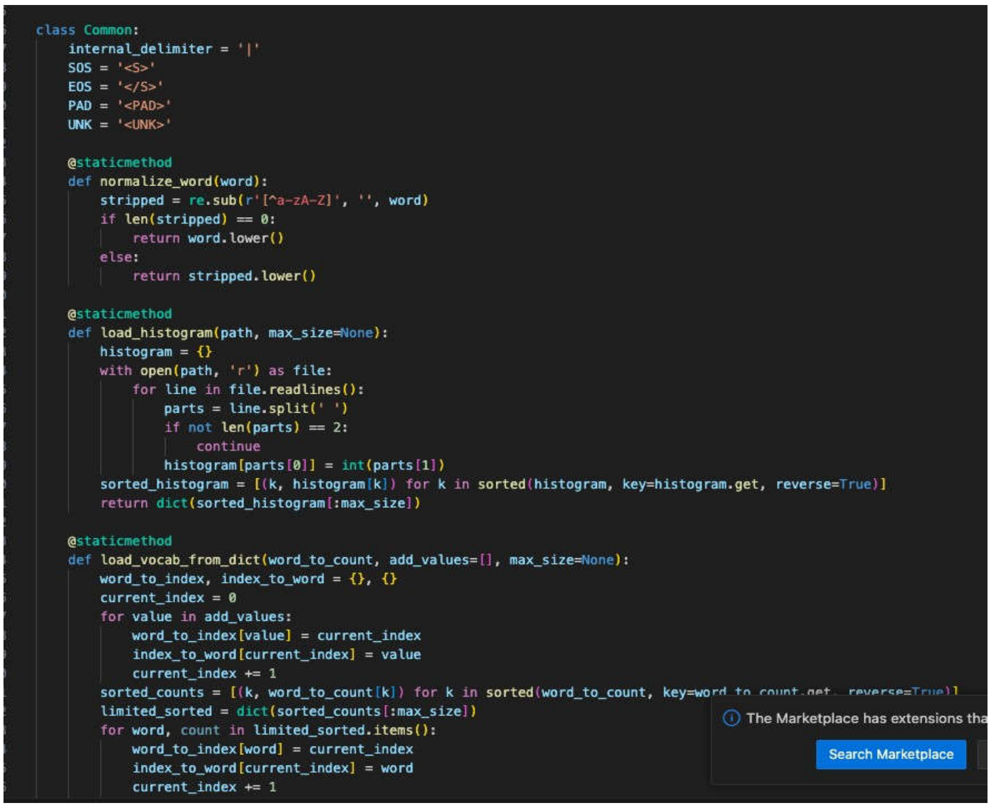


Figure 3.4: Distribution of comment lengths below 40 words in length

3.4.2 Building Vocabularies

Java lang and NLTK each parse the source code and comments into tokens. After that, all tokens are made lowercase. The generic tokens NUM and STR are used in place of the numerals and strings during the training. Source code and AST sequences are limited to 200 and 500 characters, respectively, in length. The shorter sequences are padded with a special symbol called PAD, and the longer sequences are divided into 500-token sequences. During training, we add the unique tokens START and EOS to the decoder sequences. START initiates the decoding process, and The EOS signifies its conclusion. The cap on comments is 30 characters. A unique token called UNK is used in place of the out-of- vocabulary tokens.



```python
class Common:
    internal_delimiter = '|'
    SOS = '<S>'
    EOS = '</S>'
    PAD = '<PAD>'
    UNK = '<UNK>'

    @staticmethod
    def normalize_word(word):
        stripped = re.sub(r'[^a-zA-Z]', '', word)
        if len(stripped) == 0:
            return word.lower()
        else:
            return stripped.lower()

    @staticmethod
    def load_histogram(path, max_size=None):
        histogram = {}
        with open(path, 'r') as file:
            for line in file.readlines():
                parts = line.split(' ')
                if not len(parts) == 2:
                    continue
                histogram[parts[0]] = int(parts[1])
        sorted_histogram = [(k, histogram[k]) for k in sorted(histogram, key=histogram.get, reverse=True)]
        return dict(sorted_histogram[:max_size])

    @staticmethod
    def load_vocab_from_dict(word_to_count, add_values=[], max_size=None):
        word_to_index, index_to_word = {}, {}
        current_index = 0
        for value in add_values:
            word_to_index[value] = current_index
            index_to_word[current_index] = value
            current_index += 1
        sorted_counts = [(k, word_to_count[k]) for k in sorted(word_to_count, key=word_to_count.get, reverse=True)]
        limited_sorted = dict(sorted_counts[:max_size])
        for word, count in limited_sorted.items():
            word_to_index[word] = current_index
            index_to_word[current_index] = word
            current_index += 1
```

Figure 3.5: Build vocabulairae function

We then address the Out of Vocabulary problem (OOV) after analyzing the distribution. The OOV problem, as previously established, arises from an LSTM's inability to forecast words it has never encountered before. This is due to the fact that variable names in code are made up and therefore it is irrelevant whether various words are concatenated

3.5 Abstract Syntax Tree

Luckily, working with code does have some advantages. Compared to more traditional natural language tasks, code contains a clear semantic relationship between the different lines of the code. This relationship can be captured using Abstract Syntax Trees (AST), which is a tree based representation of the code, where each node is a statement and the connections are the relations between nodes. Figure 3.7 shows an example of an AST for a given function 3.5.1

3.5.1 SBT traversal

After the AST generation, we propose a new approach to parse it using SBT. The goal of using SBT is to generate sequence of keywords, identifiers and modifiers. SBT uses brackets to present a subtree given a node. The procedure of SBT applied on the function 3.5.1 is listed as

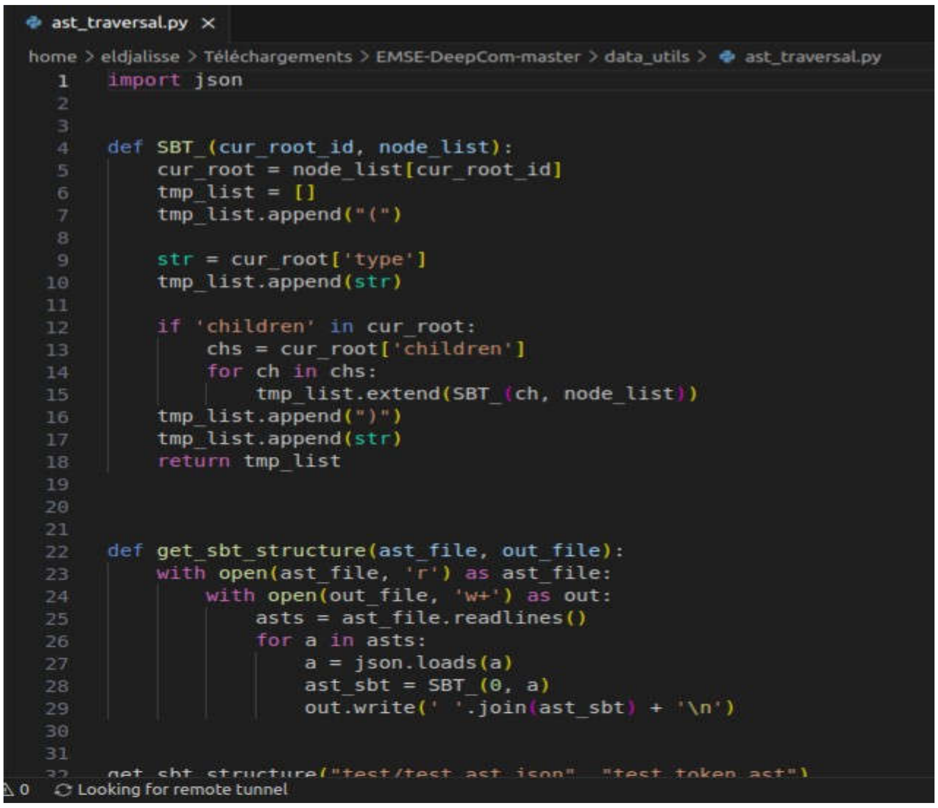
```
 /**
* Extracts request method name bound to request identifier
*/
public String extractFor(Integer id){
    LOG.debug("Extracting method with ID:{}", id); return requests.remove(id);
}
```

follows :

Listing 3.1: Java example

- The tree structure is first represented from the root node using a pair of brackets, with the root node itself placed behind the right bracket.
- After that, scan the root node's subtrees and place all of its root nodes between brackets.
- Recursively traverse each subtree until all nodes are traversed and get the final sequence

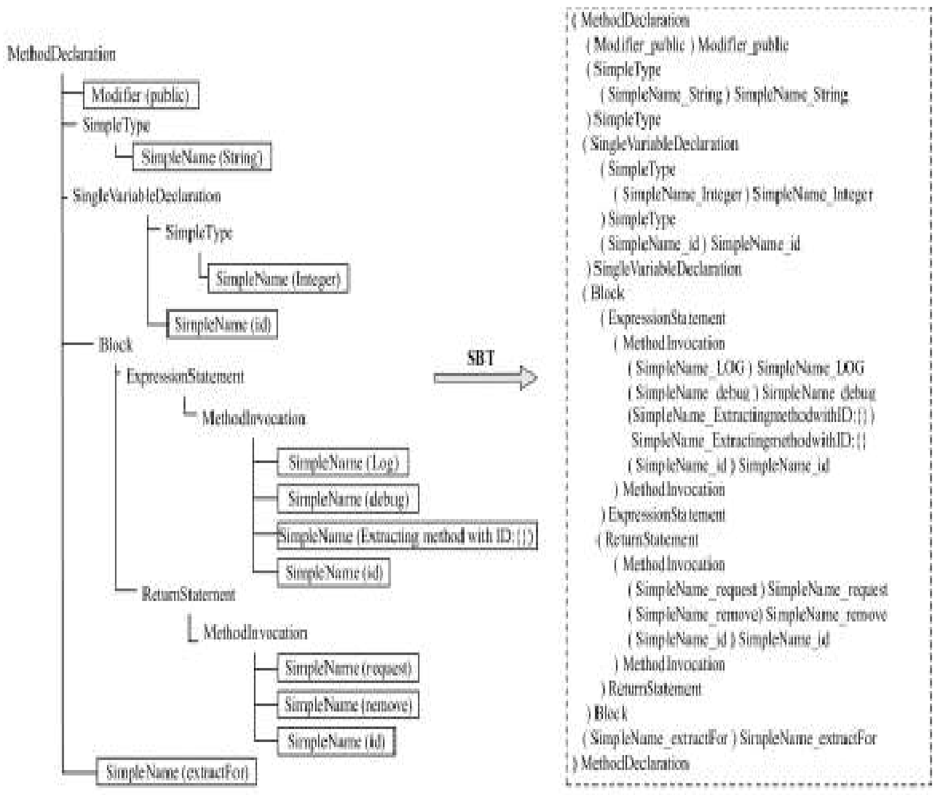
Our Model processes each AST into a sequence following the steps above. For example, the AST sequence of the following Java method Listing 3.1 extracted from Github is shown in Figure 3.7.


ast_traversal.py
home > eldjalisse > Téléchargements > EMSE-DeepCom-master > data_utils > ast_traversal.py
0 Looking for remote tunnel

```python
import json


def SBT_(cur_root_id, node_list):
    cur_root = node_list[cur_root_id]
    tmp_list = []
    tmp_list.append("(")

    str = cur_root['type']
    tmp_list.append(str)

    if 'children' in cur_root:
        chs = cur_root['children']
        for ch in chs:
            tmp_list.extend(SBT_(ch, node_list))
    tmp_list.append(")")
    tmp_list.append(str)
    return tmp_list



def get_sbt_structure(ast_file, out_file):
    with open(ast_file, 'r') as ast_file:
        with open(out_file, 'w+') as out:
            asts = ast_file.readlines()
            for a in asts:
                a = json.loads(a)
                ast_sbt = SBT_(0, a)
                out.write(' '.join(ast_sbt) + '\n')


get_sbt_structure("test/test_ast.json", "test_token_ast")
```

Figure 3.6: Function of the SBT

The nodes outside of boxes are non-terminal nodes, and nodes in boxes signify terminal nodes. The source code's structure information is specified via non-terminal nodes. Non-terminals can be of several types, including ExpressionStatement and ReturnStatement. The program text is encoded by the terminal nodes that the leaf nodes correspond to. A terminal node has a type as well as a value that can be strings, operators, variable names, etc. The terms T_non and T_termV_term are used to denote non-terminal and terminal nodes, respectively. Types of non-terminal and terminal nodes are indicated, respectively, by T_non and T_term. T_termV_term stands for type-value pairs of terminal nodes that occur frequently . If T_termV_term is out-of-vocabulary, we use its type $T_term$to represent it. For example, if SimpleName-extractFor is out-of-vocabulary, the token will be replaced by SimpleName.



```
( MethodDeclaration
  ( Modifier_public ) Modifier_public
  ( SimpleType
    ( SimpleName_String ) SimpleName_String
  ) SimpleType
  ( SingleVariableDeclaration
    ( SimpleType
      ( SimpleName_Integer ) SimpleName_Integer
    ) SimpleType
    ( SimpleName_id ) SimpleName_id
  ) SingleVariableDeclaration
  ( Block
    ( ExpressionStatement
      ( MethodInvocation
        ( SimpleName_LOG ) SimpleName_LOG
        ( SimpleName_debug ) SimpleName_debug
        (SimpleName_ExtractingmethodwithID:{} )
         SimpleName_ExtractingmethodwithID:{}
        ( SimpleName_id ) SimpleName_id
      ) MethodInvocation
    ) ExpressionStatement
    ( ReturnStatement
      ( MethodInvocation
        ( SimpleName_request ) SimpleName_request
        ( SimpleName_remove) SimpleName_remove
        ( SimpleName_id ) SimpleName_id
      ) MethodInvocation
    ) ReturnStatement
  ) Block
  ( SimpleName_extractFor ) SimpleName_extractFor
) MethodDeclaration
```

Figure 3.7: an example of the Java method

3.6 Code2Seq architecture

Our model adopts Code2Seq translating source code to natural language. The Seq2Seq, as shown in Figure 3.8, consists a two-layered Long ShortTerm Memory (LSTM) to encode AST sequences, and another deep LSTM to decode the target natural language. As illustrated in the figure, the encoder receives sequences generated by traversing the AST of Java methods using the SBT during the training phase. Subsequently, the decoder is tasked with learning the string sequences that correspond to the comments associated with the methods. The encoding and decoding process are explained in the following subsections.

3.6.1 Encoding

We're attempting to solve a sequence-to-sequence problem with our model. An encoder-decoder structure is an architecture that is employed. A sequence must first be encoded before it can be decoded.

The encoder is responsible for encoding every AST sequence of the Java method into a fixed-size vector. At each time stamp t, it reads one token of the AST sequence, then updates and records the current hidden state. $h_t = f(h_t - 1, x_t)$ and $c = q(h_1, ..., h_m)$

where c is a vector created from the sequence of the hidden states and h_t is a hidden state at timestamp t. Our study uses LSTM as f and other nonlinear functions like f and q. The rule is that $q(h_1, ..., h_m) = h_m$. Attention mechanism defines individual c_i for each target word y_i as a weighted sum of all hidden states $h1, ..., h_m$ rather than producing target words using the same context vector c $(c = q(h_1, ..., h_m) = h_m)$.A graphical representation of this can be seen in ??.

3.6.2 Decoding

The decoder aims to generate the target sequence y by sequentially predicting a word y_i conditioned on the context vector c and the previously generated words $y_1, ..., y_i 1$ where $y = y_1, ..., y_n$ and for each conditional probability is modeled as

$$p(y_i | y_1, ..., y_i 1, x) = g(y_i 1, s_i, c_i)$$

where g is a LSTM that outputs the probability of y_i, and s_i is the hidden state of the decoder LSTM for time stamp i. The probability is conditioned on a distinct context vector c_i for each target word y_i. And s_i is computed by

$$s_i = f(s_i - 1, y_i 1, c_i)$$

and The context vector c_i is computed as a weighted sum of hidden state hi in encoder.

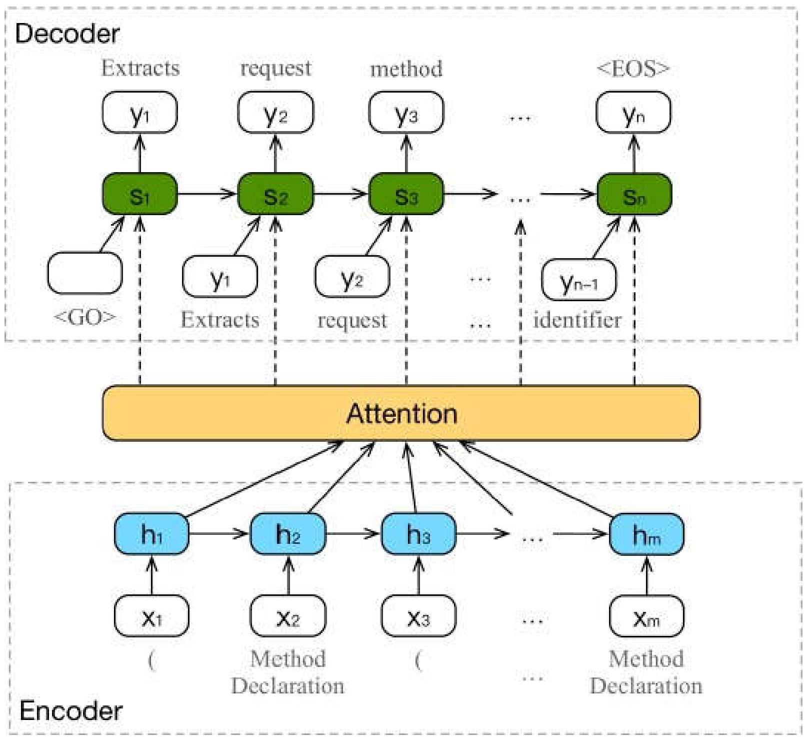


Figure 3.8: Graphic representation of Encoder and Decoder.

An LSTM is a network that uses a time-dependent hidden state (c_t) that keeps track of what the network has seen in the past. The network can choose to add knowledge into the hidden state and choose to forget knowledge. Below is shown that h_{t-1} is an input. This assures that the future sequence is influenced by previous important features. Figure 3.9 gives a graphical representation of the described LSTM, where small circles denote element-wise operations and big circles are layers [19].

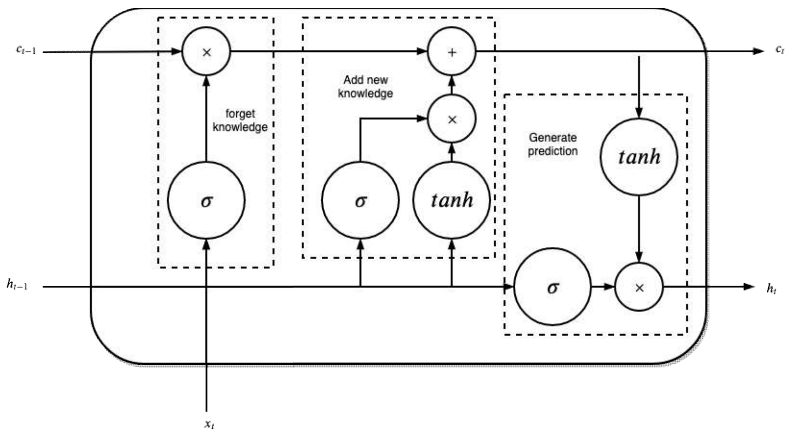
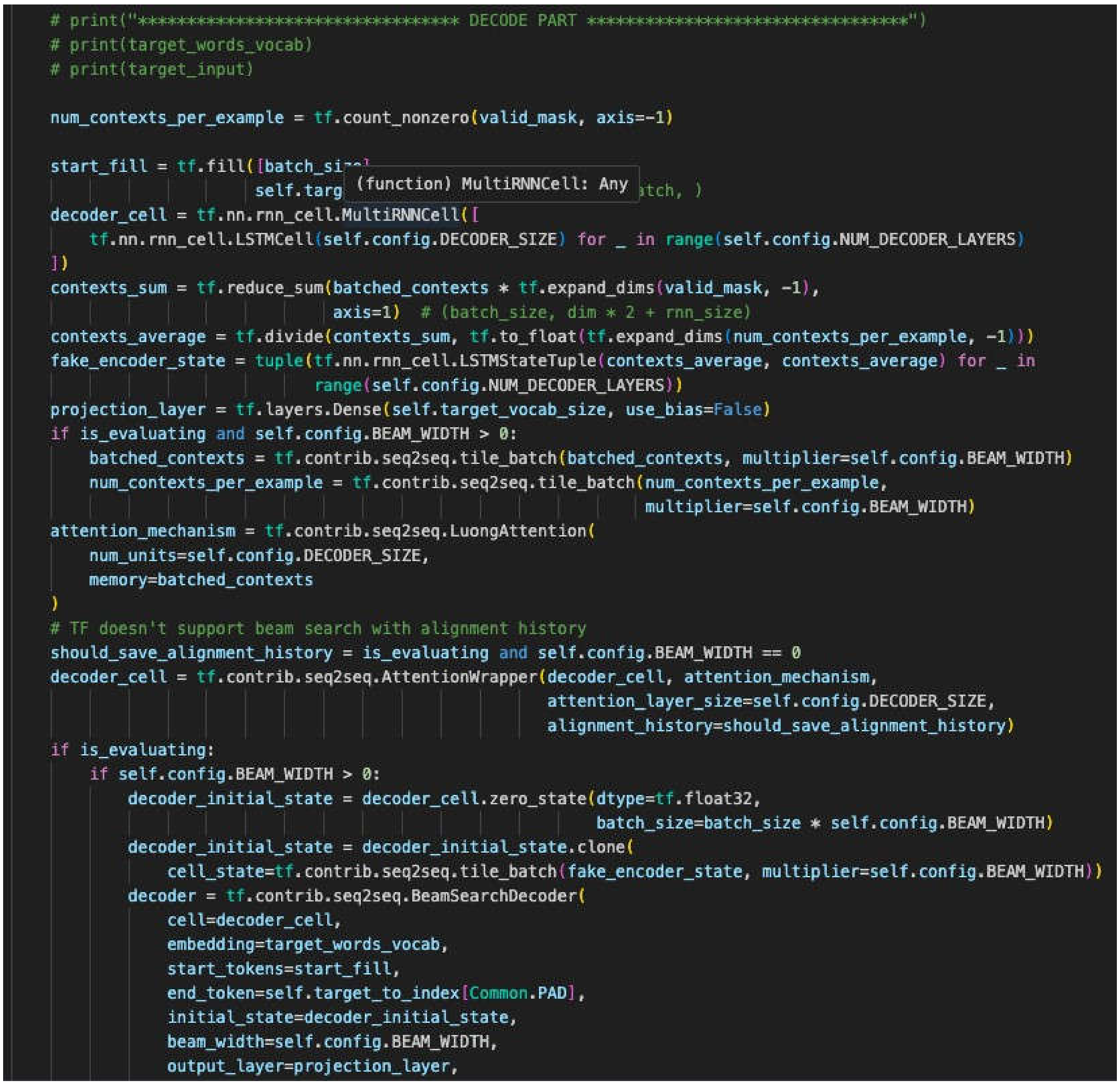


Figure 3.9: Example LSTM, operation groups are overlaid for clarity.

The value returned at time step t as the prediction is h_t. In order to overcome problems when a sequence starts and stops, special characters are used. These are often denoted as $< sos >$and $< eos >$for start and end of sequence respectively.



```
# print("********************************** DECODE PART **********************************")
# print(target_words_vocab)
# print(target_input)

num_contexts_per_example = tf.count_nonzero(valid_mask, axis=-1)

start_fill = tf.fill([batch_si
                      self.targ (function) MultiRNNCell: Any atch, )
decoder_cell = tf.nn.rnn_cell.MultiRNNCell([
    tf.nn.rnn_cell.LSTMCell(self.config.DECODER_SIZE) for _ in range(self.config.NUM_DECODER_LAYERS)
])
contexts_sum = tf.reduce_sum(batched_contexts * tf.expand_dims(valid_mask, -1),
                             axis=1)  # (batch_size, dim * 2 + rnn_size)
contexts_average = tf.divide(contexts_sum, tf.to_float(tf.expand_dims(num_contexts_per_example, -1)))
fake_encoder_state = tuple(tf.nn.rnn_cell.LSTMStateTuple(contexts_average, contexts_average) for _ in
                           range(self.config.NUM_DECODER_LAYERS))
projection_layer = tf.layers.Dense(self.target_vocab_size, use_bias=False)
if is_evaluating and self.config.BEAM_WIDTH > 0:
    batched_contexts = tf.contrib.seq2seq.tile_batch(batched_contexts, multiplier=self.config.BEAM_WIDTH)
    num_contexts_per_example = tf.contrib.seq2seq.tile_batch(num_contexts_per_example,
                                                             multiplier=self.config.BEAM_WIDTH)
attention_mechanism = tf.contrib.seq2seq.LuongAttention(
    num_units=self.config.DECODER_SIZE,
    memory=batched_contexts
)
# TF doesn't support beam search with alignment history
should_save_alignment_history = is_evaluating and self.config.BEAM_WIDTH == 0
decoder_cell = tf.contrib.seq2seq.AttentionWrapper(decoder_cell, attention_mechanism,
                                                   attention_layer_size=self.config.DECODER_SIZE,
                                                   alignment_history=should_save_alignment_history)

if is_evaluating:
    if self.config.BEAM_WIDTH > 0:
        decoder_initial_state = decoder_cell.zero_state(dtype=tf.float32,
                                                        batch_size=batch_size * self.config.BEAM_WIDTH)
        decoder_initial_state = decoder_initial_state.clone(
            cell_state=tf.contrib.seq2seq.tile_batch(fake_encoder_state, multiplier=self.config.BEAM_WIDTH))
        decoder = tf.contrib.seq2seq.BeamSearchDecoder(
            cell=decoder_cell,
            embedding=target_words_vocab,
            start_tokens=start_fill,
            end_token=self.target_to_index[Common.PAD],
            initial_state=decoder_initial_state,
            beam_width=self.config.BEAM_WIDTH,
            output_layer=projection_layer,
```

Figure 3.10: The Decoder part

3.7 Evaluation metrics

We use Information Retrieval (IR) metrics and Machine Translation (MT) metrics to evaluate the proposed method. For IR metrics, we report the precision, recall, F1 of different approaches. For machine translation metrics, we use the BLEU to evaluate our approach. The definition of these metrics are introduced as follows.

3.7.1 Precision

In statistics, precision is defined as the proportion of accurately predicted positive observations to all positively anticipated observations. In accordance with the suggested method, we calculate the precision According to (Denkowski and Lavie 2014). Both function words and words with substance are given varying weights. Typically, content words include details that help us understand the phrases. For grammar, function words are essential. In other words, content words provide us with the most crucial information, whereas function words connect those words.is computed as: $Precision = TruePositive / (TruePositive + FalsePositive)$

3.7.2 Recall

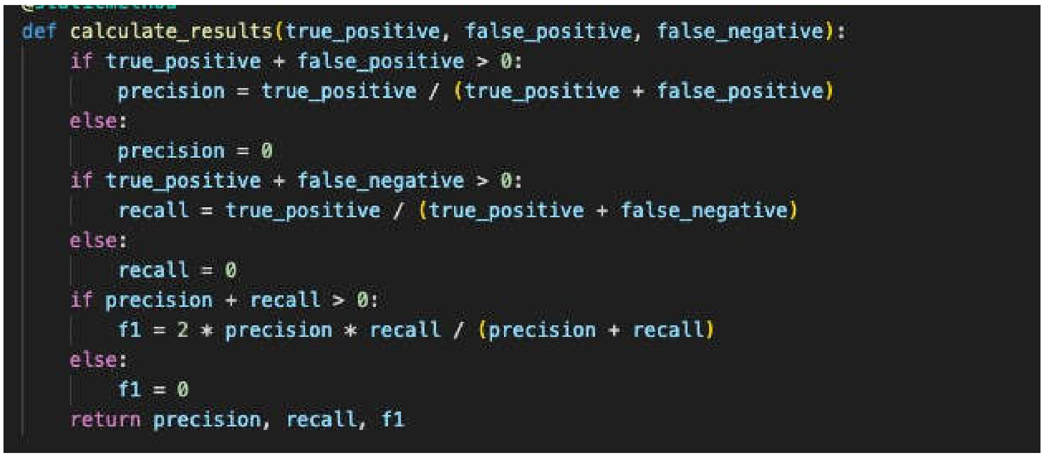
Similar to precision, we calculate the weighted recall to evaluate approaches.is calculated as :

$Recall{=}Truepositive/(Truepositive{+}Falsenegative)$

3.7.3 F1

A summary measure that combines both precision and recall; it evaluates if an increase in precision (recall) outweighs a reduction in recall (precision). In many cases, High recall indicates the sacrifice of precision, and vice versa. is computed as : $F1 = 2{*}\ Precision/(Precision + Recall)$

- True Positive (TP): Correctly predicting a label.
- True Negative (TN): Correctly predicting the other label.
- False Positive (FP): Falsely Predicting a label.
- False Negative (FN): Missing and incoming label.



```
def calculate_results(true_positive, false_positive, false_negative):
    if true_positive + false_positive > 0:
        precision = true_positive / (true_positive + false_positive)
    else:
        precision = 0
    if true_positive + false_negative > 0:
        recall = true_positive / (true_positive + false_negative)
    else:
        recall = 0
    if precision + recall > 0:
        f1 = 2 * precision * recall / (precision + recall)
    else:
        f1 = 0
    return precision, recall, f1
```

Figure 3.11: IR metrics function

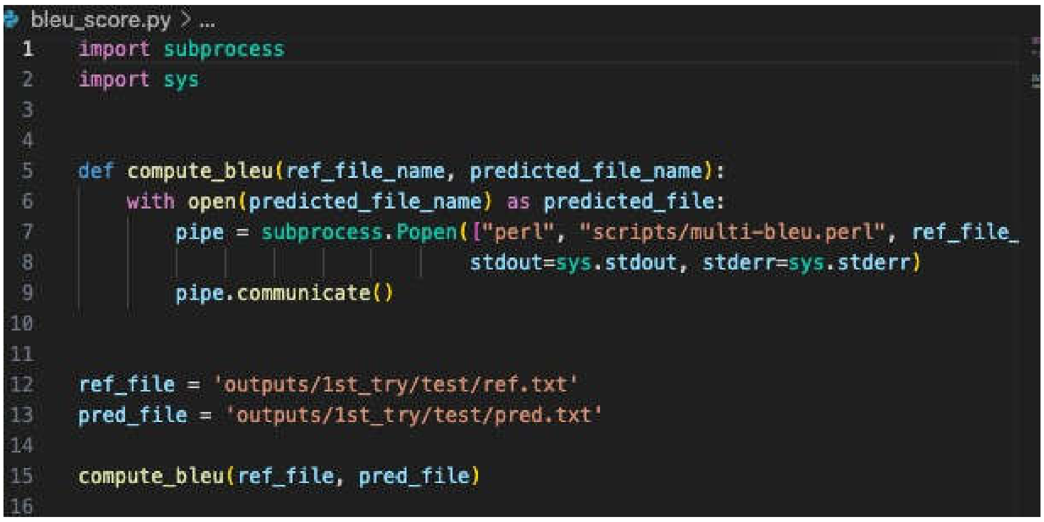
3.7.4 BLEU score

The Bilingual evaluation understudy (BLEU-4) was the primary evaluation metric used to assess the comments produced by our models [17]. Using a corpus of potential translations, BLEU evaluates the accuracy of a translation (in our instance, from source code to comments). It is frequently used to gauge how accurate neural machine translation (NMT)[20]. BLEU measures how close the output of the machine is to the one of a human expert. The closer they are, the higher the BLEU score will be [7]. The BLEU score ranges between 0 and 1, with 1 being perfect natural language. BLEU has shown a high correlation with human quality judgment [21] and is one of the most popular metrics to measure the quality of machine translated text.

Our final evaluation result is obtained as the average of the BLEU scores for each of the analyzed snippets. The n-gram similarity is calculated by BLEU and used to generate a score using the following equation:

$$BLEU = BP \cdot exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

As the reference corpus in our scenario has only one comment per code snippet, BP = 1. N is the maximum length of n-grams being utilized in the comparison[17], which in our case is N=4; w_n is a weight vector that adds to 1, p_n is the modified n-gram precision



```
bleu_score.py > ...
import subprocess
import sys


def compute_bleu(ref_file_name, predicted_file_name):
    with open(predicted_file_name) as predicted_file:
        pipe = subprocess.Popen(["perl", "scripts/multi-bleu.perl", ref_file_
                                stdout=sys.stdout, stderr=sys.stderr)
        pipe.communicate()


ref_file = 'outputs/1st_try/test/ref.txt'
pred_file = 'outputs/1st_try/test/pred.txt'

compute_bleu(ref_file, pred_file)
```

Figure 3.12: Implementation of BLEU-4 calculation

We also measured precision, recall, and F1 score as used in [22] and [23] over all the sub-tokens of the target sequence, case-insensitive.

3.8 Conclusion

Automatic comment generation for Java code can be very beneficial. Both for readability of old code, but also for better comment suggestions as a programmer aid. With the DeepCom as the baseline.
In this chapter, we have presented an overview of the related work, introduced our proposition, and provided a detailed description of our approach. In the subsequent sections, we will present the results of our experiments and evaluate the effectiveness of our proposition.

Chapter 4

Implementation, Experiments and Results

4.1 Introduction

So far, we introduced the process of generating comments from Java source code using an NMT-based technique. In this chapter, we present the technical details for implementing this process by training our proposed approach(Code2Seq) and also by using Open NMT, an open-source and extensible NMT architecture, on a subset of the dataset the same which used in deepcom. First, we briefly present the tools, languages, and programming environment used in this work. Second, we describe the proposed approach and the Open NMT architecture, training and validation phases. Finally, We conclude the chapter by presenting a sample of obtained results and a discussion.

4.2 Environment of Development

4.2.1 Python

Figure 4.1: Overleaf logo.

Python is a high-level programming language created by Guido van Rossum and released in 1991[24].Python has a portable type system, dynamic, extensible, free, very simple syntax, code shorter than C or Java, multi-thread, object-oriented, scalable [25],Reading and writing Python code is simple, and it is brief without being obscure. Python is an effective expressive programming language [26]

4.2.2 Google Colaboratory

Figure 4.2: Overleaf logo.

Google Colaboratory[27], often abbreviated as "Colab" is a cloud service, offered by Google (free), based on Jupyter Notebook and intended for training and research in machine learning. This platform allows to train machine learning models directly into the cloud. So without having to install anything on our computer except a browser.This environment allows us to write and execute Python code in your browser, with no configuration required, free access to GPUs, easy sharing.

4.2.3 Visual Studio Code

Figure 4.3: Visual studio logo.

Visual Studio Code is a source code editor that can be used with a variety of programming languages, including Java, JavaScript, Go, Node.js, and C++. It is based on the Electron Framework, which is used to develop Web applications that run on the Blink presentation engine. [28]

4.3 Tools of Devlopelment

4.3.1 Tensorflow

Figure 4.4: Tensorflow logo.

TensorFlow is an open-source software library for high-performance digital computing. Its flexible architecture enables easy deployment of computing across a variety of platforms (CPU, GPU, TPU), and from desktop computers to server clusters to mobile and peripheral devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it has strong support for machine learning and deep learning, and flexible digital computation is used in many other scientific fields.[29]

4.3.2 Numpy

Figure 4.5: Tensorflow logo.

NumPy is a library for the Python programming language, adding support for large multidimensional matrices and matrices, as well as a large collection of high-level mathematical functions to run on these matrices.NumPy’s predecessor, Numeric, was originally created by Jim Hugunin with contributions from several other developers.
In 2005, Travis Oliphant created NumPs by incorporating Numarray’s capabilities into Numerics, with numerous modifications.[24]

4.4 Experimental methodology

This section demonstrates how the evaluation of the proposed method is conducted. We provide two implementations of this method; The first using Code2Seq architecture and the second using Open NMT framework. The two experiments use the deepcom dataset that we described in the previous chapter. In the first experiment, (*Code2Seq based model*) we used Code2Seq code to train a new model with the aforementioned dataset with the inclusion of the ASTs based sequences. This task is more challenging when generating comments than when predicting function names. This difficulty arises from the fact that comments are usually longer than function names. Also, we are generating natural language instead of a sequence of descriptive words. In the second experiment, we use a research-friendly framework called Open NMT. This framework enables us to test novel approaches effortlessly

The results of both methods are then compared with the results of the DeepCom[11] as the baseline.

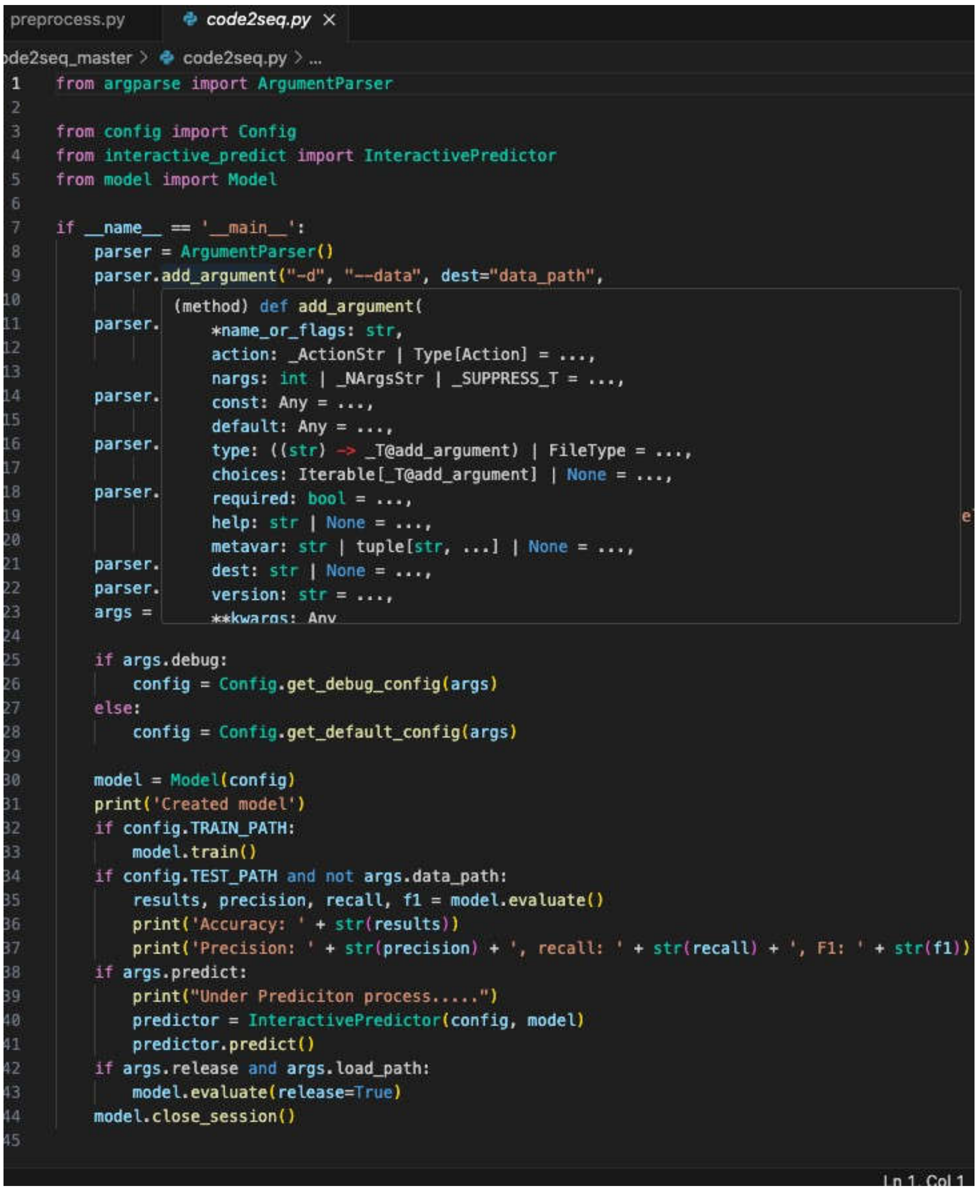
(*Method1 OpenNMT*), it is created to be research-friendly so that researchers may test out novel approaches to tasks like summarization, language modeling, and translation. While for the second one For each of these 2 experiments, we calculated the BLEU score as a measure of the quality of the obtained results.

4.5 Experiments

our experiment (*Method2*), we traversed the Java AST extractor used in Code2Seq. Instead of using the name of the examined function of the Java snippet. Instead of using each variable with a generic name and its type, we also added the actual name of the variable [30]. The reason behind this choice was that the name of each variable may play an important role in the functionality of the examined snippet. Thus, by knowing its name we will be able to generate more accurate comments.

4.6 First Experiment: Code2Seq based-model experimentation

The adapted Code2Seq model is illustrated in Figure 3.1. We address the code generator task as a machine translation problem that translates program code to a natural language sequences. To embed the structural information, Code2Seq uses AST sequences as its input. The AST sequences are generated by applying SBT on the AST which can not only express the tree structure but also keep no ambiguity. The proposed Code2Seq model mainly contains three components: An AST builder with the SBT traversal component, an attention-based Seq2Seq model builder, and a trained model (generated by the two previous components) to generate comment given a code snippets.

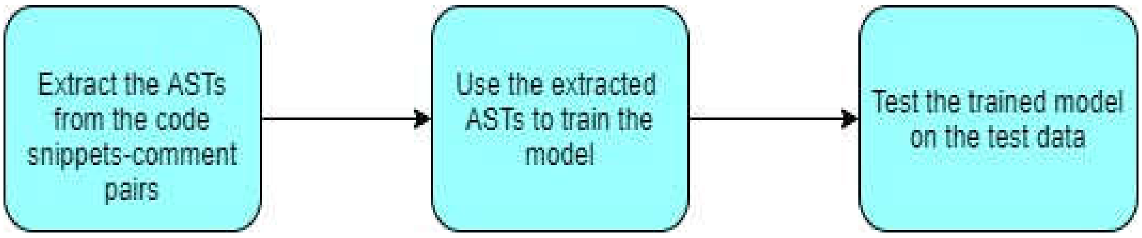


```
preprocess.py    code2seq.py ×
de2seq_master > code2seq.py > ...
from argparse import ArgumentParser

from config import Config
from interactive_predict import InteractivePredictor
from model import Model

if __name__ == '__main__':
    parser = ArgumentParser()
    parser.add_argument("-d", "--data", dest="data_path",
                  (method) def add_argument(
    parser.           *name_or_flags: str,
                      action: _ActionStr | Type[Action] = ...,
                      nargs: int | _NArgsStr | _SUPPRESS_T = ...,
    parser.           const: Any = ...,
                      default: Any = ...,
    parser.           type: ((str) -> _T@add_argument) | FileType = ...,
                      choices: Iterable[_T@add_argument] | None = ...,
    parser.           required: bool = ...,
                      help: str | None = ...,                          e
                      metavar: str | tuple[str, ...] | None = ...,
    parser.           dest: str | None = ...,
    parser.           version: str = ...,
    args =            **kwargs: Any

    if args.debug:
        config = Config.get_debug_config(args)
    else:
        config = Config.get_default_config(args)

    model = Model(config)
    print('Created model')
    if config.TRAIN_PATH:
        model.train()
    if config.TEST_PATH and not args.data_path:
        results, precision, recall, f1 = model.evaluate()
        print('Accuracy: ' + str(results))
        print('Precision: ' + str(precision) + ', recall: ' + str(recall) + ', F1: ' + str(f1))
    if args.predict:
        print("Under Prediciton process.....")
        predictor = InteractivePredictor(config, model)
        predictor.predict()
    if args.release and args.load_path:
        model.evaluate(release=True)
    model.close_session()

Ln 1, Col 1
```

Figure 4.6: The Code2seq function from the proposed model

4.6.1 Description

Our model translates the source code to a high-level description at the method level. our method combines the source code and the AST sequences to generate the code comments. It learns both the lexical and syntactical information to generate the comments. Our framework consists of three stages: data processing, model training, and testing.



Extract the ASTs from the code snippets-comment pairs
Use the extracted ASTs to train the model
Test the trained model on the test data

Figure 4.7:

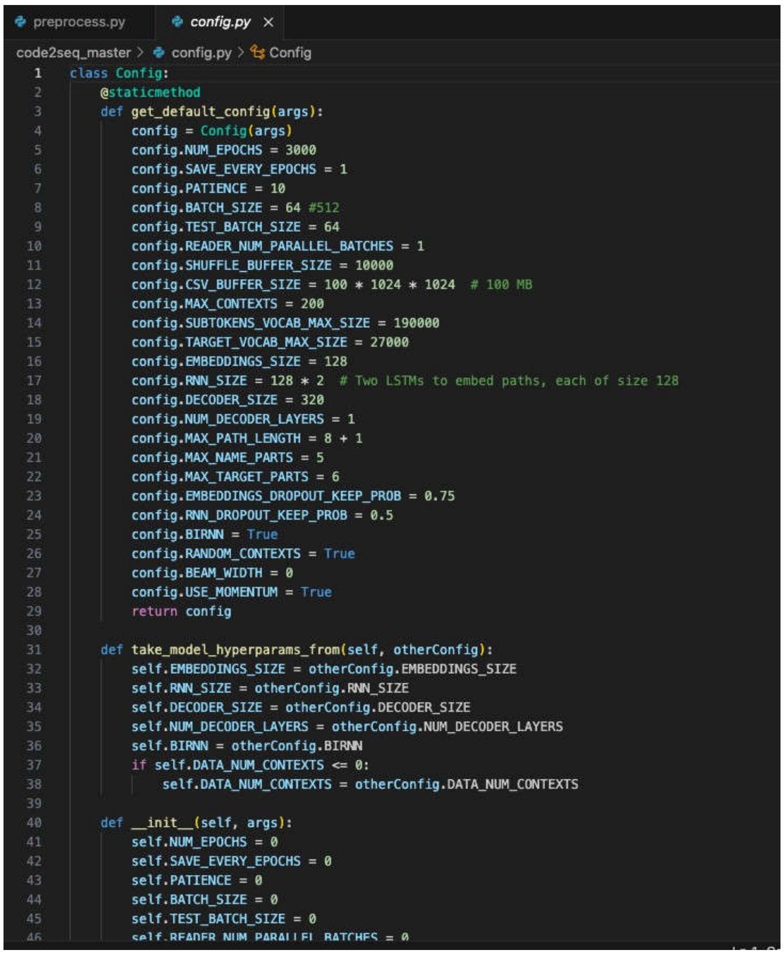
The source code we obtained from GitHub is parsed and preprocessed into a parallel corpus of Java methods and their corresponding comments. In order to learn the structural information, the Java methods are converted into AST sequences by a special traversal approach before input into the model. With the parallel corpus of the source code, traversed AST sequences, and comments, we build and train generative neural models based on the idea of NMT. There are three challenges during training process:

- How to represent ASTs to preserve the structural information and keep the representation unambiguous while traversing the ASTs?
- How to learn both the lexical information and the syntactic information?
- How to reduce the out-of-vocabulary tokens in the source code?

4.6.2 Training

Code2Seq is trained on the Tensorflow framework, During training, the model is validated every 2,000 minibatches on the validation set by BLEU score which is a commonly used evaluation metric for NMT. The maximum number of epochs is 50. The validation set is used to estimate how well the model has been trained and to estimate model properties. Finally, the accuracy of the model on test set gives a realistic estimate of the performance of the model on unseen data. However, the validation process takes extra time. Taking time and model capacity into consideration, we validate every 2,000 minibatches and select the model which performs best on the validation dataset as the best model. The best model is then evaluated on the test set by computing average BLEU scores. For implementation, we build our model using Tensorflow which is an open-source deep learning framework. The time of training lasts about 80 hours for each model. The hyperparameters are shown as follows figure 4.8:

- We use two-layered LSTMs with 256 dimensions of the hidden states and 128-dimensional word embeddings.
- The learning rate was 0.01 and decayed by 0.05 every epoch.
- We applied dropout of 0.3 and a recurrent dropout of 0.5 on the LSTM encoding the AST


preprocess.py config.py
code2seq_master > config.py > Config

```python
class Config:
    @staticmethod
    def get_default_config(args):
        config = Config(args)
        config.NUM_EPOCHS = 3000
        config.SAVE_EVERY_EPOCHS = 1
        config.PATIENCE = 10
        config.BATCH_SIZE = 64 #512
        config.TEST_BATCH_SIZE = 64
        config.READER_NUM_PARALLEL_BATCHES = 1
        config.SHUFFLE_BUFFER_SIZE = 10000
        config.CSV_BUFFER_SIZE = 100 * 1024 * 1024  # 100 MB
        config.MAX_CONTEXTS = 200
        config.SUBTOKENS_VOCAB_MAX_SIZE = 190000
        config.TARGET_VOCAB_MAX_SIZE = 27000
        config.EMBEDDINGS_SIZE = 128
        config.RNN_SIZE = 128 * 2  # Two LSTMs to embed paths, each of size 128
        config.DECODER_SIZE = 320
        config.NUM_DECODER_LAYERS = 1
        config.MAX_PATH_LENGTH = 8 + 1
        config.MAX_NAME_PARTS = 5
        config.MAX_TARGET_PARTS = 6
        config.EMBEDDINGS_DROPOUT_KEEP_PROB = 0.75
        config.RNN_DROPOUT_KEEP_PROB = 0.5
        config.BIRNN = True
        config.RANDOM_CONTEXTS = True
        config.BEAM_WIDTH = 0
        config.USE_MOMENTUM = True
        return config

    def take_model_hyperparams_from(self, otherConfig):
        self.EMBEDDINGS_SIZE = otherConfig.EMBEDDINGS_SIZE
        self.RNN_SIZE = otherConfig.RNN_SIZE
        self.DECODER_SIZE = otherConfig.DECODER_SIZE
        self.NUM_DECODER_LAYERS = otherConfig.NUM_DECODER_LAYERS
        self.BIRNN = otherConfig.BIRNN
        if self.DATA_NUM_CONTEXTS <= 0:
            self.DATA_NUM_CONTEXTS = otherConfig.DATA_NUM_CONTEXTS

    def __init__(self, args):
        self.NUM_EPOCHS = 0
        self.SAVE_EVERY_EPOCHS = 0
        self.PATIENCE = 0
        self.BATCH_SIZE = 0
        self.TEST_BATCH_SIZE = 0
        self.READER_NUM_PARALLEL_BATCHES = 0
```

Figure 4.8: The Configuration file

4.6.3 Validation and results

We evaluate the difference between comments generated automatically and those posted by humans. The average BLEU-4 scores of various techniques for producing comments for Java methods are shown in Table 4.1.

Approaches	BLEU-4 score
DeepCom	38.17
Our Method	38.72

Table 4.1: Evaluation results on Java Methods

Approaches	Precision	Recall	F1
our Method	46.94	27.44	34.63

Table 4.2: Performance on Java code

As the structural information is included, the BLEU-4 score increases even further. The SBT-based model is much better and is able to understand the semantic and syntactic information contained inside Java methods Our Method confirms the effectiveness of solving the problem of Out of Vocabulary in the ASTs. The results are depicted in Table 4.4. The results show that the proposed method is capable of understanding the syntactic and semantic meaning of java source codes and hence generates meaningful comments automatically.

left=2cm,right=0.25cm,bottom=0.25cm,top=0.25cm

4.6.4 Discussion

As shown in previous sections, our model can achieve promising performance in generating comments for Java methods. However, there are some other observations worth further investigation. In this section, we will report some observations, including:

1. When The proposed generates comments with high BLEU score?
2. Why there are unknown words in the generated comments?

To analyze the reason why generates comments with high BLEU scores, and unknown words, The detailed steps are as below: • We first split the generated comments into two distinct sets, namely, the high BLEU score comments(more than 70%), and comments with UNK. In search at the Comments with a High BLEU Score we get that There are many comments generated by the proposed approach with high BLEU score and we are interested in which kinds of comments with high BLEU score.For example, as Case 5 in Table 4.3 shows, the method “sort” aims to sort an array using quick sort, our proposed model captures the correct functionality and generates the correct comment. However, deepcom ignores much important information and just generates one keyword “sort”. There are many Java source code-generated comments the same as the references. Most of these Java methods have clear functionality and the code conventions are universal. the method proposed can generate exactly correct comments from the source code(Case 1 in Table 4.3), which validate the capability of our approach to encode Java methods and decode comments. The comment generated by Deepcom is related to the source code, but it losses a lot of functionalities of Java methods. Investigating the Reasons for Unknown Words in Generated Comments that ,there are unknown words in the generated comments and these words have no meaning for describing the Java methods. Similar to the evaluation of the above discussion, we also randomly select 100 generated comments with unknown words.we analyze the accurate words that are replaced by unknown token UNK shown in table
4.4.

As Case 3 in Table 4.3 shows, the proposed approach fails to predict the token “SeparaterPainter” which is the method name defined by developers. Our model is not good at learning the method or identifiers names occurred in comments cause of Developers define various names while programming and most of these tokens appearing at most once in the comments. However,

Samples (100 generated comments)	
Classname	12
Methodename	30
Variablename	10
Others	50

Table 4.4: The accurate words of the UNK words in a Sample

we can not determine tokens of comments are identifiers from source code or ordinary natural language words. It is hard for the proposed approach to learn these user-defined tokens in comments that have been replaced by the unknown token UNK during data processing.

4.7 Second Experiment: Open NMT-based model

We believe that building the comment generator for the Java language from scratch is a tedious task. Thus, we adapted the open NMT architecture, which is an open source platform for training translators between spoken languages, to build a model for generating comment from source code. The training process is conducted on a large subset of Java codes and their corresponding naturel languge comments . Full details are provided in the following subsections.

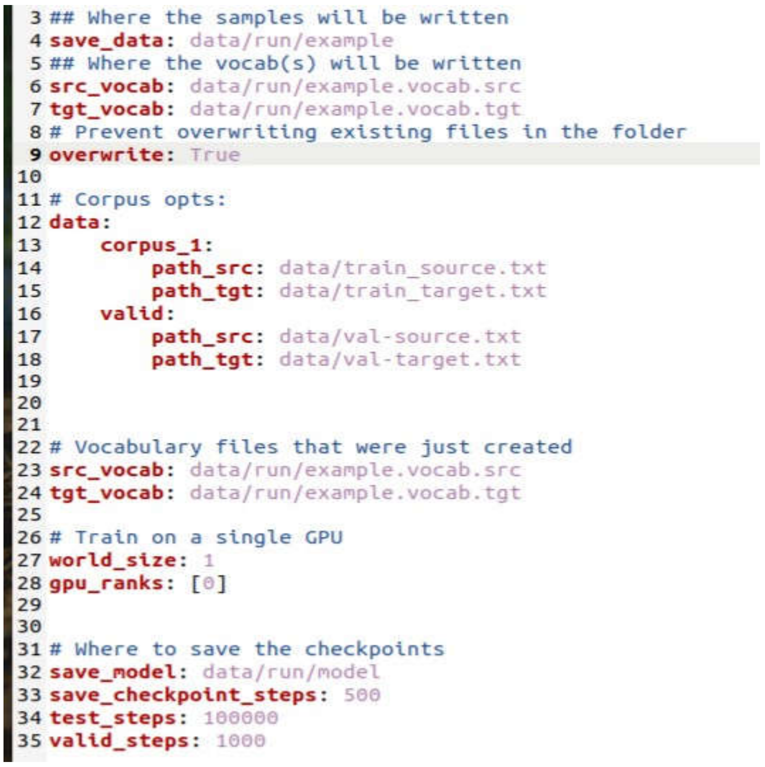
4.7.1 Open-NMT

OpenNMT [20] is a complete library for training and deploying neural machine translation models. The system is the successor to seq2seq-attn, developed at Harvard, and has been completely rewritten for ease of efficiency, readability, and generalizability. It includes vanilla NMT models along with support for attention, gating, stacking, input feeding, regularization, beam search and all other options necessary for state-of-the-art performance. The core system of OpenNMT is implemented using the Lua/Torch mathematical framework, allowing easy extension through Torch’s internal neural network components. Additionally, Adam Lerer from Facebook Research has extended the library to support the Python/PyTorch framework while maintaining the same API [20].

The pretraining of OpenNMT on Our dataset the same of deepcomm goes through several steps that starts by the configuration file after the partition and the preparation of the source and target files for both training and validation. These steps are detailed bellow.

4.7.2 Open NMT configuration

For training the model, Open NMT requires a YAML configuration file, known as config in OpenNMT. This file defines the parameters for the training step. The location of the target and source data for both the validation and training. In addition, the file holds the following parameters.



```
## Where the samples will be written
save_data: data/run/example
## Where the vocab(s) will be written
src_vocab: data/run/example.vocab.src
tgt_vocab: data/run/example.vocab.tgt
# Prevent overwriting existing files in the folder
overwrite: True

# Corpus opts:
data:
    corpus_1:
        path_src: data/train_source.txt
        path_tgt: data/train_target.txt
    valid:
        path_src: data/val-source.txt
        path_tgt: data/val-target.txt



# Vocabulary files that were just created
src_vocab: data/run/example.vocab.src
tgt_vocab: data/run/example.vocab.tgt

# Train on a single GPU
world_size: 1
gpu_ranks: [0]


# Where to save the checkpoints
save_model: data/run/model
save_checkpoint_steps: 500
test_steps: 100000
valid_steps: 1000
```

Figure 4.9: Configuration file for OpenNMT method

- overwrites : if this option is set in TRUE, you can write overwritten during the training process.
- data : in this part, we define the paths to the training and validation data files.
- save data : it means setting the base filename and location where the vocabulary files and preprocessed data will be saved.
- save model :sets the base filename and location where the trained model checkpoints will be saved.

– save checkpoint steps : save checkpoint steps: it is a saved snapshot of the model at a particular moment. For example, as shown in Figure 4.9, we set "save checkpoint steps" to 500, which means that a checkpoint will be saved for every 500 training steps.

– train steps and valid steps Indicates the total number of training and validation steps that will be performed.

4.7.3 Training data and code

In this project, we utilized the same dataset of deepcomm, which contains Java code and there corresponding natural language comments, to train a model for generating comments. To effectively use this dataset with OpenNMT, we had to split it into six separate files: "train_source.txt,""valsource.txt,""testsource.txt,""train_target.txt,""valtarget.txt," and "testtarget.txt." The src files hold the source Java code for training, validation, and testing, respectively, and the target files will hold the target test cases corresponding to the source Java code. This split is a crucial step in preparing the data for training and evaluation using OpenNMT. The table 4.7.3 represents the number of examples in the dataset for different sets: training, validation, and test. Each set contains a certain number of source code and target-code pairs.

Set	Source	Target
Training	119013	119013
Validation	20000	20000
Test	20000	20000

The data has been uploaded to the "content" directory in Google Colaboratory as per the following figure 4.10

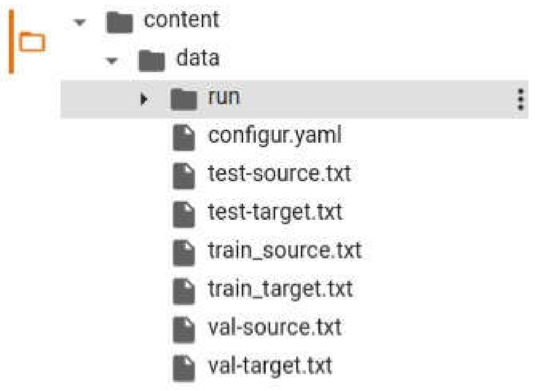
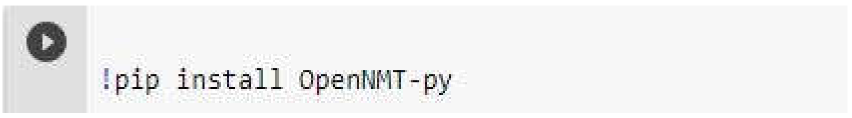

content
data
run
configur.yaml
test-source.txt
test-target.txt
train_source.txt
train_target.txt
val-source.txt
val-target.txt

Figure 4.10: Dataset path.

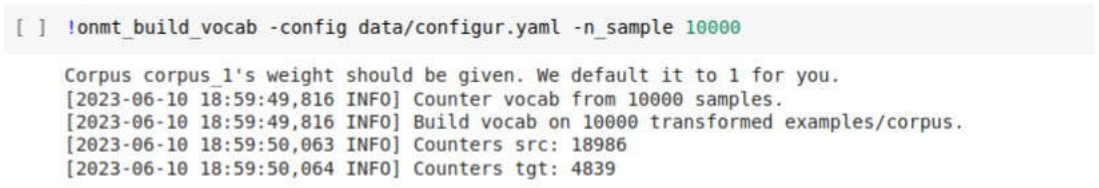
First, we need to install OpenNMT-py. In order to do that, we use the following command figure4.11 in Google Colab:



```
!pip install OpenNMT-py
```

Figure 4.11: The installation command .

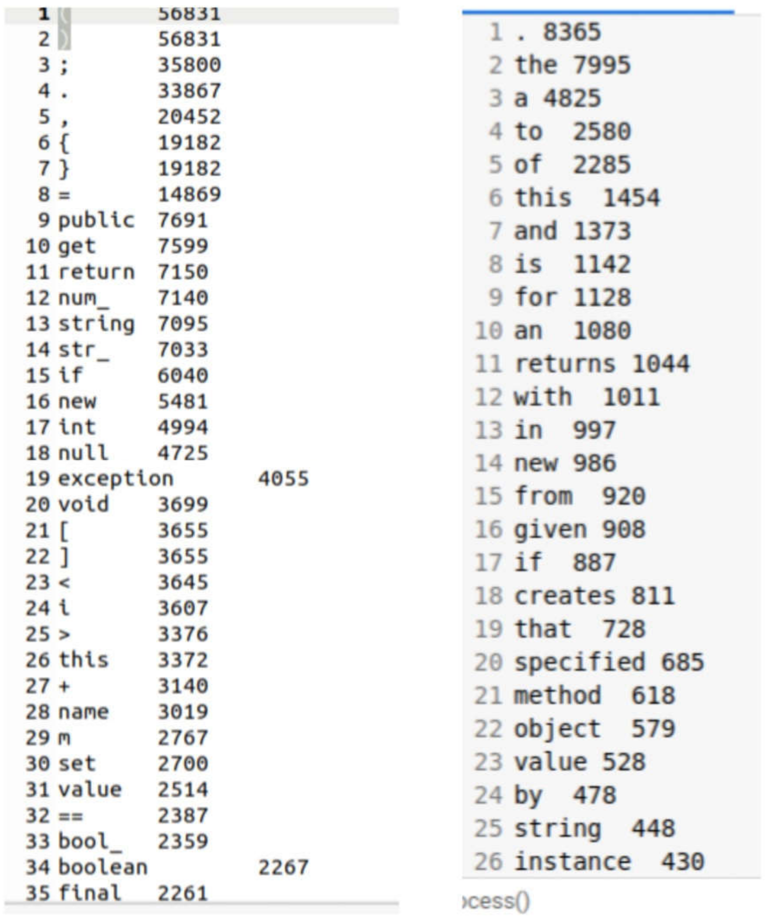
Next, we execute the command shown in Figure 4.12 to build the vocabulary using the provided configuration file.



```
[ ] !onmt_build_vocab -config data/configur.yaml -n_sample 10000

Corpus corpus_1's weight should be given. We default it to 1 for you.
[2023-06-10 18:59:49,816 INFO] Counter vocab from 10000 samples.
[2023-06-10 18:59:49,816 INFO] Build vocab on 10000 transformed examples/corpus.
[2023-06-10 18:59:50,063 INFO] Counters src: 18986
[2023-06-10 18:59:50,064 INFO] Counters tgt: 4839
```

Figure 4.12: Command for building the Vocabulary.

Samples from the vocabulary files (source files which is the vocabulary of java source code and target files which is the vocabulary of comment generation) are shown in the figure4.13 below :

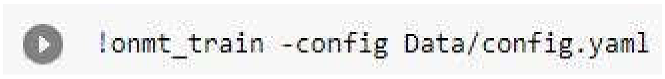


```
 1          56831
 2          56831
 3 ;        35800
 4 .        33867
 5 ,        20452
 6 {        19182
 7 }        19182
 8 =        14869
 9 public   7691
10 get      7599
11 return   7150
12 num_     7140
13 string   7095
14 str_     7033
15 if       6040
16 new      5481
17 int      4994
18 null     4725
19 exception        4055
20 void     3699
21 [        3655
22 ]        3655
23 <        3645
24 i        3607
25 >        3376
26 this     3372
27 +        3140
28 name     3019
29 m        2767
30 set      2700
31 value    2514
32 ==       2387
33 bool_    2359
34 boolean          2267
35 final    2261
```

```
 1 . 8365
 2 the 7995
 3 a 4825
 4 to  2580
 5 of  2285
 6 this  1454
 7 and 1373
 8 is  1142
 9 for 1128
10 an  1080
11 returns 1044
12 with  1011
13 in  997
14 new 986
15 from  920
16 given 908
17 if  887
18 creates 811
19 that  728
20 specified 685
21 method  618
22 object  579
23 value 528
24 by  478
25 string  448
26 instance  430
```

Figure 4.13: Example from source and target vocabulary files.

The vocabulary files contain a list of tokens with their corresponding numerical representations. Once the vocabulary file is prepared, we move to the training step by executing the following command in figure4.14.

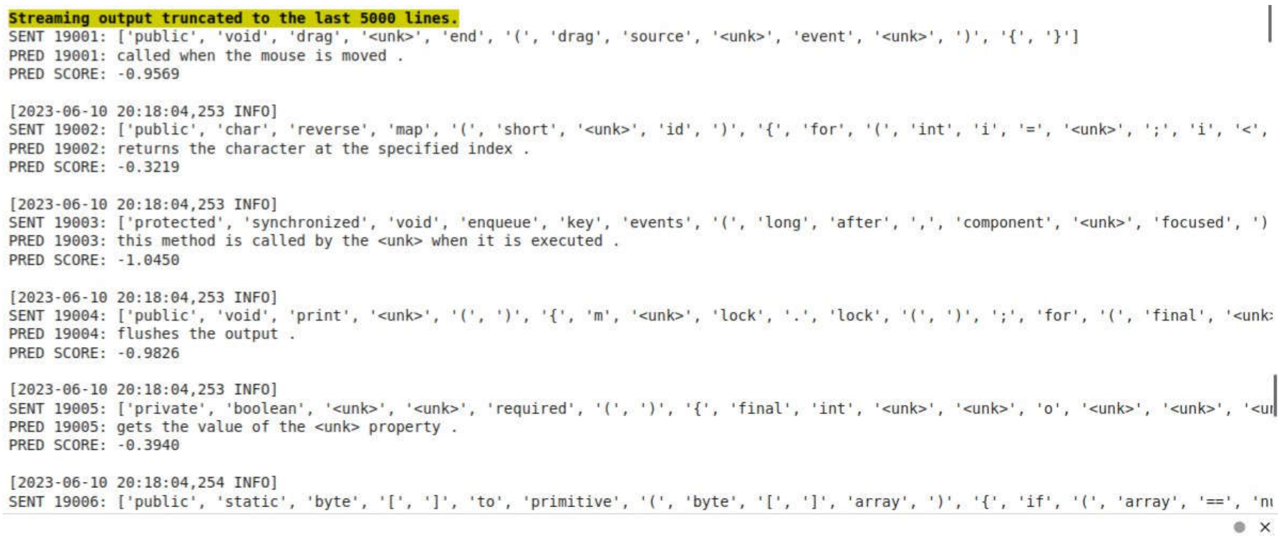


```
!onmt_train -config Data/config.yaml
```

Figure 4.14: Training command

4.7.4 Results

After completing the training step in OpenNMT-py, we can move forward to the translation phase. During this phase, we will translate the contents of the source file Data/test-source.txt using the trained model. The translated output will be saved to the file named Data/pred_1000.txt. To perform the translation, we execute the following command in figure 4.15



```
Streaming output truncated to the last 5000 lines.
SENT 19001: ['public', 'void', 'drag', '<unk>', 'end', '(', 'drag', 'source', '<unk>', 'event', '<unk>', ')', '{', '}']
PRED 19001: called when the mouse is moved .
PRED SCORE: -0.9569

[2023-06-10 20:18:04,253 INFO]
SENT 19002: ['public', 'char', 'reverse', 'map', '(', 'short', '<unk>', 'id', ')', '{', 'for', '(', 'int', 'i', '=', '<unk>', ';', 'i', '<',
PRED 19002: returns the character at the specified index .
PRED SCORE: -0.3219

[2023-06-10 20:18:04,253 INFO]
SENT 19003: ['protected', 'synchronized', 'void', 'enqueue', 'key', 'events', '(', 'long', 'after', ',', 'component', '<unk>', 'focused', ')
PRED 19003: this method is called by the <unk> when it is executed .
PRED SCORE: -1.0450

[2023-06-10 20:18:04,253 INFO]
SENT 19004: ['public', 'void', 'print', '<unk>', '(', ')', '{', 'm', '<unk>', 'lock', '.', 'lock', '(', ')', ';', 'for', '(', 'final', '<unk
PRED 19004: flushes the output .
PRED SCORE: -0.9826

[2023-06-10 20:18:04,253 INFO]
SENT 19005: ['private', 'boolean', '<unk>', '<unk>', 'required', '(', ')', '{', 'final', 'int', '<unk>', '<unk>', 'o', '<unk>', '<unk>', '<u
PRED 19005: gets the value of the <unk> property .
PRED SCORE: -0.3940

[2023-06-10 20:18:04,254 INFO]
SENT 19006: ['public', 'static', 'byte', '[', ']', 'to', 'primitive', '(', 'byte', '[', ']', 'array', ')', '{', 'if', '(', 'array', '==', 'n
```

Figure 4.15: The Obtained results

In this command, we specify the path to the trained model using the-

```
!onmt_translate -model data/run/model_step_10000.pt -src data/test-source.txt
  -output data/pred_1000.txt -gpu 0 -verbose
```

model option, the source file is provided with the -src option

1

2

Listing 4.1: Validation Command

4.8 Conclusion

Our Work focused on replicating the code2seq model with added capabilities such as predicting natural language and traversed ASTs, From our results we see that both our methods do perform as well as the baseline in terms of the BLEU-4 scores. In this chapter, we describe our proposed model and show the obtained results also we present the OpenNMT approach and how does it work and show the results obtained by OpenNMT.

General conclusion

In this study, we have explored the concept of producing autocomments and investigated its potential applications and limitations. Through our investigation, we have observed that autocomments generation can be a powerful tool in multiple domains, including customer assistance, code documentation, social media interactions, and more.

To achieve this purpose, we first offered a complete explanation of Neural Machine Translation in Chapter 1. We looked into the underlying concepts and procedures of NMT, including code-to-sequence architectures, attention mechanisms, and training methodologies. This chapter established an excellent comprehension of NMT, developing the framework for this study.

In Chapter 2, we focused on source code and the documentation and analysis source code methodologies. We described alternative documentation approaches and proposed the concept of Abstract Syntax Tree (AST) and Structure Based Traversel(SBT). Understanding the analysis source code and AST was essential for comprehending the future chapters and their contributions.

Chapter 3 presented our proposed models and the process of generating comments from source code. We discussed the novel approaches and methodologies employed in this research. Additionally, we reviewed related work, highlighting existing studies and approaches relevant to autocomments generation from source code. This chapter showcased the advancements and contributions made in the field.

In Chapter 4, we provided implementation details, experimental approach, and the validation methodology applied in this project. We addressed the frameworks, tools, and libraries utilized to design and execute our proposed models. In addition, we showed the results derived from the suggested technique and OpenNMT training framework with the Deepcom dataset. Results are demonstrating the efficacy and performance of the our approach .

Although our study exhibited promising results in producing autocomment from Java source code, further refining of the NMT models can boost their efficiency. Future work can focus on researching advanced NMT architectures, including domain-specific information, and optimizing training procedures to increase the quality and accuracy of the generated auto comments. However, the proposed approach was much better then deepcom. With the first expirement as our best method, an important takeaway is that ASTs traversed generally hold the problem of Out of Vocabulary affecting the training process and a suitable AST generation technique. Nevertheless, looking at the comments generated by our best performing model , we see that our proposed approach is capable of understanding the syntactic and semantic meaning of java codes to generate comments automatically.ur model outperforms the state-of-the-art approaches and achieves better results on the automatic measure metric named BLEU.

In conclusion, this work seeks to contribute to the field of automatic comment generation by exploring the application of NMT techniques to generate comments from Java code snippets. The findings and insights gained from this research can potentially improve the efficiency and quality of software development processes.

Bibliography

[1] Rene Y Choi, Aaron S Coyner, Jayashree Kalpathy-Cramer, Michael F Chiang, and J Peter Campbell. Introduction to machine learning, neural networks, and deep learning. *Translational Vision Science & Technology*, 9(2):14–14, 2020.

[2] Krishan K Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No. 02CH37318)*, pages 235–241. IEEE, 2002.

[3] C.C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing, 2018.

[4] S Zargar. Introduction to sequence learning models: Rnn, lstm, gru. *no. April*, 2021.

[5] Jingwen Wang, Wenhao Jiang, Lin Ma, Wei Liu, and Yong Xu. Bidirectional attentive fusion with context gating for dense video captioning. *CoRR*, abs/1804.00100, 2018.

[6] Edmund Wong, Taiyue Liu, and Lin Tan. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 380–389. IEEE, 2015.

[7] Kishore Papineni Salim Roukos Todd Ward and John Henderson Florence Reeder. Corpus-based comprehensive and diagnostic mt evaluation: Initial arabic, chinese, french, and spanish results. 2002.

[8] D. Sarkar. Implementing code source methods and fonction engineering for text data: The skip-gram model - kdnuggets, n.d.

[9] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[10] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2009.

[11] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210. ACM, 2018.

[12] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[13] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 101–110. ACM, 2011.

[14] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 71–80. IEEE, 2011.

[15] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *arXiv preprint arXiv:1808.01400*, 2018.

[16] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, Jun 2019.

[17] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL ’02, pages 311–318, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[18] An Nguyen Le, Ander Martinez, Akifumi Yoshimoto, and Yuji Matsumoto. Improving sequence to sequence neural machine translation by utilizing syntactic dependency information. In

Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 21–29, Taipei, Taiwan, November 2017. Asian Federation of Natural Language Processing.

[19] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[20] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. Opennmt: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810*, 2017.

[21] Chris Callison-Burch, Miles Osborne, and Philipp Koehn. Reevaluation the role of bleu in machine translation research. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, 2006.

[22] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016.

[23] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.

[24] resentation du language python,. *http://www.linux-center.org/articles/ 9812/python.html*, page 36, 2020.

[25] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*, volume 620. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[26] Mark Summerfield. *Programming in Python 3: a complete introduction to the Python language*. Addison-Wesley Professional, 2010.

[27] Ekaba Bisong and Ekaba Bisong. Google colaboratory. *Building machine learning and deep learning models on google cloud platform: a comprehensive guide for beginners*, pages 59–64, 2019.

[28] Murad Khan, Bilal Jan, and Haleem and Farman. Visual studio code free software directory. *Free Software Foundation*, page 42, 2017.

[29] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *Osdi*, volume 16, pages 265–283. Savannah, GA, USA, 2016.

[30] Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. Augmenting java method comments generation with context information based on neural networks. *Journal of Systems and Software*, 156:328 – 340, 2019.