MASTER

Transformer-based Source Code Description Generation
An ensemble learning-based approach

Mantzaris, Antonios

Award date:
2022

Awarding institution:
Royal Institute of Technology

Link to publication

KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Transformer-based Source Code Description Generation

An ensemble learning-based approach

ANTONIOS MANTZARIS

Stockholm, Sweden, 2022

Transformer-based Source Code Description Generation

An ensemble learning-based approach

ANTONIOS MANTZARIS

Master’s Programme, ICT Innovation, 120 credits
Date: June 20, 2022

Supervisors: Amirhossein Layegh, Fatemeh Rahimian
Examiner: Amir H. Payberah
School of Electrical Engineering and Computer Science
Host company: Research Institutes of Sweden, RISE
Swedish title: Transformatorbaserad Generering av Källkodsbeskrivning
Swedish subtitle: En ensemblemodell tillvägagångssätt

Abstract

Code comprehension can be significantly benefited from high-level source code summaries. For the majority of the developers, understanding another developer's code or code that was written in the past by them, is a time-consuming and frustrating task. This is necessary though in software maintenance or in cases where several people are working on the same project. A fast, reliable and informative source code description generator can automate this procedure, which is often avoided by developers. The rise of Transformers has turned the attention to them leading to the development of various Transformer-based models that tackle the task of source code summarization from different perspectives. Most of these models though are treating each other in a competitive manner when their complementarity could be proven beneficial. To this end, an ensemble learning-based approach is followed to explore the feasibility and effectiveness of the collaboration of more than one powerful Transformer-based models. The used base models are PLBart and GraphCodeBERT, two models with different focuses, and the ensemble technique is stacking. The results show that such a model can improve the performance and informativeness of individual models. However, it requires changes in the configuration of the respective models, that might harm them, and also further fine-tuning at the aggregation phase to find the most suitable base models' weights and next-token probabilities combination, for the at the time ensemble. The results also revealed the need for human evaluation since metrics like BiLingual Evaluation Understudy (BLEU) are not always representative of the quality of the produced summary. Even if the outcome is promising, further work should follow, driven by this approach and based on the limitations that are not resolved in this work, for the development of a potential State Of The Art (SOTA) model.

Keywords

Natural Language Processing, Code Summarization, Transformers, Text Generation, Ensemble Learning, BLEU

Sammanfattning

Mjukvaruunderhåll samt kodförståelse är två områden som märkbart kan gynnas av källkodssammanfattning på hög nivå. För majoriteten av dagens utvecklare är det en tidskrävande och frustrerande uppgift att förstå en annan utvecklares kod.. För majoriteten av utvecklarna är det en tidskrävande och frustrerande uppgift att förstå en annan utvecklares kod eller kod som skrivits tidigare an dem. Detta är nödvändigt vid underhåll av programvara eller när flera personer arbetar med samma projekt. En snabb, pålitlig och informativ källkodsbeskrivningsgenerator kan automatisera denna procedur, som ofta undviks av utvecklare. Framväxten av Transformers har riktat uppmärksamheten mot dem, vilket har lett till utvecklingen av olika Transformer-baserade modeller som tar sig an uppgiften att sammanfatta källkod ur olika perspektiv. De flesta av dessa modeller behandlar dock varandra på ett konkurrenskraftigt sätt när deras komplementaritet kan bevisas vara mer fördelaktigt. För detta ändamål följs en ensemble-inlärningsbaserad strategi för att utforska genomförbarheten och effektiviteten av samarbetet mellan mer än en kraftfull transformatorbaserad modell. De använda basmodellerna är PLBart och GraphCodeBERT, två modeller med olika fokus, och ensemblingstekniken staplas. Resultaten visar att en sådan modell kan förbättra prestanda och informativitet hos enskilda modeller. Det kräver dock förändringar i konfigurationen av respektive modeller som kan leda till skada, och även ytterligare finjusteringar i aggregeringsfasen för att hitta de mest lämpliga basmodellernas vikter och nästa symboliska sannolikhetskombination för den dåvarande ensemblen. Resultaten visade också behovet av mänsklig utvärdering eftersom mätvärden som BLEU inte alltid är representativa för kvaliteten på den producerade sammanfattningen. Även om resultaten är lovande bör ytterligare arbete följa, drivet av detta tillvägagångssätt och baserat på de begränsningar som inte är lösta i detta arbete, för utvecklingen av en potentiell SOTA-modell.

Nyckelord

Naturlig språkbehandling, Kodsammanfattning, Transformatorer, Textgenerering, Ensemble Lärande, BLEU

Acknowledgments

Firstly, I would like to thank my family for their support during all of my studies, which included not only financial assistance, but many other forms of support that made this journey possible and also enjoyable. My examiner, Amir H. Payberah, deserves special recognition for the opportunity he gave to me to work on this project and his support and guidance, starting with his courses, and moving to the establishment and final implementation of the master thesis. As contributing members, I would like to express my gratitude to RISE and particularly Fatemeh Rahimian, but also to my supervisor from KTH, Amirhossein Layegh. Finally, many thanks to EIT Digital Master School and its partners, the Eindhoven University of Technology and KTH Royal Institute of Technology, for organizing this program and offering top-class education.

Stockholm, June 2022
Antonios Mantzaris

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

AdaBoost	Adaptive Boosting
API	Application Programming Interface
AST	Abstract Syntax Tree
AWGN	Additive White Gaussian Noise
BiLSTM	Bidirectional LSTM
BLEU	BiLingual Evaluation Understudy
BPE	Byte Pair Encoding
CAP	Code-AST Prediction
CASS	Context Aware Semantics Structure
CNN	Convolutional Neural Network
ConvGNN	Convolutional Graph Neural Networks
DPR	Dense Passage Retriever
EP	Edge Prediction
GeLU	Gaussian Error Linear Units
GNN	Graph Neural Network
GPU	Graphics Processing Unit
GRU	Gated Reccurrent Unit
IP	Identifier Prediction
IR	Information Retrieval
KTH	KTH Royal Institute of Technology
LSTM	Long Short Term Memory
MASS	Masked Sequence to Sequence
MCL	Multi-modal Contrastive Learning
MLM	Masked-Language Modeling
MNG	Method Name Generation
MVN	Multi-View Network

NA	Node Alignment
NL	Natural Language
NLP	Natural Language Processing
NMT	Neural Machine Translation
NN	Neural Network
OOV	Out Of Vocabulary
PD	Path Decomposition
PDG	Program Dependence Graphs
PL	Programming Language
POT	Pre-Order Traversal
ReLU	Rectified Linear Units
RISE	Research Institutes of Sweden
RNN	Reccurrent Neural Network
RTD	Replace Token Detection
SBT	Structure Based Traversal
seq2seq	Sequence-to-Sequence
SiT	Structure-induced Transformer
SOTA	State Of The Art
SPT	Simplified Parse Tree
SVM	Support Vector Machines
TEP	AST Edge Prediction
URL	Uniform Resource Locator
XFG	ConteXtual Flow Graph

Chapter 1

Introduction

Source code summarization refers to the use of natural language to describe a code snippet. This description can tell users about the code's functionality, easing the software maintenance process in which code comprehension is the most time and energy intensive task [1]. However, such summaries are frequently missing from the code or are out-of-date. The reason for this is that summary is the least creative activity a developer has to deal with, hence it is frequently overlooked or performed haphazardly. The good news is that this method is automatable. Automated source code summarization is a popular software engineering research topic wherein machine translation models translate code snippets into relevant natural language descriptions. The topic has an increasing interest in the last 10 years with different approaches developed following the at the time dominant technologies. Like any automation procedure, source code description generation increases programmers' productivity by reducing their workload and giving them more time to solve more significant problems than code documentation. Thus, this research may have an impact on all programmers of the world and the companies that they are involved with.

Producing natural language out of source code is a challenging task. It differs from standard translation from one natural language to another because source code and natural language summaries are heterogeneous. They could present differences in lexical or syntactic level. Moreover, the special structure of source code, if ignored, can lead to inability of the developed model to learn its semantics. This is crucial since it can mean the misinterpretation of the code and hence the production of an irrelevant comment. Issues like that, are approached by the at the time dominant framework of the more generic field of research, leading to the increase of attention to the Transformers [2] in our

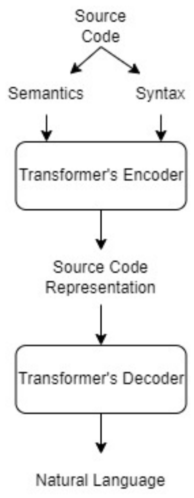


Figure 1.1: Transformer-based Source Code Summarization

days. Many other techniques have been suggested, all of them contributing on this evolution.

In this master thesis, several approaches related to source code description and general-purpose code snippets' representations generation will be discussed. Moreover, a new Transformer-based approach will be suggested based on work done so far and their limitations. A Transformer-based model's general structure is depicted in Figure 1.1.

1.1 Background

Transformers, initially presented by [2], are sequence-to-sequence neural network architectures that are built on top of Reccurrent Neural Networks (RNNs) [3, Chapter 10] inabilities. These inabilities or limitations of RNN-based architectures are the non-parallelization of computation because of their sequential format, recursion is expensive and despite the Gated Reccurrent Units (GRUs)' and Long Short Term Memorys (LSTMs)' [4] contribution on capturing longer range dependencies and avoiding vanishing or exploding gradients, not all relationships are still captured. To that end, the encoder-decoder architecture of the Transformers was introduced alongside self-attention mechanisms that are capable of capturing short and long-range relationships among the input and target data. This allows the contextual representation of the data which leads to semantically more accurate code to natural language translations. Due to the complex and bipolar nature of code, meaning that it provides information both from its text but also structure and syntax, different and additional to the main transformers' architecture techniques have been proposed, mainly trying to incorporate code snippet's syntax in the developed model [5, 6]. This is most of the times tackled with the usage of Abstract Syntax Trees (ASTs) [7] which can provide information about the code's syntax, or its variants such as Data Flows [8]. Many models have been developed following these characteristics and have indicated positive results. A more thorough mentioning and analysis of them occurs in 2.8.

Ensemble learning [9] on the other hand, refers to the aggregation of different models/estimators, with the motive of ”wisdom of the crowd”. This way diversity is added, helping the avoidance of local optima and overfitting models. Different models are used in a complementary manner rather than a competitive one allowing one model to overcome another model's limitations and vice-versa. Ensemble learning [10] can be divided into three major categories based on the dependency of their used models, data usage and models' aggregation. These categories are bagging, stacking and boosting, and for reasons that are explained later, stacking is preferred in the suggested approach.

1.2 Problem

All the already developed approaches have their positives and negatives. After a thorough research, the most common limitations are highlighted and summarized. This way the problem to be investigated and its tackling methodology are determined.

Statistical, RNN, Convolutional Neural Network (CNN) and Information Retrieval (IR) based models will not be further investigated because of their fundamental limitations that are mentioned in their respective sections in 2.8. Concerning Transformer-based models the main problems noticed go as follows.

First problem is the quantity of data. Complex Deep Learning architectures such as the Transformers, can benefit from large amounts of data. However, most of the existing approaches require a bilingual corpus in the code snippet-comment pair format which are of limited size. In our case, large scale datasets consisting of source code snippets could be used for the understanding and eventual representation of them. Such pre-trained models could provide useful representations of the syntactically and semantically complex code snippets that could be fine-tuned for several downstream tasks, including the task of summarization. Such representation models already exist but have significant limitations. Specifically, they should be trained on Programming Language (PL) data with PL-related tasks. Additionally, the structure should be considered and also since the wanted outcome is a generative, the pre-training tasks should be generative as well.

The second problem is related to the structure capturing of the code. Many researchers have used ASTs to capture the structure and showed some improvement compared to the usage of code as plain text. However, [5] has shown that linearized ASTs incorporated into Transformers bring no significant improvement to the model as they do with RNN-based models. This is probably because of the non-sequential functioning of the self-attention mechanism which is one of the main parts of any Transformer. Instead, they suggest incorporating structure within the self-attention mechanism. Others such as [6, 11, 12] have also suggested the usage of matrices as an alternative of the linearized AST.

Third and final issue is the usage of absolute positional embeddings. Methods like [13, 14, 15] are using a relative positional attention which is proven more effective but also efficient. The second and third problem can be combined into one since their confrontation can be concurrent with the development of an appropriate mechanism.

1.3 Research Questions

The research questions of this thesis are:

- Can the aforementioned problems be tackled at the same time by ensembling complementary models?
- What are the necessary steps for using ensemble learning on the task of source code description generation?

1.4 Goals

All of the problems in 1.2 have been tackled from several approaches but individually and/or for different tasks. In this thesis, the goal is to develop a model inspired by previous work, aiming at tackling these. Specifically, Transformers such as PLBART [16] could be used concerning the first problem for the production of useful source code representations and eventual fine-tuning on the task of summarization. On the other hand, GraphCodeBERT [15] uses the code structure through data flows that utilize relative positional information. Hence, inspired by LeClair's [17] proposal and encouraged by [18]'s optimistic results on their experimentation with ensemble models on a similar task, we aim at combining the newest, most promising and to the discovered problems-specific models for the creation of a Transformer-based source code description generation model.

Therefore, the goal of this project is to combine transformer-based models that are fine-tuned for the task of source code description generation. This has been divided into the following three sub-goals:

1. Modify the selected models so that their outputs are appropriate for aggregation.
2. Fine-tune the selected models.
3. Develop an ensembling learning model.

1.5 Research Methodology

The project is using the empirical method. Empirical research is based on analyses derived from personal experimentation while the analytical approach is based on mathematical calculations on existing experimental work. All models including their combination in the final ensemble model are investigated and tested with the same dataset to provide metrics that would lead to a valid and robust conclusions. For testing our hypothesis, liberty for experimentation on different models is required since the exact approach has not been tested yet by other researchers, thus an analytical approach would not be suitable.

Specifically, PLBart and GraphCodeBERT are both fine-tuned on the same dataset. GraphCodeBERT had not been fine-tuned yet for the task of code summarization. This was accomplished by adding a transformer's decoder since GraphCodeBERT's outcome is just the source code's representations, after using a bidirectional encoder [19]. These two models, based on their structure and pre-training techniques could tackle parts of the problems described in 1.2. Their complementarity was influential for their combination via the creation of an ensemble model. Finally, the need for a large amount of data accompanied by the outcomes of [20] lead to the usage of the stacking method.

1.6 Structure of the thesis

Chapter 2 presents relevant background information about the followed approach's main elements but also a thorough investigation on related work on this area. Chapter 3 presents the methodology used to solve the problem. It mostly consists of details on the used models. Chapter 4 informs about the dataset, evaluation metrics and development's environment but most importantly presents the major results of the research. This leads to chapter 5 where the results are further analysed and interpreted. Finally, in chapter 6 the outcomes of the project are summarized and presented but also the limitations, the suggested future work and a reflection on the project from several perspectives.

Chapter 2

Background

This chapter provides basic background information about source code summarization. Additionally, this chapter describes transformers, their usage for code summarization, and eventually ensemble learning. The necessary details about the theoretical framework but also the specific tools and models used are elaborated. This is crucial for the facilitation of the readability and understanding of the project in general, as well as the clarification of the reasoning behind the decision-making process. The chapter also refers to related work in the area of investigation providing several alternatives that have been proposed over the last ten years.

2.1 Source Code Summarization

As described in 1 source code's interpretation relies on both semantic and syntactic information [21]. Concerning the first one, the wanted information can be extracted by breaking the input code into smaller components such as words, sequences of characters, or just characters. This is called tokenization and it is described in 2.3. Then, the used model is responsible for finding relationships among the produced tokens which are preferably of long-range since in both programming and natural language two words can be not one next to the other and still have strong a correlation [22]. To this end, architectures like LSTMs and Transformers have been used, which are able to capture long-range relationships by utilizing the attention mechanism to produce context-aware representations of the input. Transformers and Attention are described in 2.4.2 and 2.4.1 respectively. Concerning the syntax, ASTs are widely used and many researchers have attempted to incorporate them in transformer-based models [23, 24, 25, 12, 6, 26, 5, 15]. Derived from ASTs, data flows have been introduced, representing relationships between AST nodes [27].

2.2 Code Structure

The main way to extract information about the structure of a code snippet is getting its AST. An AST is a representation of the code in a unique way so that an AST would correspond to just its code. It is tree-structured where the edges correspond to the hierarchy of the code. Concerning the nodes, the leaves are called terminals and represent identifiers whereas the non-leaf nodes are called non-terminals and represent structures such as loops, expressions and variable declarations. In this work, data flows are used, therefore for further information on ASTs someone can refer to [7]. However, we provide a simple example that can clarify the structure and usability of ASTs and it is split in the example Code 1 and its produced AST Figure 2.1.

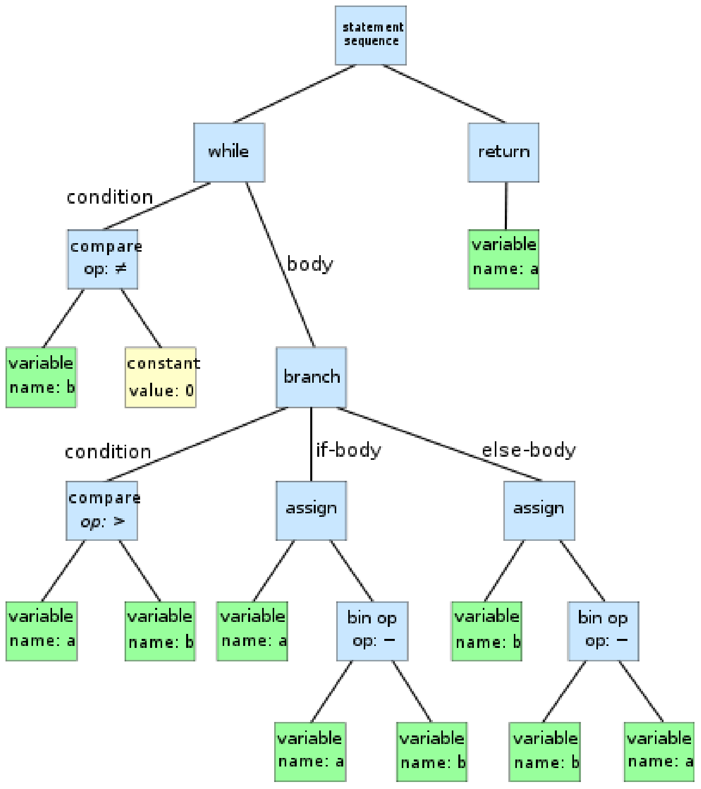


Figure 2.1: AST representation of Listing 1

Listing 1 Example code for AST 2.1

```
while b != 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```

Data Flow

Data flows are graphs representing the relationships between variables. The nodes correspond to the variables themselves while the edges indicate the relationships among them [8]. Data flows are able to extract additional semantic information out of the code and its structure by exploiting these relationships. In contrast to the traditional ASTs, data flows are simpler, more concise and more robust to small syntactical changes in the code that produce no altered outcomes. The construction of a data flow given a code snippet C is given in Algorithm 1.

Algorithm 1 Data Flow Algorithm

```
Input: $C = \{c_1, c_2, ..., c_n\} code\ snippet$
    1. Parse code into an AST.
    2. Leaves of the AST are the variables/nodes $V = \{v_1, v_2, ..., v_k\}$ of the graph.
    3. The edges are defined as $V = \epsilon = (v_i, v_j)$, if $v_j$ comes from $v_i$.
    4. All the directed edges acquired from step3 are forming $E = \{\epsilon_1, \epsilon_2, ..., \epsilon_l\}$.
    5. $G(C) = (V, E)$.
```

2.3 Tokenization

Word tokenization is essential in Natural Language Processing (NLP), hence also in source code summarization. The given text is split into its components/words in order to form the input of the model. This would look as follows: $inputsentence =$ “import pandas as pd”, $tokenized\ sentence = [import, pandas, as, pd]$. Before that, the vocabulary

is defined based on the distinct words of a large corpus. The produced vocabulary consist of the known words that will eventually carry some semantic information that can be used for any NLP task. However, a common issue in occasions like this is, facing Out Of Vocabulary (OOV) words. An example illustrating this issue is the following:
$vacabulary = \{..., play, playing, ...\}$. Let "played" not in vocabulary. If played is within an input sentence, it will be marked as unknown, since it does not belong in the vocabulary. A method to overcome this issue is sub-word tokenization where during the construction of the vocabulary, the words are split into sub-words, "playing" to "play", "ing", leading to more flexible vocabulary that could handle unknown words such as "played". The way this is done can vary. Some of the most used techniques are Byte Pair Encoding (BPE) [28], Byte-level BPE, Unigram [29], WordPiece [30] and SentencePiece [31]. In this project the SentencePiece method was used, specifically pre-trained on PLBart using both programming and natural language.

SentencePiece was mostly chosen because of the Programming Language part. The reason is that SentencePiece is the only tokenizer which does not assume that the input words are split by space. This makes it useful for languages like Chinese, Japanese, Thai or Code for which there had only been developed only language specific tokenizers so far. To this end, SentencePiece treats spaces as character and then uses BPE or Unigram.

On BPE the initial vocabulary consist of all of the characters seen in the corpus. Then, the frequency of all possible symbol pairs is calculated and the most frequent pair is merged and added in the vocabulary. The same procedure is repeated until the pre-determined vocabulary size is reached. Then Byte-level BPE is a variant of BPE where byte-level sequences are used instead of character-level and is prefered on multi-lingual settings.

Unigram follows a different approach, starting from a large vocabulary and moving down to a smaller one of size equal to the wanted. This is accomplished by computing the log-likelihood when a symbol is removed. The symbols increasing the log-likelihood loss the less are eventually removed.

$$\sum_{i=1}^{N} log(\sum_{x \in S(x_i)} p(x)),$$

where $S(x_i)$ is the set of all possible tokenizations for word x_i.

However, Unigram is not used directly but along with WordPiece. WordPiece works similarly to BPE with the difference that instead of the frequency of a symbol pair, their likelihood is considered, which is given by the formula :

$$\frac{p(st)}{p(s)p(t)},$$

where s, t are the symbols to be merged.

2.4 Transformers

A Transformer is deep learning architecture mainly used in the fields of NLP and Computer Vision [32]. Nowadays, Transformers have taken over RNNs because of their ability to handle sequential data simultaneously. In this section the main mechanisms used in a transformer will be elaborated as well as its architecture, the ways that a code snippet's structure can be fed in it and finally some transformer-based models followed by a description of their decoding mechanisms for the tasks of translation and summarization.

2.4.1 Self-Attention Mechanism

The self-attention mechanism is responsible for finding relationships among the input tokens to facilitate the production of context-aware representations of them. These representations are in the format of an embedding which is the result of a mapping procedure of each token to a smaller m-dimensional space. Other ways to obtain these embeddings include Word2Vec [33], GloVe [34] and FastText [35]. In these approaches the embedding of a "word" is influenced only by its surrounding "words" or is totally independent. This is why, architectures like RNNs, LSTM, GRU and Transformers are enhanced with the addition of the attention mechanisms in order to capture information all along the input text.

Self-attention has three main components; Queries, Keys and Values. Query corresponds to the at the time input-token while Key corresponds to other tokens for which the correlation with the Query will be investigated. Queries and Keys will indicate the matching score between two tokens which will eventually be used as weight to the Values and consequently the final self-attention.

Firstly, a query (W^Q), key (W^K) and value (W^V) weight matrix is initialized randomly and multiplied to the input text's token embeddings matrix T. Let x be the number of distinct tokens within T. Then the output of the previous multiplication, Query, Key and Value matrices $Q, K \in \Re^{x \times d_k}$ and $V \in \Re^{x \times d_v}$ respectively, where d_k, d_v refer to the dimension of the key and value embedding of the tokens. The calculation of the attention goes as follows. At the first step we get the dot product $Q \cdot K^T$ to get the similarity between the words. In step two, the previous dot product is divided by the squared root of the key's dimension which helps on getting more stable gradients. After that, Softmax is applied to normalize the scores between 0 and 1. Finally, to get the Attention Matrix, the dot product of the previous output and the Value matrix is computed from which it is indicated the influence of one token to the other. The summarized formula is the following:

$$Softmax(\frac{Q \cdot K^T}{\sqrt{d_k}})V \tag{2.1}$$

In order to avoid specific words being dominated by specific others and to provide more balanced and well-rounded results that defy any possible bias it is possible to use multiple attention layers simultaneously. For each attention layer, different sets of Query, Key and Value matrices are taken. They all follow the aforementioned procedure and produce their respective Attention Matrix. All the Attention Matrices are eventually concatenated to produce the final Attention. This is called Multi-head Attention and it is given by the following formula:

$$MultiHead\ Attention = Concatenate(Z_1, Z_2, ..., Z_k)W_0, \tag{2.2}$$

where W_0 is a weight matrix, Z_i the Attention Matrices and W_i^Q, W_i^K, W_i^V $for\ i \in [1, k]$ the distinct weight matrices of the k-available attention heads.

2.4.2 Transformer's Architecture

Transformers belong to the encoder-decoder sequence-to-sequence networks [36]. Specifically, it consists of stacked encoders and decoders where the encoders are responsible for the production of representations of the input which are eventually fed to the decoders to generate the desired output. As can be seen in Figure 2.2, the output of each encoder and decoder is passed as input to the next encoder and decoder respectively. This only differs in the case of the last encoder and the first decoder of their stacks. The output of the last encoder is the representation of the input and is fed as input to the decoders. Concerning the first decoder, it receives as input the produced representation

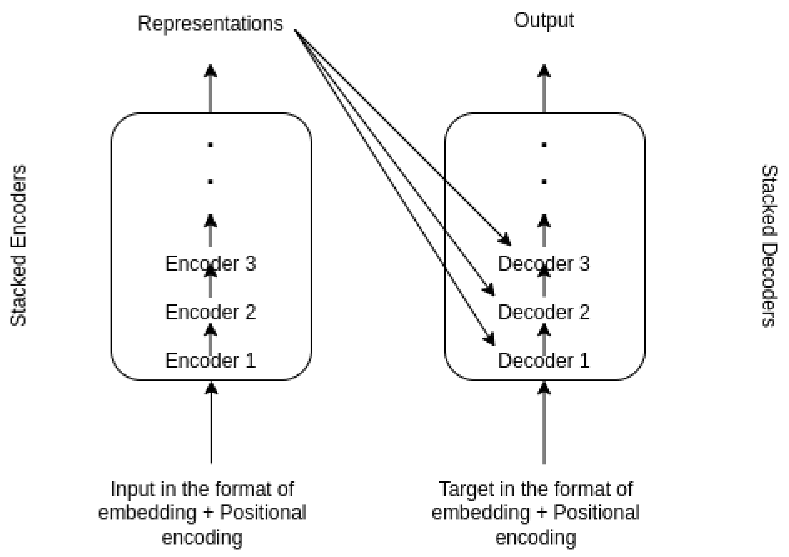

Representations
Output
Stacked Encoders
Encoder 3
Encoder 2
Encoder 1
Decoder 3
Decoder 2
Decoder 1
Stacked Decoders
Input in the format of embedding + Positional encoding
Target in the format of embedding + Positional encoding

Figure 2.2: Encoder-Decoder Sequence-to-Sequence Architecture

of the initial input, as implied from the encoders' side, but also the embedding of the target during the training phase. The rest decoders, similarly to the encoders stack, get information both from the produced representations and their previous decoder.

Positional Encoding

Moving into the details of this architecture, positional encoding that appears in the source and target input is responsible for providing order information. Transformers are processing data not sequentially like RNNs but in parallel, hence without any additional mechanism, the order of the input is lost. The positional encoding is in the format of a matrix that is summed to the input matrix which comprises all the input tokens' embeddings. The suggested from [2] way to that is Equation 2.3. After this step, the input is ready to be fed to the first encoder or decoder whose details are the next described subject.

$$\begin{aligned} P(pos, 2i) &= sin(\frac{pos}{1000^{\frac{2i}{d_{model}}}}) \\ P(pos, 2i+1) &= cos(\frac{pos}{1000^{\frac{2i}{d_{model}}}}) \end{aligned} \tag{2.3}$$

Encoder

Each encoder consists of a (multi-head) attention, an add and norm layer and a feedforward network. The attention mechanism and its role have been described in 2.4.1. The add and norm component then connects the input and output of each sublayer of an encoder. Its purpose is to "add" the input to the at the time sublayer and then "normalize" the output values with the scope of faster learning because of the avoidance of significant changes compared to its input. Finally, the feedforward layer constitutes two-thirds of a transformer's parameters and can capture patterns that will eventually be related to the target vocabulary [37].

Decoder

The decoders have a similar to the encoders' architecture and functionality. However, as mentioned previously in this section, they are taking into consideration both the target and the encoder's output. To that end, a decoder's block consists of an additional (multi-head) attention. The new attention mechanism is Masked (Multi-head) attention. As its name indicates, there are masks used at each step hiding the embeddings of tokens coming from the target and have not been generated yet. This is necessary in order to bring in compliance with the generation and training procedures. Practically speaking, this is happening by changing the values of the masked tokens in the attention matrix to $-\infty$. The rest components are the same as the ones described in the encoders' section.

Both encoder's and decoder's architectures are depicted in Figure 2.3

2.5 Transformer-based Models for Source-code Summarization

Among others, transformers have been also used for the task of source code summarization. Below, we present two of the transformer-based models that fit to this project's objectives and hence used later in our approach. These models are PLBart and GraphCodeBERT.

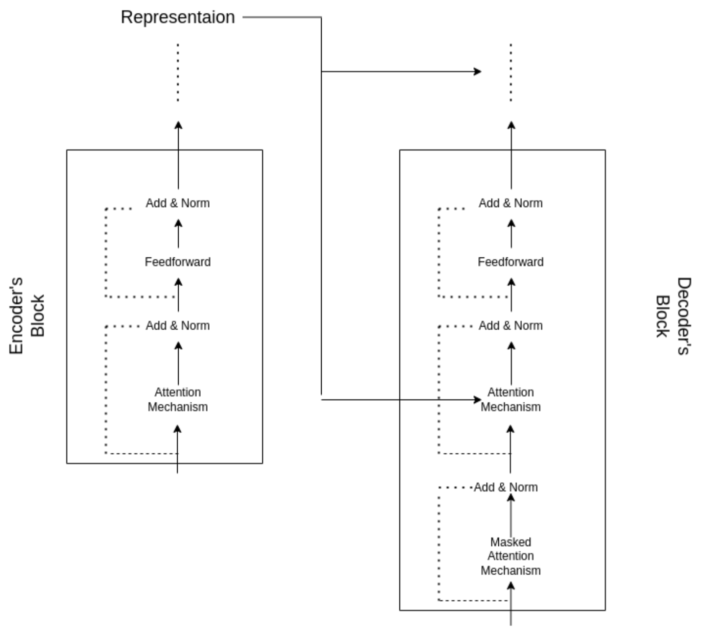


Figure 2.3: Encoder Block (left), Decoder Block (right)

2.5.1 PLBart

PLBart is inspired by BART [38] and was introduced by [16] to tackle program and natural language understanding and generation tasks. The element that differentiates it from other models is its capability to be pre-trained on unlabelled data. This allows the learning of representations that are attained from vast data sources of different data types, in this case source code and natural language. This learning can be then transferred and used for various applications. A second characteristic that makes PLBart powerful is related again to its pre-training but now refers mostly to its architecture. PLBart is an encoder-decoder sequence-to-sequence model. This means that the model which is meant to be used for several downstream tasks has an already pre-trained decoder. Most of the approaches, besides PLBart, that aim the production of general purpose representations, require an additional decoder that needs to be trained from scratch. This implies the need of more data and time for the training of the decoder.

These characteristics are complementary of the ones of GraphCodeBERT and are going to be described in the next subsection, so this is why the combination of these two models could be proven beneficial.

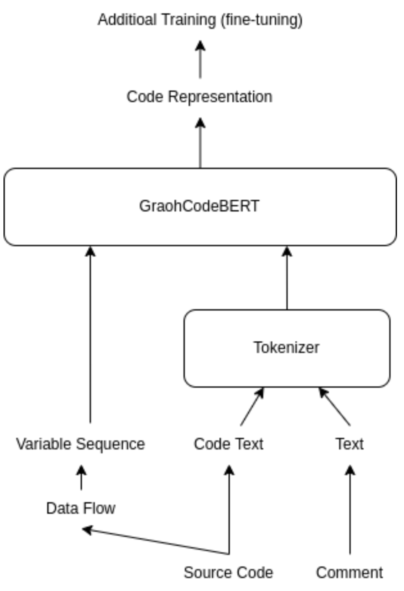


Figure 2.4: GraphCodeBERT overview

2.5.2 GraphCodeBERT

Similarly to PLBart, GraphCodeBERT [15] produces general-purpose representations of the input code and then is fine-tuned properly for the at the time downstream task. It is another transformer-based model that introduces some new elements. These elements are mainly related to the structure of the code and the tasks that the model is pre-trained on. Concerning the structure of the code, GraphCodeBERT avoids the direct usage of ASTs that has been proven problematic [5, 12] but suggests the utilization of data flows. Compared to the ASTs, data flows were proved superior by the creators of GraphCodeBERT because of their simplicity, since Transformers' efficiency decreases as the input length increases. Concerning the second differentiating element, the model is pre-trained on three tasks, Masked-Language Modeling (MLM) as in BERT and two new structure-aware tasks, edge prediction and node alignment. A general observation is that GraphCodeBERT's main contribution is the efficient exploitation of the code's structure both in pre-training and also fine-tuning and inference. An overview of GraphCodeBERT is depicted in Figure 2.4

2.6 Text Generation via Deterministic Decoding

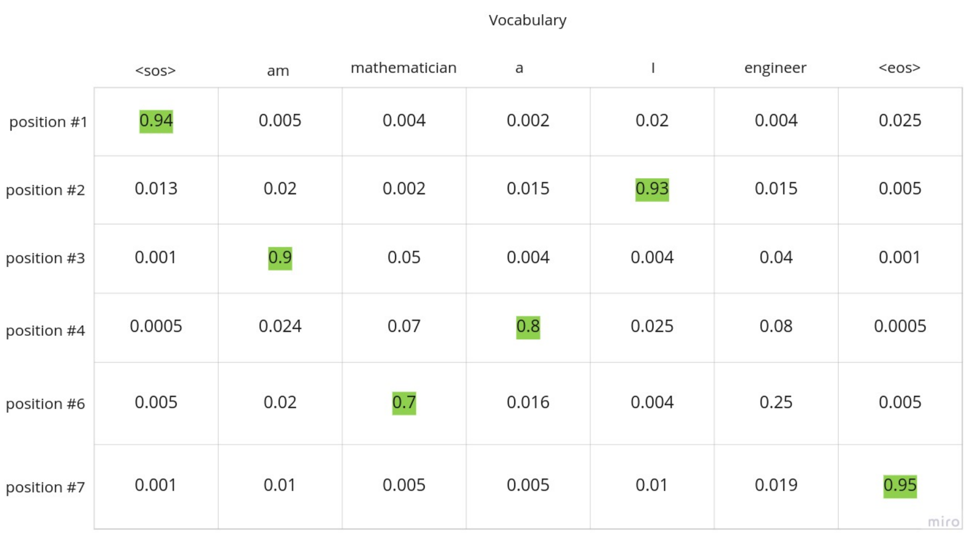
Once the models have been pre-trained and fine-tuned, it is time for them to generate the output. In code summarization, the final layer of a model is Softmax layer of the size of the input vocabulary. The softmax function is responsible for assigning to each of the tokens in the vocabulary probabilities that correspond to the likeness of the next word being that token. There are two main ways used in this project and are the Greedy and the Beam Search. In both occasions, the next word generation procedure comes to an end when an End-Of-Sentence token is generated. [39]

2.6.1 Greedy Search

Greedy search is the simplest but also naive way to select the next "word" in a text generation task. At each position of the output sentence, greedy search will take the "word" with the highest probability. This way it takes the optimal candidate of the time ignoring the previous or future outputs. It is simple though, implemented just by adding an argmax function on the softmaxed output and it has shown decent results with transformers when the output's length remains relatively short [40]. An example of the greedy search can be seen in Figure 2.5.

2.6.2 Beam Search

Beam search [41] on the other hand is inspired by greedy search but builds on its weaknesses. Instead of taking one token, the optimal of each time, multiple are considered based on conditional probabilities. Initially, the number of the tokens to be considered is defined and called beam width. For the prediction of the N^{th} token the $(N-1)^{th}$ selections of number equal to the beam width are taken and their best combinations with the tokens of the vocabulary are determined based on their conditional probabilities. Once again the top-|beam width| are selected. The procedure continues like this till an End-Of-Sequence token is generated. Increasing the beam size can on the one hand improve accuracy but on the other hand adds computational cost since more choices need to be considered every time. With this mechanism, local optima can be avoided since at any time, previously selected tokens can be dropped if the combination of other selected tokens with upcoming ones produce higher



	Vocabulary						
	<sos>	am	mathematician	a	I	engineer	<eos>
position #1	0.94	0.005	0.004	0.002	0.02	0.004	0.025
position #2	0.013	0.02	0.002	0.015	0.93	0.015	0.005
position #3	0.001	0.9	0.05	0.004	0.004	0.04	0.001
position #4	0.0005	0.024	0.07	0.8	0.025	0.08	0.0005
position #6	0.005	0.02	0.7	0.016	0.004	0.25	0.005
position #7	0.001	0.01	0.005	0.005	0.01	0.019	0.95

Figure 2.5: Greedy Search, output sentence: I am a mathematician

probabilities. In case the beam width is selected to be equal to one, then the beam search is identical to the greedy search.

2.7 Ensemble Learning

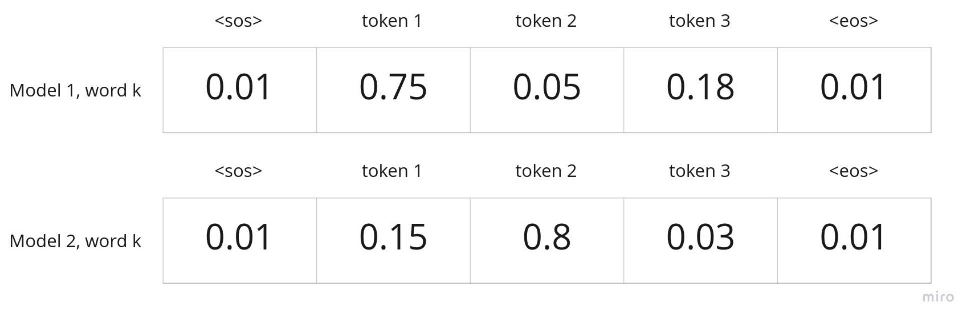
Ensemble learning is based on two basic ideas: different models could be good at different “parts” of data and individual mistakes of could be “averaged out”. The common ground of both ideas is that eventually several models are aggregated to produce a single output that aims at getting the most out of each model and correcting their weaknesses. Ensemble models has shown great performances in many areas [10] but also some signs of potential in code summarization specifically [17, 18, 42].

In ensemble models there are three factors to be considered. These are: the models to be combined, the data to be used for training and the aggregation technique. In this project the first two factors are pre-determined and there is some experimentation on the aggregation technique.

Concerning the models, the choices to be made are: what models should be used and on what degree of dependency. Initially, the models should be relatively uncorrelated but good enough in order for the ensemble to be meaningful. Then, the selected models could be dependent or independent to each other. When the models are dependent, the objective of a model is to fix the mistakes of the previous model. Such techniques belong to the general category of boosting [43] and two characteristic examples are Adaptive Boosting (AdaBoost) [44] and Gradient Boosting [45]. In the first one, models are obtained by re-weighting the training data at every iteration. Data points that were misclassified get higher weight so that the ensemble can focus on them and fix the misclassification. On the other hand, gradient boosting's objective at each iteration is to predict the residual error of the ensemble and try to reduce it. It is an additive model, meaning that the predictions at each iteration are summed and the base models are most of the times regression trees.

Moving to the second factor and the data to be used, there are two different approaches that can be followed. These are bagging and stacking. Bagging (Bootstrap Aggregating) [46] is the technique where the same model is trained on different parts of the dataset. This way each sub-model becomes an expert on their respective sub-set and then their predictions are aggregated to reduce this overfitting and generalize. The most famous bagging technique is the Random Forest [47] where randomized trees are aggregated by averaging their predictions and are usually very accurate. In stacking [48] though, there are different models, each trained on the entire dataset and at the end their predictions are aggregated. This approach aims at inducing diversity and tackling different parts of the same problem from different angles. On top of that, an additional model can be put to learn how to better combine the individual predictions. This meta-learner is called stacked generalization ensemble [49] and is efficient but also expensive.

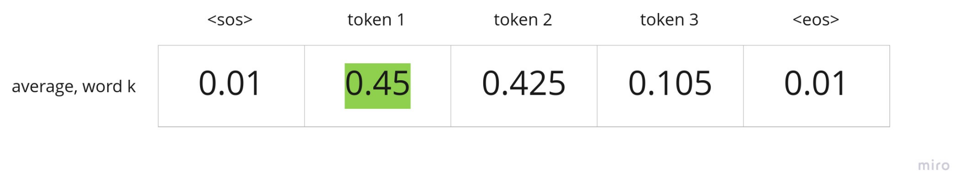
Finally, concerning the aggregation techniques, in the case of code summarization, it would happen at the softmax output of each model. There are several ways to do so, starting from simpler approaches such as using the average, weighted average or maximum and moving to more complex ones like employing Support Vector Machines (SVM) [50] or neural networks [51] that were mentioned before as meta-learners. Simple examples for the visualization of the aforementioned techniques are provided in Figures 2.7, 2.8, 2.9 and correspond to the initial values illustrated in Figure 2.6 (the examples are not realistic but over-simplified for demonstration purposes).



	<sos>	token 1	token 2	token 3	<eos>
Model 1, word k	0.01	0.75	0.05	0.18	0.01

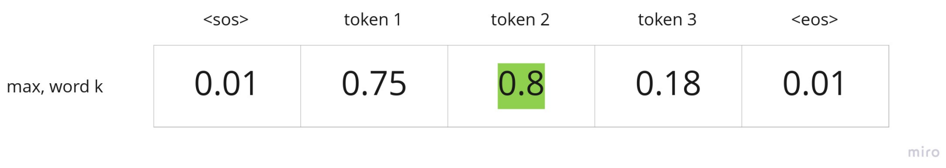
	<sos>	token 1	token 2	token 3	<eos>
Model 2, word k	0.01	0.15	0.8	0.03	0.01

Figure 2.6: Softmax output of two models for k^{th} word



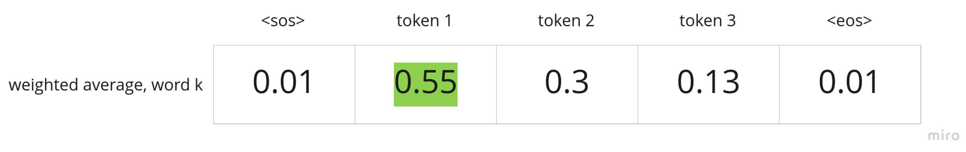
	<sos>	token 1	token 2	token 3	<eos>
average, word k	0.01	0.45	0.425	0.105	0.01

Figure 2.7: Selected token for k^{th} word of average is applied



	<sos>	token 1	token 2	token 3	<eos>
max, word k	0.01	0.75	0.8	0.18	0.01

Figure 2.8: Selected token for k^{th} word of maximum is applied



	<sos>	token 1	token 2	token 3	<eos>
weighted average, word k	0.01	0.55	0.3	0.13	0.01

Figure 2.9: Selected token for k^{th} word of weighted average is applied where 2 is the assigned to model 1 weight and 1 is the respective weight for model 2

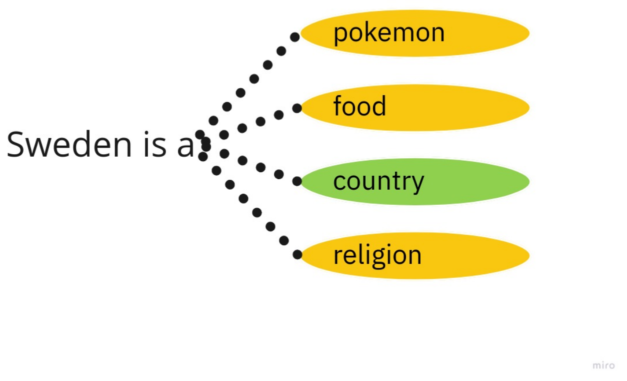


Figure 2.10: Next word prediction based on conditional probabilities. In this example there are only four possible choices derived from the available corpus.

2.8 Related Work on Source Code Summarization

Three main approaches have been followed for the objective of source code summarization. These are, statistical language models, IR or template-based models and Neural Network (NN) based models. Statistical language models are using statistics for making predictions. IR-based models are exploiting similar code snippets' comments. Finally, NN-based models are mostly following the encoder-decoder architecture and are approaching source-code summarization as a Neural Machine Translation (NMT) task.

2.8.1 Statistical Language Models

Statistical language models aim the prediction of comment from source code in a comments completion format. They belong to the n-gram Language Model family since they are using n-grams and conditional probabilities based on statistics to predict the next comment token 2.10 [52]. The first issue with this approach is sparsity. Increasing the size of n-grams (n) makes the problem even worse. Either case, there is a lack of generalization, this approach's capabilities are limited and there are more advanced methods that can generate more complete and quality documentation.

2.8.2 Neural Machine Translation

The rise of Deep Learning and Neural Networks accompanied by the increase of available data led the researcher to tackle the previous problems from an NN perspective.

Convolution-based Models

Several NN architectures have been utilized for the purpose of source code summarization, starting with convolution-based ones. [53] proposed a Convolutional Attention Network that produces concise method-like names out of source code. The convolutional network allows the learning of features while the attention mechanism is essential for capturing short but also long-range relations among them. A copy mechanism is also introduced, which identifies important source code tokens that are out-of-vocabulary and can be used as are in the summary by utilizing one of the two attention mechanisms.

LSTM-based Models

The previous Convolutional Attention Network produces a method-like name summary, something that [54] tried to fix by introducing CodeNN. CodeNN is a Neural Attention Model using the at the time thriving LSTMs [55] for the formation of the natural language text with attention for the selection of the content. The issues with this approach are the usage of one-hot encoding which limits the scalability and flexibility of the model but also the processing of the code as plain text which omits any structural information.

CodeSum [56] on the other hand, keeps structural information since the source code is input as AST sequences [7] which are obtained by using the Structure Based Traversal (SBT). This method though is sensitive to different programming styles and identifiers. Similar to CodeSum, Code2Seq [57] does not linearize the AST with a traversal but uses compositional paths over it. Then a similar encoder-decoder network with attention is used like in all seq2seq models. All encoder-decoder-based models so far generate the output sequence from scratch during testing, while the ground truth is missing and hence the prediction of the next words is based on the previously generated words. This can lead to sub-optimal decoders due to the exposure bias issue.

Therefore, a new approach is introduced, Hybrid2Seq + Deep Reinforcement Learning [58] which contains an AST-based LSTM and an

LSTM for structural and sequential representation of the code respectively, fused by a hybrid-attention mechanism. Then a pre-trained actor-critic network is employed for decoding based on the BiLingual Evaluation Understudy (BLEU) metric reward. This approach has limitations such as the experimentation only with python, the usage of only BLEU score as a reward and the absence of human evaluation. [59] exposed another issue with the aforementioned model and is related to the AST-based LSTM. Specifically, this LSTM is based on [60]'s tree-LSTM and its inability to handle trees with nodes containing an arbitrary number of ordered children. This is why Multi-way Tree-LSTM is suggested as an encoder, in which the AST sequence is fed to a bidirectional LSTM to utilize interactions among children and is followed by an attention-based LSTM decoder. The results show no significant improvement and additionally, the baseline models used are not the state-of-the-art since CodeNN [54] was proved worse than codeSum [56] and DeepCom [61], where the latter one was already modified by its authors to perform better [62].

Ast-attendgru [63] is another robust model that is regularly used as a baseline model by other researchers. They noticed that without clear identifiers, the models can not produce good summaries, thus it would be beneficial to learn the code's structure independently from its text. Code's text is treated as plain text and structure as a modified to the [61]'s AST in which all words are replaced by the special token ⟨OTHER⟩, known as SBT-AO and it focuses only on the structure. This method performs better when the methods' names do not clearly state what the methods do but worse otherwise. Similarly works the Hybrid-DeepCom [62], an expansion of DeepCom [61], which analyzes source code and its AST sequence obtained by SBT at the same time. After the encoding, a hybrid attention mechanism fuses the lexical and syntactic information in order to proceed with the comment generation. In addition, for handling out-of-vocabulary tokens, the identifiers are split according to the camel casing naming convention. Recent research introduces the hierarchical code representation for code summarization [64]. Specifically, the code and its AST are split into statement level and then into token and subtoken level. A hierarchical Bidirectional LSTM (BiLSTM) encoder is applied followed by an attention mechanism. Token representations are aggregated to form the statement representations. Once the hierarchical encoding is done an attentional decoder (unidirectional LSTM) produces the probability vectors of the respective code and AST which are eventually averaged to complete the at the time LSTM step. The informativeness of the produced summaries is higher than its comparative models but still the longer the code is the worse the summary gets and it also fails to generate content out of the given code.

Another interesting approach is CO3 [65] which performs dual learning in the tasks of code search and code summarization. Dual learning is inspired by [66] and refers to the in parallel learning of code summarization and generation tasks. Their hidden states' similarity is then used for the code retrieval task purposes.

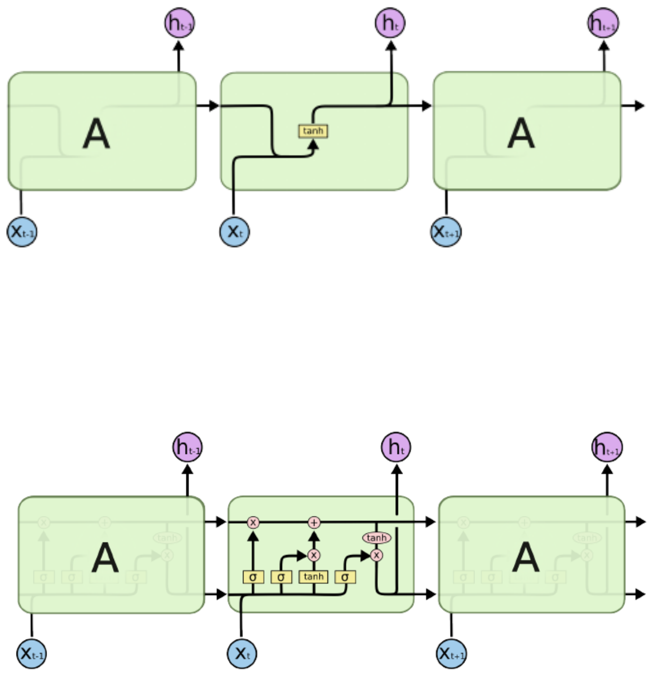


Figure 2.11: RNN vs LSTM [67]

Transformer-based Models

So far we have only mentioned CNN and RNN-based approaches. However, CNNs can not capture long-range dependencies and RNNs can not parallelize the computation because of their sequential computation. Based on these limitations and thanks to the development of the Transformers, transformer-based models were introduced.

Firstly, NeuralCodeSum [13] was developed and led the way to other transformer-based approaches. In addition to the regular self-attention mechanism, a copy attention mechanism is added to allow word generation from both the vocabulary and the source code. Furthermore, the positional encodings are not suitable for source code, hence the relative positional representation of pair of tokens is used [68], indicating better performance. ASTs were also tested but rejected because of the increase in the input sequence size which influences the transformers' complexity. This paper showed that transformers can learn the code's structure and the fact that the core architecture of Transformers is untouched, gives promising signs for further research. Then CodeBERT [69] and CuBERT [70] are two models based on BERT [19] and aim at the generation of general-purpose representations that can support downstream PL-Natural Language (NL) tasks. Quality source code embeddings are essential for tasks like code summarization or code search, as [71] claims by stating that better embeddings are needed for the full exploitation of their approach. CodeBERT has the same architecture as RoBERTa [72] but is trained with both bimodal (PL and NL) and unimodal (PL or NL) data on Standard MLM [73] and Replace Token Detection (RTD) [74]. CuBERT on the other hand, pre-trains a BERT-large by treating PL and NL as paired separated sentences. CuBERT was also used by [75] for producing embeddings carrying the method's functionality information. They show that fine-tuning CuBERT in a contrastive manner can lead to good clustering of code methods, however, this clustering is restricted to just 5 clusters/functionalities making the approach promising but not suitable for code summarization. Both CodeBERT and CuBERT showed interesting results and both mentioned the incorporation of structural code information as a promising future goal.

CodeTransformer [14] is also referring to the importance of relative positional self-attention but uses both context and structure (AST) for the encoding of the source code. They showed that especially in multilingual code summarization the combination of context and structure is beneficial since in this case, the structure can be crucial.

Towards this structure-exploitation direction, TranS^3 [76] was developed. It is used for unifying code summarization and code search by utilizing a Deep Reinforcement Learning Model. The actor-network is an encoder-decoder network where a tree-transformer is used as the encoder and is based on indent-based semantics. The results are showing improvement both over RNN-based or standard Transformer based encoders but the tree-transformer-based one relies significantly on the quality of the program form but also applies only to Python code. Another Python-associated method is PyMT5 [77] which treats source code and natural language the same, shared vocabulary, and tackles the method prediction and code summarization. However, it performs worse than TranS^3, which is anticipated because of its simplicity and omission of source code information.

Back to general-purpose code representations, GraphCodeBERT [15], which is also used in this project, suggests the utilization of data flow instead of ASTs to capture semantic rather than syntactic information since programmers do not always follow the naming conventions. The model is pre-trained on three tasks, MLM as in BERT and two new structure-aware tasks, edge prediction and node alignment. A comparison is also made with ASTs, where data flow was proved superior because of its simplicity since Transformers' efficiency decreases as the input length increases. More details about GraphCodeBERT can be seen in 2.5.2. Inspired by GraphCodeBERT, SynCoBERT [78] introduces Identifier Prediction (IP), AST Edge Prediction (TEP) and Multi-modal Contrastive Learning (MCL) to exploit identifiers' symbolic and syntactic information, the edge's between leaf and non-leaf syntactic information and avoid the dominance of high-frequency tokens respectively.

Transformers-based Models with Alternative Incorporations of Structure

Transformers though lose both computational efficiency and performance since ASTs are carrying noisy information. Moreover, incorporating a linearized AST into a Transformer has shown no performance gain because of the self-attention which acts more like a non-sequential process. These issues are tackled by [6] by not allowing information exchange among all nodes. At first, a Pre-Order Traversal (POT) is proposed over SBT and Path Decomposition (PD) for time efficiency. Then the linearized AST is used for the creation of two relationship matrices, ancestor-descendant and sibling, followed by multi-head attention on both and eventual concatenation. This way ASTs can be encoded and used without causing computational overhead. SG-Trans [11] uses matrices too. Specifically, token, statement and data flow (shallower than AST) adjacency matrices followed by their respective self-attentions and copy attention. An interesting element of this approach is the Hierarchical Structure-Variant Attention where the distribution of attention heads is not uniform but lower levels focus more on local structure (tokens, statements) while higher on the global structure. Similarly, Structure-induced Transformer (SiT) [12] expands ASTs to Multi-View Networks (MVNs) such as an abstract tree, control flow and data dependency, then represents them as adjacency matrices. Structure-induced encoder follows containing Structure-induced self-attention layers in order to finally be decoded by an original Transformer encoder. Another issue of ASTs is that they can be syntactically dense, leading to the memorization of syntax and not the learning of the semantics. [5] suggests five ways to pass ASTs into Transformers with the ”sequential relative attention” being the most effective and efficient because of its structure. Also [26] attempts to pass ASTs to Transformer by introducing PathTrans, based on root-paths, and TravTrans, a single AST traversal using pre-order or depth-first search and its variant which enhances the self-attention mechanism with a path-matrix. TPTrans-α [25] encodes both relative and absolute paths to capture patterns, relationships between nodes and program behavior respectively with a bi-directional GRU. Issues though with JavaScript snippets whose paths were larger indicate that additional work should follow. Another approach trained for method name prediction and tries to incorporate trees into Transformers is Meth2Seq [79]. They exploit Program Dependence Graphss (PDGs) such as control and data flow graphs for code semantic structure information, Intermediate Representations (IR) of statements for operational semantic information and Natural Language Comments for the explicit summarized semantics offered by developers in order to get hybrid code representations that can be used for several Programming Language related downstream tasks. Next approach [24] encodes ASTs via ConvGNN [80] to capture structural information and then Transformer-XL for semantic information. However, so far the authors considered only structural information as input something that they aim to change in the near future. MMTrans [23] also utilizes ConvGNN for capturing the local semantic information enhanced by the AST's SBT sequence which targets the global semantic information.

Another multi-modal (here code sequences, AST) Transformer-based model, inspired by Hybrid-DeepCom mentioned in the previous paragraph, is ComFormer [81]. They introduce Byte-BPE for handling the OOV words, SimSBT a simplified AST traversal and three different approaches to the encoding phase such as Jointly, Shared and Single encoder. Finally, Beam search is used for the comment generation. It shows promising results but the investigated baselines were limited especially concerning the Transformer-based approaches.

Transformer-based Models with Decoder Pre-Training

Moving to PLBART [16], the second model that was used for this project, it is a multilingual representation learning model. Its architecture is the same as BART-base [38] with the addition of an extra normalization layer on top of both encoder and decoder. Therefore, it is pre-trained with unlabeled data of both code and natural language on de-noising auto-encoding of different noise functions such as masking, deletion and infilling. For more information, you can refer to 2.5.1. SPT-Code [82] is another approach pre-trained on unlabeled data but just of code in order not to limit the dataset possibilities. In addition, they introduce pre-training for both the encoder and decoder on code-related tasks. These tasks are Masked Sequence to Sequence (MASS), Code-AST Prediction (CAP) and Method Name Generation (MNG) and the input dataset consist of the code, its AST linearized in a shorter version and its method name for Natural Language representation.

2.8.3 Information Retrieval-based Models

NMT-based approaches generate mostly frequent words and sometimes lose readability and informativeness. On the other hand, IR-based models are not enough by themselves since due to limitations on retrieval corpus, the retrieved snippet could be not semantically similar. Therefore, a combination of both is required for the production of good summaries.

Re^{2}Com [83] is one of them and it comprises two modules, Retrieve and Refine. Retrieve module gets the most similar in the lexical level code snippet by using BM25 metric [84] and Lucene open-source search engine [85]. The code, its AST and its similar code accompanied with its comment are input into the Refine module consisting of four BiLSTM encoders and following a Sequence-to-Sequence (seq2seq) NN, the comment is generated, based on the semantic similarity of the code snippets. A non-linear sigmoid function gives the similarity and according to it, the decoder can pay more attention to the retrieved information or the input code. Another approach follows named Rencos [86]. At first, an attentional encoder-decoder model is trained. Then, given an input, the most similar snippets are returned based on syntactic and semantic similarity via their ASTs or the embedding produced by the pre-trained model respectively. Finally, the pre-trained model encodes first and then decodes the input and the two retrieved snippets simultaneously taking always into consideration the similarity. The similarities here are calculated by the Lucene engine and cosine-similarity for syntactic and semantic level respectively.

Next model is REDCODER [87]. Similarly, a retriever and a generator are used but in a more complex, compelling way. The retriever, which is Dense Passage Retriever (DPR) based [88], uses the dot product for similarity metric on the embeddings obtained from two encoders, specifically CodeBERT [69] and GraphCodeBERT [15]. Then, the retrieved sequences are concatenated with the input sequence and fed to PLBART [16] which makes the final estimation. Hybrid-GNN comes next and aims at the combination of the benefits of retrieval and generation-based models but also the dealing with Graph Neural Networks (GNNs)' inability to capture global information. Once again similar snippets are retrieved, further encoded with a BiLSTM follows and an attention-based dynamic graph that captures global interactions. Both are fed to the Hybrid-GNN for static and dynamic message passing, eventually concatenated and fed into an attention-based LSTM decoder. At the limitations of this approach, we could include the non-semantic nature of the retrieval mechanism, which is also suggested as future work from the authors.

Then [89], a relatively different IR-based approach, re-uses [83]'s retrieval method followed by a cross-encoder which concatenates input and retrieved code and encodes them using CodeBERT [69]. The produced embedding is fed to a classifier whose output indicates the semantic similarity. If the retrieved snippet is semantically similar to the input, then its comment is used, otherwise, DeepCom [61] is used without though passing any information obtained by IR in order to exclude textually similar results that would confuse the model. The possible not usage of the NMT model makes this approach faster but at the same time could lead to semantically similar documentation but not adjusted to the input code. This could confuse the user since non-conventional naming is often used. In the same category belongs EditSum [20]. The difference with the previous suggestions is that it uses the retrieved snippet's comment as a prototype which it tries to revise based on the differences between the input and retrieved snippets. It is better at predicting low-frequency words but still, the initial retrieval is based on lexical similarity and the not-so-high usefulness levels indicate potential redundant or not correct information. The IR-based models presented in this paragraph may have two issues. They either use lexical similarity at their IR part [83, 89, 90] or use semantic similarity but in an online code retrieval manner [86, 87] which makes them expensive.

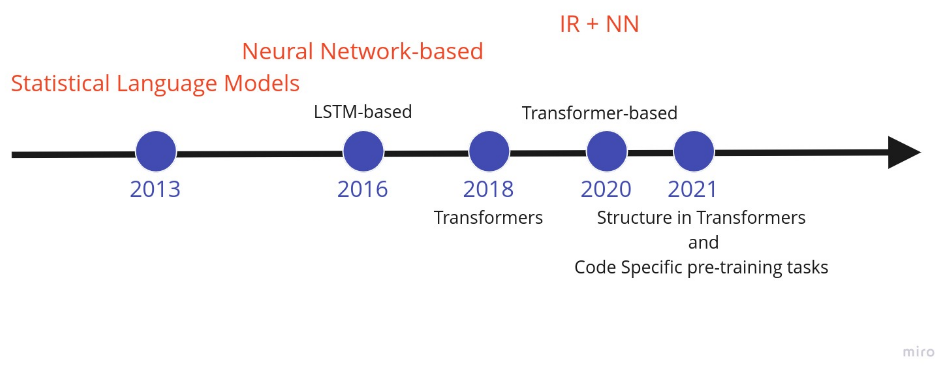


Figure 2.12: Timeline of source code summarization evolution

2.8.4 Non-Conventional Approaches

In addition, we present some techniques that are not following the standard NN or IR-based approaches but are worth mentioning.

The first one uses graphs, as they better represent code than any sequence [80] and specifically Convolutional Graph Neural Networks (ConvGNN) to encode the AST's nodes and edges. The efficiency of this method is one of its differentiation factors since Themisto [91] uses it for its deep learning-based approach because GNN-based's inference time was significantly better than other transformer-based such as BERT, T5, or GPT-3.

The next one is called ContraCode [92] and is a self-supervised contrastive learning [93] algorithm. Self-supervision allows better generalization while the contrastive algorithm learns representations by trying to minimize the distance between similar items and maximize the distance between dissimilar. It produces useful representations but the improvement compared to the Transformers is not significant because of the inconsistency in names by the programmers and the used metrics.

Then, from MISIM [94] we can consider their Context Aware Semantics Structure (CASS) which is configurable and can tackle previous syntax or semantic-based code representations' limitations. Concerning the syntax-based, such as ASTs, they may lead to the memorization of syntax rather than the learning of the semantics. Semantic-based representations such as ConteXtual Flow Graph (XFG) [95] and Simplified Parse Tree (SPT) [96] have also limitations. XFGs can only be used on compilable code and SPTs are structurally-driven and hence may capture irrelevant to the code semantics information.

LeClair [17] proposed treating the different models in a complementary manner than a competitive one so that the capabilities of all models are exploited increasing the data source diversity and avoiding overfitting and local minima. Therefore, some stacking and bagging techniques were researched emerging with some interesting results.

Inspired by the tree/graph heavy computation consumption and their inability to work on corrupted or partial code, [97] suggested ADAMO, a transfer learning-based model. Specifically, the model's architecture consists of an encoder and a decoder linked by a Refiner, Additive White Gaussian Noise (AWGN), that reduces the incoordination between them. It also benefits from continuous pre-training and intermediate fine-tuning which are only compatible with sequential data showing that sequential data is still powerful and further research in that direction could bring significant improvement.

2.9 Summary

Besides the model architecture, many factors can influence a research like this, thus require further attention. Such factors can be; the quality of the dataset, dataset-splitting (method or project-wise), identifier splitting, duplication, programming languages used, the quality of the evaluation metrics but also human evaluation [98]. However, at the architecture level in which we are focusing, the aforementioned models were researched and their main characteristics, as well as drawbacks, are presented in Appendix A.

Chapter 3

Methods

In this chapter, the followed approach will be elaborated on. This includes a thorough explanation of the used models as well as a detailed description of their final implementation. The goal is to clarify the reasoning behind any decision that was made, provide details that would ensure the validity and ethical conduct of the work, and also inform the reader about the overview of the research method used in this thesis.

3.1 Research Process

This research follows the academic and scientific research standards. An extensive literature review initiated the process of the study of the subject, the related work and the recent discoveries. This allowed the highlight of unsolved issues that we tried to tackle with a new approach which was inspired by this review as well. Then based on our resources, expertise and judgment, an ensemble technique was chosen, consisting of the stacking of two models, GraphCodeBERT and PLBart. For their implementation, an appropriate dataset was selected and pre-processed, both models were adjusted to our needs and fine-tuned for the task of source code summarization and eventually combined for the testing of our hypothesis. Multiple aggregation techniques were tested on the stacking of the models and their contributions are eventually explained according to the results of the conducted experiments. All these stages are further described in this chapter.

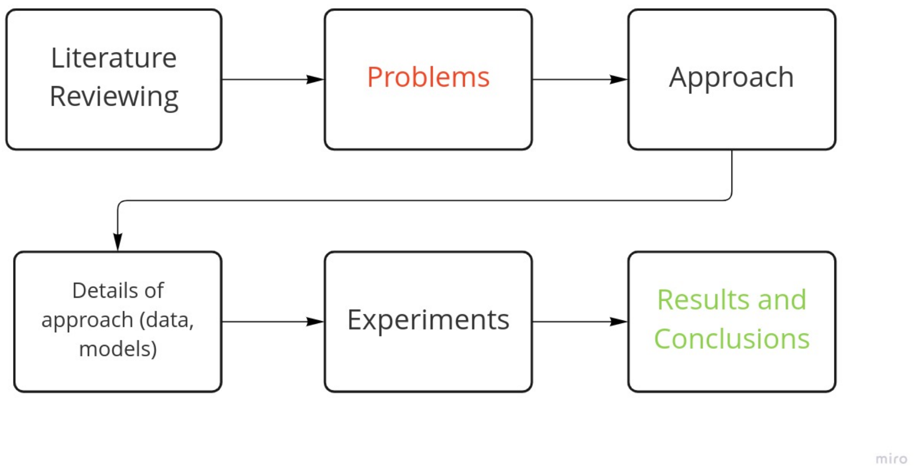


Figure 3.1: Research Process Lay-Out

3.2 Developed Model

The goal of this thesis is to examine whether combining two powerful models that tackle the same problem from different angles can be more beneficial than using them individually. To this end, an ensemble model was developed with the technique of stacking. The usage of ensembling learning was inspired by two factors. First was the interesting research of LeClair [17] who tried a similar approach but on an introductory and superficial level. Therefore, it was a nice opportunity to take that research one step further. The second factor comes out of the discovered problems of previous work. It was noticed that different models can tackle parts of these problems and hence it was rational to assume that maybe their combination could tackle all at once. Thus, the complementary base-models that were fine-tuned on the task of source code summarization are PLBart and GraphCodeBERT. Their results would be aggregated to produce the final outcome. Several aggregation techniques were tested to conclude which one fitted better. Before that though, the two models should be modified to produce results that are compatible for aggregation. For clarification of this statement, the initial fine-tuned PLBart and GraphCodeBERT were producing similar sentences but fundamentally different in the crucial phase of the probabilities production. An example illustrating the following issue is the following. PLBart would return as the most likely first word the token “Load” whereas the GraphCodeBERT would return the token “L”. Similar issues had to be dealt with and led to our final implementation. In this section, these details will be shared and explained for clarification purposes but also for highlighting potential problems and their solutions when a similar approach is followed.

3.2.1 Ensemble

The ensemble learning has been described in 2.7. Three main factors were mentioned there which are the models, the training data and the aggregation technique. Firstly, stacking was chosen because of the need for a large amount of data but also because of the objective of the thesis. Previous related works had focused on tackling specific issues from a specific point of view. However, ensembling models that are quite different and experts to their “domain” has shown promising results in other classification tasks [99]. Therefore, combining different experts developed by other researchers was selected as our strategy. From this observation, the training dataset and the models to be used were made clearer. We did not use the entire dataset to fine-tune the selected models, but this would be the right thing to do if resources and time are available.

One of the unanswered questions was; which models should be used? After a thorough examination of the available choices, PLBart and GraphCodeBERT were selected. PLBart could solve the first problem out of the three that were mentioned in 1.2 which was related to the quantity of data. PLBart is pre-trained on both programming and natural language which are not necessarily presented in pairs. Nowadays, there is a huge amount of code available online on websites such as Github [100], StackOverflow [101], Kaggle [102]. Text is available to an even higher degree for obvious reasons. This allows PLBart to be pre-trained on a huge corpus. At the same time, PLBart is one of the few models developed for bimodal tasks that would use an encoder-decoder architecture and are pre-training the decoder as well.

To the best of our knowledge, this is an innovative approach since so far in cases where transfer learning was involved [103], the pre-trained model would include an encoder which would produce a general-purpose code representation and then a decoder, or even more than one decoder-heads, would be added for the fine-tuning. Another strength of PLBart is the pre-training tasks. These are generative in this case and combined with the availability of data and the encoder-decoder architecture, the fine-tuning can be accelerated since the model already produces similar to the desired outputs.

The second selected model was GraphCodeBERT. The remaining in 1.2 problems refer to the structure of the code and the way that is implemented in a transformer-based model. The first element that GraphCodeBERT introduces is the exploitation of the structure of the code. Most of the models that are doing so, make use of ASTs. However, as mentioned before, the incorporation of ASTs into Transformers is problematic because of their extended size when are linearized. The utilization of Data Flows and Graph-Guided Masked Attention that are explained in 2.5.2 appear to be solving these issues.

In a task such as summarization, the aggregation is occurring on the output probabilities. Each model predicts the next token according to the probabilities that are assigned to each token within the vocabulary. From this, we can conclude that in order to have a meaningful aggregation, the vocabulary of all models should be the same and of the same order (e.g., all vocabularies have size 50005 and token number 134 corresponds the token "the"). Relative measures have been taken to lead to the final predictions as described and illustrated in Figures 2.6,2.7,2.8 and 2.9. In this work the mean, weighted mean, maximum and weighted maximum were tested and their results are illustrated and discussed in 4.4 and 5 respectively.

3.2.2 PLBart

As mentioned in 2.5.1 PLBART is inspired by Bart, but is adjusted to fit the PL peculiarities. Specifically, its seq2seq architecture is the same as BART-base, having six encoder and decoder layers and approximately 140 million parameters, with the addition of an extra normalization layer on top of both encoders and decoders. This addition was introduced by [104] and it is claimed by them that it stabilizes training with half-precision floating point format (FP16) precision.

Bart

Bart, Bidirectional and Auto-Regressive Transformers, is the origin of PLBart. Its architecture is the standard sequence-to-sequence Transformers from [2] with the difference that the Gaussian Error Linear Units (GeLU) activation functions [105] are preferred over Rectified Linear Units (ReLU). The encoder is bidirectional like BERT [73] and the decoder is auto-regressive as described in 2.4.2. Its details are shared with PLBart and hence are elaborated once.

Pre-training

PLBart is pre-trained on denoising auto-encoding. This means that the model is trained to fix a text that has been modified with the addition of some noise. This noise is added artificially with the usage of different noising functions. In the case of PLBart, the used noising methods are token masking, deletion and infilling. In token masking, tokens are randomly sampled and replaced by a mask token. Similarly, in token deletion, instead of token replacement the token is deleted. Finally, token infilling is like token replacement but instead of tokens, text spans are sampled and eventually replaced by a mask token. This type of pre-training allows simultaneous training on both code and natural language. Their functionality is illustrated in Table 3.1 for easier understanding. This way, the diversity and availability of data increases. Diversity-wise, the possibility of training on both code and natural language because of their shared alphabet and hence tokens, makes the model more versatile and capable of discovering relationships between different languages. Availability-wise, since data can be unlabelled, of any format, without the need of having pairs of code and its description, the available sources become abundant.

Original VS Noised data		
Original	Noisy	Technique
Load the thorium runtime modules	Load [MASK] thorium runtime [MASK]	**Token masking**
`def getMax (x , y , z) : return max (x, y, z)`	`def getMax (x , y , z) : max (x, , z)`	**Token deletion**
Return the maximum value given x, y, z	Return the maximum value given [MASK]	**Token infilling**

Table 3.1: Examples showing the noising effect of the three used noising methods. In token replacement, tokens [the] and [modules] were replaced by the [MASK] token. In token deletion, tokens [return] and the second [y] were deleted. In token infilling the span [x,y,z] was replaced by the [MASK] token.

Fine-tuning

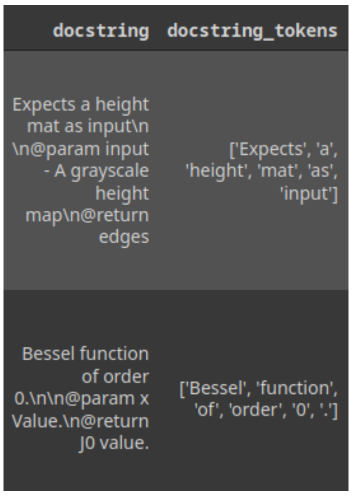
After pre-training, PLBart can be used for several code, code-to-natural language, or natural language-to-code downstream tasks such as code generation, code translation, code classification and of course code summarization. To do so, the creators of PLBart used the fairseq toolkit [106] on top of the pre-trained model, which allows its users to train their models for translation, summarization, language modeling and other text generation tasks. For the fine-tuning, inference and experimentation with PLBart, several pre-trained models, tokenizers and configurations are available at the huggingface library [107] and are compatible with Pytorch [108], TensorFlow [109] and JAX [110].

PLBart Implementation

In this section, the details on the implementation of PLBart and its fine-tuning are elaborated. PLBart has an encoder-decoder architecture. This basic architecture is implemented by the `huggingface` library under the `PLBartModel` class. However, it produces the raw hidden-states without any task-specific head on top. To this end, a linear head is put on top and takes over the language modeling role. This can be added manually or alternatively `PLBartForConditionalGeneration` can be used. An overview of this is illustrated in Figure 3.3.

Starting from the input data, the code and text are treated the same because of the lexical similarities that they share. `PLBartTokenizer` allows this, since it is pre-trained on both types' corpora and hence contains a large collection of tokens from both sides. This tokenizer is based on SentencePiece whose details are described in 2.3. After data exploration, it was discovered that many python code snippets were containing at least one comment. Most of the time, this comment would refer to the functionality of the at-the-time method. To this end, comments of the format ”’`comment`”’ and ”””`comment`””” were deleted before the training. In contrast, java snippets were clearer than the python ones and hence were used directly, but the docstrings were containing noise as can be seen in Figure 3.2. Therefore, similar actions were taken for the java docstrings. Specifically, the docstring token column was used to bring the collection of the important to the code description parts into a sentence-like sequence. This highlights the significance of data exploration and engineering before any training.

For the training of the model, the Trainer API from `huggingface` is



docstring	docstring_tokens
Expects a height mat as input\n \n@param input - A grayscale height map\n@return edges	['Expects', 'a', 'height', 'mat', 'as', 'input']
Bessel function of order 0.\n\n@param x Value.\n@return J0 value.	['Bessel', 'function', 'of', 'order', '0', '.']

Figure 3.2: Java snippets' "docstring" and "docstring tokens" columns that are used as descriptors of the code.

used. This requires the conversion of any type of data to `huggingface`'s Dataset. Once this is done, the tokenized docstring's tensor is renamed into "labels" and together with the "input_ids" and "attention_mask" of the given code constitute the input of the model.

After the training of the model, someone can proceed with the inference. For this, the "input_ids" and "attention_mask" of the code to be summarized need to be fed to the trained model. The possible outputs are the summary or the logits of this input. In our case, the logits are more useful since we want to manually apply a softmax function on them in order to obtain the next token probabilities that are supposed to be aggregated with the respective probabilities of GraphCodeBERT.

3.2.3 GraphCodeBERT

GraphCodeBERT's input and architecture can be indicated in its name. The first part of the name indicates the model's input. The graph refers to the AST, which is a tree and trees are specific types of graphs. Then, its ending, BERT, reveals information about its architecture. GraphCodeBERT is based on BERT [19] whose architecture is the standard Transformer that was initially introduced in [2]. These, together with the structure-specific pre-training tasks that were mentioned in 2.5.2 and the final implementation for code summarization are the subjects of interest in this section.

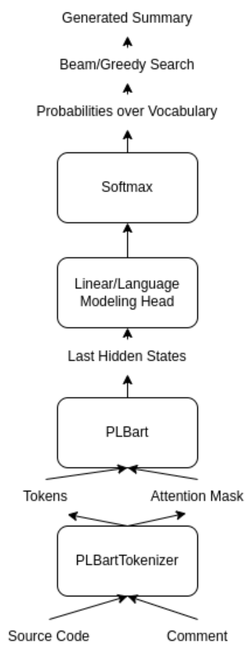


Figure 3.3: PLBart's depiction in detail. Comment is not considered during inference, since it is the goal.

Model's input

The input here, instead of being just code or natural language, is pairs of code and their comments in order to further support bi-modal tasks such as code summarization. Initially, given the source code $C = \{c_1, c_2, ..., c_n\}$, the dataflow $G(C) = (V, E)$ is constructed as described in 2.2. The final sequence input then is $X = \{[CLS], W, [SEP], C, [SEP], V\}$ where $W = \{w_1, w_2, ..., w_c\}$ corresponds to the respective comment and [CLS], [SEP] refer to the start of sequence and the data type separator respectively. Once this is done, the input X is converted to vectors H^0 which are the result of the summation of the tokens' embeddings and their positional embedding. In occasions where positional information is obtained from a dataflow, we refer to these positional embeddings as special. This procedure is depicted in Figure 3.4. A multi-layer transformer is followed, consisting of multi-head attention layers 2.4.1 and feed-forward networks as described in 2.4.2. However, incorporating the data flow in this procedure requires specialized attention to filter out irrelevant signals. This is why Graph-Guided Masked

Attention is introduced. The desired outcome is to consider relationships between nodes that have a direct edge linking one to the other. To this end, a mask matrix is constructed following Equation 3.1. A simple example illustrating the functionality of this matrix can be seen in Figure 3.5.

$$M_ij = \begin{cases} 0 & check = TRUE \\ -\infty & otherwise \end{cases} \tag{3.1}$$

where,
$check = \{q_i \in \{[CLS], [SEP]\}$ *or* $q_i, k_j \in W \cup C$ *or* $(q_i, k_j) \in E \cup E'\}$,
$q_i : query, k_j : key$,
$E' = \{(v_i, c_j)/(c_j, v_i) : v_i$ of data flow is identified by source code token $c_j\}$.

The new attention is calculated as in 2.4.1 by modifying Equation 2.1 like illustrated in Equation 3.2

$$Softmax(\frac{Q \cdot K^T}{\sqrt{d_k}} + M)V \tag{3.2}$$

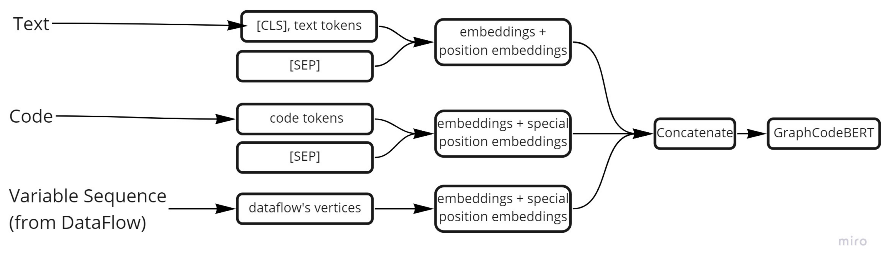
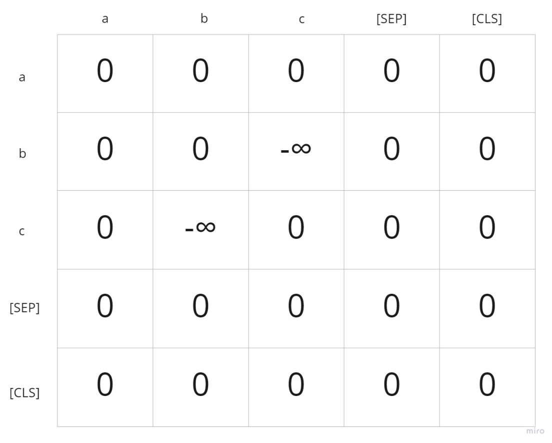

Text
[CLS], text tokens
[SEP]
embeddings + position embeddings
Code
code tokens
[SEP]
embeddings + special position embeddings
Variable Sequence (from DataFlow)
dataflow's vertices
embeddings + special position embeddings
Concatenate
GraphCodeBERT
miro

Figure 3.4: GraphCodeBERT's input after the data flow is obtained from the code snippet's AST.

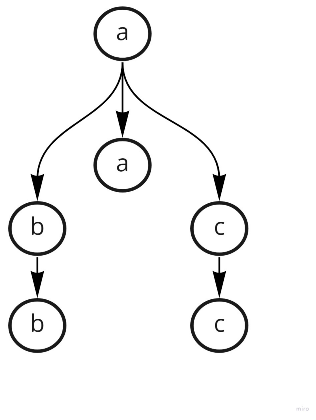
Pre-training

GraphCodeBERT is pre-trained on three tasks: MLM, Edge Prediction (EP) and Node Alignment (NA). MLM is taken from [19]. This task is widely used for pre-training transformer-based models such as [69, 70, 72]. 15% of the training data's tokens are sampled and replaced by a [MASK] token. However, this is not happening every time but 80% while for 10% the replacement is



	a	b	c	[SEP]	[CLS]
a	0	0	0	0	0
b	0	0	-∞	0	0
c	0	-∞	0	0	0
[SEP]	0	0	0	0	0
[CLS]	0	0	0	0	0

(a) Mask Matrix



(b) Dataflow

Figure 3.5: Graph-Guided Masked Attention's mask matrix (Left) on an example data flow (Right).

a random token and for the last 10% there is no action taken. The model then learns to predict these tokens and this way it completes its first stage of pre-training. The other two tasks are more structure-aware and oriented since they focus mostly on the produced data flow graph and its elements. Similarly to the MLM task, in Edge Prediction, 20% of the nodes in V of data flow $G(C) = (V, E)$ are sampled and then their direct edges are masked by changing their value in the mask matrix to $-\infty$. Once again, the model is trained to predict those masked edges. Finally, in Node Alignment the objective is to predict edges/alignment between the code tokens and the data flow nodes. This is for the model to be able to align source code tokens to variables of the data flow and therefore have both "inputs" related instead of being totally independent. More patterns and information can be extracted by doing so, contributing this way to more robust representations of code snippets. The implementation of this training is almost identical to the Edge Prediction with the only difference being the masking of edges between the code token and the sampled nodes of the data flow, hence edges belonging to E' instead of E (see Equation 3.1).

GraphCodeBERT Implementation

Once again, specific actions need to be taken for GraphCodeBERT, besides the standard implementation. It is worth mentioning that GraphCodeBERT had not been fine-tuned yet for the task of source code summarization, hence we could not have reference results or expectations. For consistency purposes, similar steps are used in both PLBart and GraphCodeBERT.

GraphCodeBERT adds the structure element to our ensemble model. The differentiation factor from other models that also do that, is that GraphCodeBERT does not directly utilize ASTs but converts them into data flows. This way, more semantic information can be extracted by exploiting the relative position of code tokens. Moreover, data flows are shorter than ASTs, something that makes them more suitable for a Transformer.

We start by tokenizing the input code and its natural language comment. As a tokenizer, we are using the `PLBartTokenizer` which is pre-trained on PLBart-base model. PLBart is pre-trained on a huge corpus of both programming and natural language and since the idea was to have a common tokenizer, this one prevailed. In parallel, the source code is parsed into AST with the help of `tree_sitter` [111]. Once the AST is obtained, the Data Flow is constructed as instructed in 2.2. Then, the `tokens, masks and variable_sequences/position_indexes`, acquired from the tokenizer and the data flow parser respectively, are ready to be fed into the model. A similar procedure occurs for the natural language part consisting only of the tokenization part.

GraphCodeBERT is a Bi-directional Encoder that produces transformer-based representations. We make use of the `hugging_face`'s pre-trained model by changing though its configuration, and specifically the vocabulary size. The new size is 50005, and is selected for being in alignment with PLBart's vocabulary. Since GraphCodeBERT is just an encoder, a transformers' decoder is put on top of it. This decoder stack consists of six layers. Then the language modeling head follows which includes two linear layers, linking the Transformer to the softmax layer that is responsible for the production of the next token probabilities. This is finally done by a Beam search 2.6 in the training phase, which is turned into a greedy search in the inference by changing the beam size into one. As far as the loss function is concerned, cross-entropy is used once again because of the nature of the predictions which are probabilistic. An illustration of this procedure can be seen in Figure 3.6.

For getting the desired summaries from this model, the natural language input is set to "None". Besides that, the rest is input to the trained model which produces the summary. For this project's scope, we also return the LogSoftmaxed output. Then $e^{LogSoftmaxed}$ is applied to get the Softmax layer which is going to be used in the aggregation procedure.

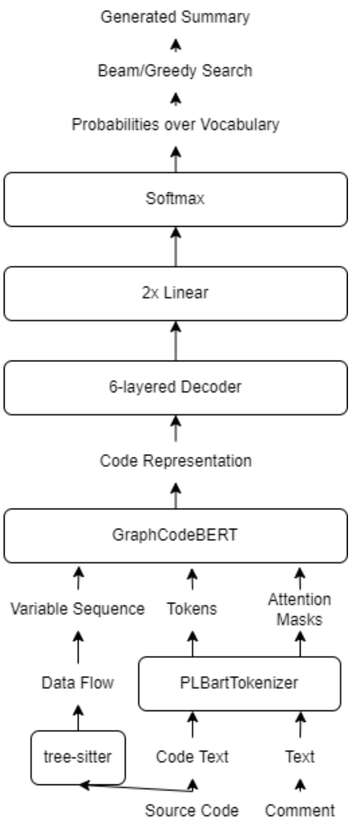


Figure 3.6: GraphCodeBERT's depiction in detail. Comment is not considered during inference, since it is the goal.

Chapter 4

Evaluation and Results

In this chapter, the results of the followed approach are presented. Before that though, practical details concerning the implementation precede. Such details are the development environment, the used dataset and the evaluation metrics. This information aims the full transparency of the work that would add reliability and validity to it but would also allow the reproducibility of the results.

4.1 Test environment

The reproduction of everything that was implemented in this thesis is easy and does not require any environment configuration. To run the given code in a Google Colab notebook, it is only required to adjust the paths to the dataset to the respective one of the at the time user. Concerning the versions of libraries and toolkits that were used, details follow:

- **Python:** 3.6
- **Pytorch:** 1.11.0
- **Cuda:** 11.3
- **Transformers:** 4.18.0
- **hugging_face_hub:** 0.5.1
- **tokenizers:** 0.12.1
- **sentencepiece:** 0.1.96
- **tree_sitter:** 0.20.0

Used Hardware

For the training and testing of the models, the utilization of Graphics Processing Units (GPUs) was necessary. To this end but also because of personal hardware limitations, Google Colab was used, specifically the PRO-version. This version was selected because of the faster GPUs, the additional memory and the longer runtimes. Concerning the GPUs, there is no granted type, but it varies from KP80 to T4 or P100, where T4 and P100 are only available in the PRO-version or PRO+ and their sizes are up to 16 GB.

4.2 Dataset

The dataset that was used throughout this Master thesis is provided by [112] and is known as the CodeSearchNet corpus. This dataset is shared by its authors and is widely used for bimodal (Programming Language, Natural Language) tasks such as Code Summarization, Code Search and Code Generation. It was selected because of its size, diversity and wide usage. Concerning the size, the initial dataset is approximately 20 GB and 3.5 GB when it is compressed. It consists of roughly two million rows divided already into train, validation and test partitions that look like Table 4.1. Diversity-wise, there are six programming languages represented, including Python, Java, JavaScript, PHP, Go and Ruby. Their representation is not balanced (see Table 4.2), but each language is trained and tested separately, hence the relative conclusions can be isolated to the at the time language. Finally, the fact that is used to such a degree, with some examples being [69, 14, 15, 25, 16, 82, 78], shows that it is a dataset compatible for experimentations such ours but also excludes the dataset-factor in a potential comparison with one of the aforementioned models.

Partition		
Train	Validation	Test
1880853	100529	89154

Table 4.1: Partition of the dataset to train, validation and test sub-sets.

4.2.1 Data Schema

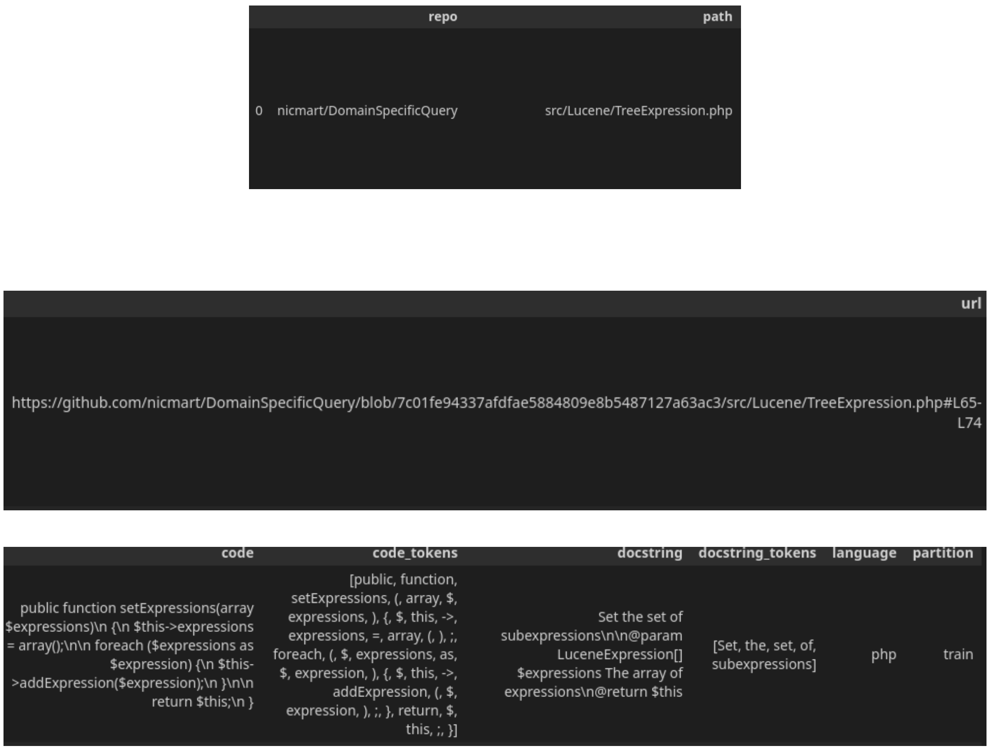
One of the strengths of PLBart is that it can be pre-trained on unlabelled code or natural language data. Thanks to the vast availability of programming

Language Distribution					
Python	Java	JavaScript	PHP	GO	Ruby
457461	496688	138625	578118	346365	53279

Table 4.2: Language representation in CodeSearchNet corpus.

and natural language datasets, the acquisition of suitable and sufficient data is not an issue. However, for the fine-tuning on the task of summarization, the situation is different, since pairs of code and text are necessary. Therefore, this dataset is suitable because of its size and schema. Specifically, it consists of the columns: repo, path, url, code, code tokens, docstring, docstring tokens, language,and partition. Their functionality is described right afterward and an example row can be seen in Figure 4.1.

- **repo:** repository of origin
- **path:** the path to the original file
- **url:** Uniform Resource Locator (URL) that leads to the snippet
- **code:** the code in a string format
- **code tokens:** tokenized code (differ from the tokens that are finally used)
- **docstring:** the docstring/comment of the code if exists
- **docstring tokens:** tokenized docstring (differ from the tokens that are finally used)
- **language:** the programming language
- **partition:** the partition where this row belongs



	repo	path
0	nicmart/DomainSpecificQuery	src/Lucene/TreeExpression.php

url
https://github.com/nicmart/DomainSpecificQuery/blob/7c01fe94337afdfae5884809e8b5487127a63ac3/src/Lucene/TreeExpression.php#L65-L74

code	code_tokens	docstring	docstring_tokens	language	partition
public function setExpressions(array $expressions)\n {\n $this->expressions = array();\n\n foreach ($expressions as $expression) {\n $this->addExpression($expression);\n }\n\n return $this;\n }	[public, function, setExpressions, (, array, $, expressions,), {, $, this, ->, expressions, =, array, (,), ;, foreach, (, $, expressions, as, $, expression,), {, $, this, ->, addExpression, (, $, expression,), ;, }, return, $, this, ;, }]	Set the set of subexpressions\n\n@param LuceneExpression[] $expressions The array of expressions\n@return $this	[Set, the, set, of, subexpressions]	php	train

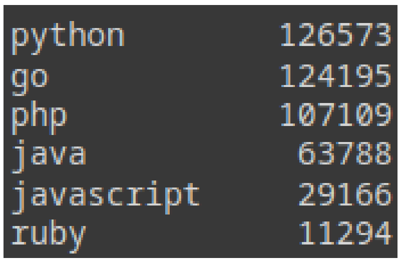
Figure 4.1: Random row from CodeSearchNet corpus

4.2.2 Data Pre-processing

Before the data was used, some pre-processing procedures preceded. These modifications to the initial dataset were aimed at cleaning and preparing it for the training. The taken strategies were followed in the past by most of the users of this dataset for this task. In this project, we start with the changes that [69] suggested, adjusted slightly to our needs. Initially, comments were removed from the code in order for it to be clean and for us to take advantage of its syntax and semantics without being influenced by any noise. In some cases, the code snippets were impossible to parse into ASTs, hence were excluded. Moreover, rows with non-English "docstring" were not included as well, since the objective of the model would be the summarization of source code into English. Finally, some of the "docstrings" were either too short or too long. By too short we mean that they were consisting of one or two words and most of the time these were in alignment with the name of the method. On the other hand, when the "docstring" was too long (longer than 100 words), it was too noisy and it either did not include useful information about the code or this information was lost among useless to our goal text. This is why examples with code-comment pairs where the comment was longer than three words and shorter than one hundred were only considered. The final dataset is explored in 4.2.3 for a better understanding of it.

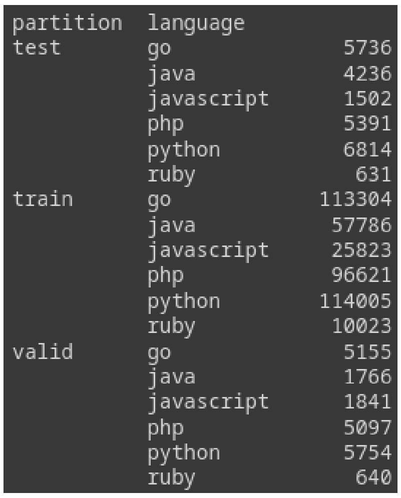
4.2.3 Data Exploration

The final dataset consists of 462125 rows. It is divided into train, validation and test subsets for the avoidance of data leakage during the training of the models. The training partition is the biggest one with 417562 rows while the validation and testing partitions' rows amount to 24310 and 20253 respectively. The representation of the included programming languages and their individual partitions can be seen in Figures 4.2. Then the average words per code snippet and "docstring" are calculated for statistical purposes but also for the selection of appropriate input and output size. From the code snippet's side, the average is 47.53 per snippet and from the natural language's side the average is as anticipated smaller, at 9.21 words per comment. Finally, the top ten most frequent words in the "code" and "docstring" columns can be seen in Figure 4.3. The most frequent tokens in the code are equality/inequality symbols, brackets and commands such as "return", "for" that are used in the majority of the methods and are common in most programming languages. The most frequent natural language words include some stop-words and others such as "param" and "return" that have a strong code-descriptive context.



```
python        126573
go            124195
php           107109
java           63788
javascript     29166
ruby           11294
```

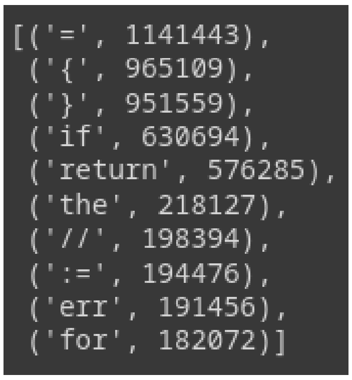
(a) Language Distribution



```
partition  language
test       go              5736
           java            4236
           javascript      1502
           php             5391
           python          6814
           ruby             631
train      go            113304
           java           57786
           javascript     25823
           php            96621
           python        114005
           ruby           10023
valid      go              5155
           java            1766
           javascript      1841
           php             5097
           python          5754
           ruby             640
```

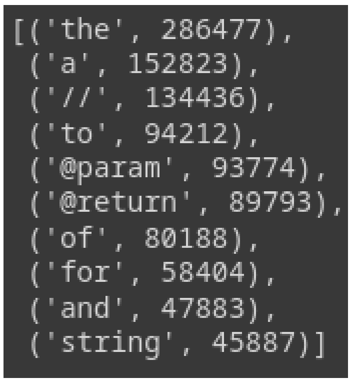
(b) Partition per Language

Figure 4.2: Dataset partitions on full dataset and per programming language.



```
[('=', 1141443),
 ('{', 965109),
 ('}', 951559),
 ('if', 630694),
 ('return', 576285),
 ('the', 218127),
 ('//', 198394),
 (':=', 194476),
 ('err', 191456),
 ('for', 182072)]
```

(a) Top10 most frequent "code" tokens



```
[('the', 286477),
 ('a', 152823),
 ('//', 134436),
 ('to', 94212),
 ('@param', 93774),
 ('@return', 89793),
 ('of', 80188),
 ('for', 58404),
 ('and', 47883),
 ('string', 45887)]
```

(b) Top10 most frequent "docstring" tokens

Figure 4.3: Most frequent tokens, not from the used Tokenizer.

4.3 Evaluation framework

There are many evaluation metrics used in this field of Machine Translation, starting from standard precision, recall and F1 score over the target sequence and going to more text generation-specific metrics such as BLEU(-N), CIDEr, METEOR, RIBES, ROUGE but also human evaluation.

Generic evaluation metrics, such as precision, recall, and F1 score, are not ideal but were used by [113, 114] and then others like [57] followed in order to be able to make comparisons between their implementation and previous ones. Another widely used metric is BLEU and its variants [115]. The final score is defined from the overlapping n-grams between the generated and the reference sentence. Moving on, CIDEr [116] was developed for image description evaluation. It uses TF-IDF values [117] of each n-gram and then utilizes the average cosine similarity between the candidate and reference sentence for accounting for both precision and recall. Next is METEOR [118] which uses word-matching modules between two strings to create a word alignment that would eventually produce the final score based on the unigram precision and recall. Then RIBES [119] is a rank-based evaluation metric for machine translation that takes into consideration the global word order which could be essential in some specific language-to-language translations. The final automatic metric is ROUGE-L [120] which is a summary-specific metric that is based on overlapping tokens, sequences, or token pairs. Finally, since the automatic metrics are not always representative of the actual quality [121, 122] a lot of researchers are performing a human evaluation of the results in addition to the automatic metrics. In the project, the BLEU score was used and this is why this is the only one further elaborated. The selection of the BLEU score among the others was mostly because of its wide usage by other researchers but also its simplicity and interpretability.

Bleu Score

The BLEU score ranges from 0 to 100 and corresponds to the percentage of similarity between the generated and the reference sentences. There are many variants of BLEU, all named BLEU-n where n differs each time and refers to the n-gram precision that is used at the time. This n-gram precision is defined in Equation 4.1 and the final BLEU score in Equation 4.2 where ω_n is the uniform weight 1/N and BP is the Brevity Penalty. The Brevity Penalty is used to avoid high scores when a sentence is short or contains a highly repeated n-gram. However, the BLEU score is made for corpus level evaluations, hence needs some modifications to be adapted to the sentence level.

These modifications are more and more necessary when the n (of n-gram) increases. The reason behind this is that it is harder to exactly match large n-grams in a “small” sentence compared to a “large” corpus. Since the BLEU score considers all n-gram precisions by using the geometric mean, all of them should be positive in order not to cancel the rest. The solution is to apply a smoothing technique. Such techniques are either replacing zeros with a constant or function generated small positive value or are using the average of the n-gram match counts from consecutive n-gram lengths (e.g., n-1, n, n+1) [123].

As can be seen, there are several different implementations of the BLEU score. The important outcome of this observation is that it should be avoided by researchers to use others’ BLEU scores for comparison of results since there is no absolute BLEU implementation. Consequently, when a model comparison is occurring, all models should be evaluated with the same BLEU score calculation function.

$$p_N = \frac{\sum_{C \in Candidates} \sum_{n-gram \in C} count_{clip}(n-gram)}{\sum_{C' \in Candidates} \sum_{n-gram \in C'} count_{clip}(n-gram')}, \tag{4.1}$$

where C corresponds to sentence-level and C’ to corpus-level.

$$BLEU = BP \cdot exp \sum_{n=1}^{N} \omega_n \log p_n \tag{4.2}$$

Our implementation of the BLEU-score is a BLEU-4 and is inspired by [124]. We use the same script for calculating the BLEU-4 score of all the developed models, including the ensemble and also the individual ones. Initially, both the reference and predicted sequences are normalized and brought into an appropriate format for the computation of the final score. These modifications include the striping of hyphenation, the joining of lines, making sure that the input sentences are all of string type and the lower-casing of all the strings. Afterward, the reference and predicted sequences are additionally edited to get all the information needed out of them for the BLEU-4 score. Then the references and predictions are mapped and fed to the smoothed-BLEU-4 score calculator, since it is used for sentence-level texts.

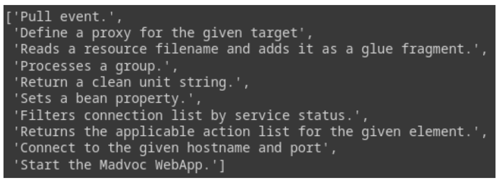
4.4 Major Results

In this section, some representatives of our work results are shared. The results include samples of the outputs that could be used for clarification and human evaluation. They also include the BLEU-4 scores of each developed model. In total, the presented models are PLBart and GraphCodeBERT that are used as baselines, and additional ensemble models that arise from different aggregation techniques on these base models.

4.4.1 PLBart Results

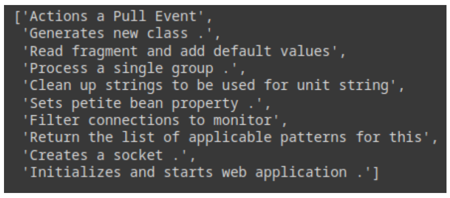
PLBart was fine-tuned on a subset of the previously presented dataset. As mentioned in 4.2.3 the dataset was partitioned into training, validation and testing. Within these partitions, the dataset was further divided according to the respective programming language of the at the time code snippet. The selected programming languages in this work were Python and Java. The size of the Python-training partition was 1207 examples, validation 68 and it was tested on 100 instances. Java's training dataset was consisting of 5779 rows, while validation was limited to 177 and the testing dataset was once again of size 100. These sizes are smaller than the available data size that was mentioned in 4.2.3. A sample was preferred for the training of the model to make it possible with our available resources but also to fit the master thesis' scheduled time slot.

Concerning the fine-tuning on the Python snippets, PLBart was trained for 20 epochs with a batch size of 8, 500 warm-up steps and a weight decay of 0.01. In this implementation, weight decay works as an L2 Regularizer and helps the avoidance of getting an overfitting model. The last cross-entropy, evaluation loss, was 0.0002 and the model was able to generalize and produce summaries that reached the BLEU-score of 18.09. However, GraphCodeBERT was not able to produce useful summaries on Python snippets after the provided, extensive for our resources, training, therefore PLBart itself could not add any value to this research and hence no further analysis was needed. Moving to the Java snippets that were eventually used for the testing of the objectives of this master thesis, PLBart was trained for 12 epochs with the rest of the hyper-parameters remaining the same as on the Python snippets. The final training loss was 0.019 and the respective BLEU-4 score on the testing dataset was 15.49. A sample of PLBart's output, accompanied by the ground truth of the input snippets can be seen in Figure 4.4.



```
['Pull event.',
 'Define a proxy for the given target',
 'Reads a resource filename and adds it as a glue fragment.',
 'Processes a group.',
 'Return a clean unit string.',
 'Sets a bean property.',
 'Filters connection list by service status.',
 'Returns the applicable action list for the given element.',
 'Connect to the given hostname and port',
 'Start the Madvoc WebApp.']
```

(a) PLBart summaries



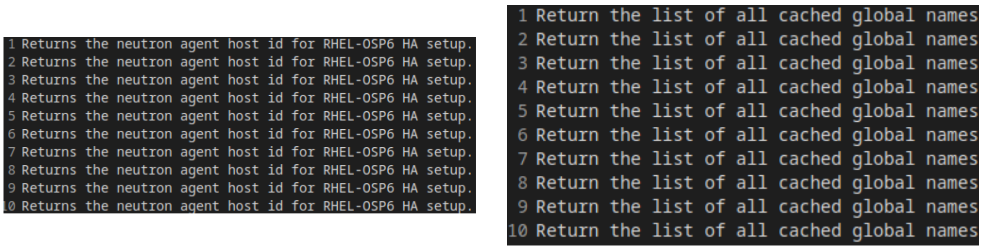
```
['Actions a Pull Event',
 'Generates new class .',
 'Read fragment and add default values',
 'Process a single group .',
 'Clean up strings to be used for unit string',
 'Sets petite bean property .',
 'Filter connections to monitor',
 'Return the list of applicable patterns for this',
 'Creates a socket .',
 'Initializes and starts web application .']
```

(b) Ground truth summaries

Figure 4.4: First 10 of the testing dataset Java code summaries. At the left is the produced summary by PLBart and at the right is the respective ground truth.

4.4.2 GraphCodeBERT Results

As mentioned in 3.2.3, GraphCodeBERT's configuration changed, to get the same vocabulary as in PLBart. This made necessary the usage of the `from_config` method of hugging face instead of the `from_pretrained`. This allows the modification of the GraphCodeBERT's output and the usage of its architecture but blocks the transfer of the weights which were updated after the pre-training on the mentioned tasks. Therefore, GraphCodeBERT was practically trained from scratch and hence required more steps to reach the desired performance level. GraphCodeBERT was initially trained on the Python partition. After more than 100 epochs on either the full Python partition or a small sample of it, the model was struggling to learn. We assume that the reason behind it is the structure of Python snippets that may not add the desired value to the model which was producing the same output for every input as can be seen in Figure 4.5. Java's outputs were more promising, thus Java snippets were the ones that were eventually used for both models and the ensemble models that they were forming.

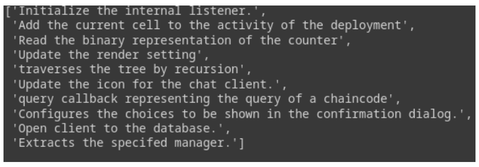


```
Returns the neutron agent host id for RHEL-OSP6 HA setup.
Returns the neutron agent host id for RHEL-OSP6 HA setup.
Returns the neutron agent host id for RHEL-OSP6 HA setup.
Returns the neutron agent host id for RHEL-OSP6 HA setup.
Returns the neutron agent host id for RHEL-OSP6 HA setup.
Returns the neutron agent host id for RHEL-OSP6 HA setup.
Returns the neutron agent host id for RHEL-OSP6 HA setup.
Returns the neutron agent host id for RHEL-OSP6 HA setup.
Returns the neutron agent host id for RHEL-OSP6 HA setup.
Returns the neutron agent host id for RHEL-OSP6 HA setup.
```

```
Return the list of all cached global names
Return the list of all cached global names
Return the list of all cached global names
Return the list of all cached global names
Return the list of all cached global names
Return the list of all cached global names
Return the list of all cached global names
Return the list of all cached global names
Return the list of all cached global names
Return the list of all cached global names
```

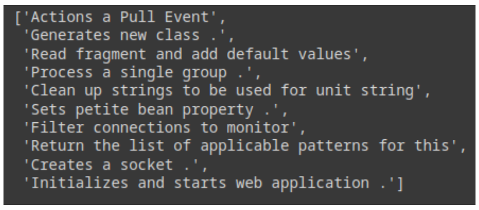
Figure 4.5: GraphCodeBERT's outputs on Python inputs.

The model was initially trained on its default, by its creators, hyper-parameters. This resulted in an over-fitting model on our relatively small dataset. As can be seen in Figure 4.6, the model was able to produce almost the exact summary out of the training code snippets whereas the results were not so precise concerning the testing data. The BLEU-score at this case was around 8 to 9.



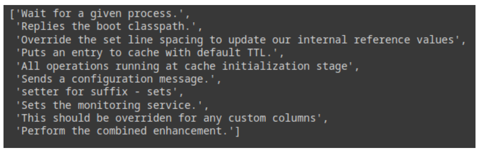
```
['Initialize the internal listener.',
 'Add the current cell to the activity of the deployment',
 'Read the binary representation of the counter',
 'Update the render setting',
 'traverses the tree by recursion',
 'Update the icon for the chat client.',
 'query callback representing the query of a chaincode',
 'Configures the choices to be shown in the confirmation dialog.',
 'Open client to the database.',
 'Extracts the specifed manager.']
```

(a) Produced summaries on testing dataset



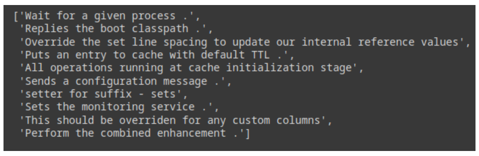
```
['Actions a Pull Event',
 'Generates new class .',
 'Read fragment and add default values',
 'Process a single group .',
 'Clean up strings to be used for unit string',
 'Sets petite bean property .',
 'Filter connections to monitor',
 'Return the list of applicable patterns for this',
 'Creates a socket .',
 'Initializes and starts web application .']
```

(b) Reference summaries from testing dataset



```
['Wait for a given process.',
 'Replies the boot classpath.',
 'Override the set line spacing to update our internal reference values',
 'Puts an entry to cache with default TTL.',
 'All operations running at cache initialization stage',
 'Sends a configuration message.',
 'setter for suffix - sets',
 'Sets the monitoring service.',
 'This should be overriden for any custom columns',
 'Perform the combined enhancement.']
```

(c) Produced summaries on training dataset

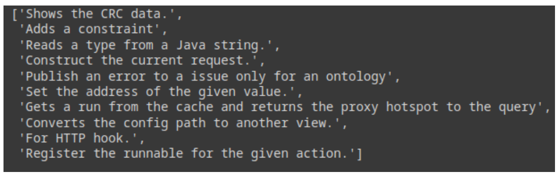


```
['Wait for a given process .',
 'Replies the boot classpath .',
 'Override the set line spacing to update our internal reference values',
 'Puts an entry to cache with default TTL .',
 'All operations running at cache initialization stage',
 'Sends a configuration message .',
 'setter for suffix - sets',
 'Sets the monitoring service .',
 'This should be overriden for any custom columns',
 'Perform the combined enhancement .']
```

(d) Reference summaries from training dataset

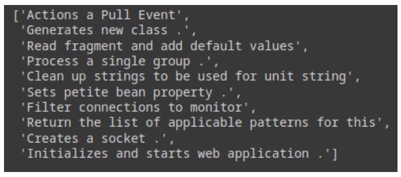
Figure 4.6: Results of GraphCodeBERT after training on default hyper-parameters.

To prevent over-fitting, the first step was to add more data. A new training dataset was taken of double the size of the one mentioned in 4.4.1, so of 11557 code snippets and their docstring. Moreover, hyper-parameter tuning followed, with the introduction of dropout and L2-regularization. The specific hyper-parameters that were used for the final model are 0.3 dropout at GraphCodeBERT (encoder), 0.5 dropout at the TransformerDecoder (decoder) and weight decay of 0.01 at the optimizer. Once again, weight decay works as an L2 regularizer. The new GraphCodeBERT was able to generalize better and hence was preferred over the previous one for the development of the ensemble model. Its best BLEU-4 score, after 100 epochs, was at 12.05, closing this way the gap from the 15.49 BLEU-4 score of PLBart. The training loss had fallen to 0.3. Its output summaries on the same input as PLBart 4.4 can be seen in Figure 4.7.



```
['Shows the CRC data.',
 'Adds a constraint',
 'Reads a type from a Java string.',
 'Construct the current request.',
 'Publish an error to a issue only for an ontology',
 'Set the address of the given value.',
 'Gets a run from the cache and returns the proxy hotspot to the query',
 'Converts the config path to another view.',
 'For HTTP hook.',
 'Register the runnable for the given action.']
```

(a) GraphCodeBERT summaries



```
['Actions a Pull Event',
 'Generates new class .',
 'Read fragment and add default values',
 'Process a single group .',
 'Clean up strings to be used for unit string',
 'Sets petite bean property .',
 'Filter connections to monitor',
 'Return the list of applicable patterns for this',
 'Creates a socket .',
 'Initializes and starts web application .']
```

(b) Ground truth summaries

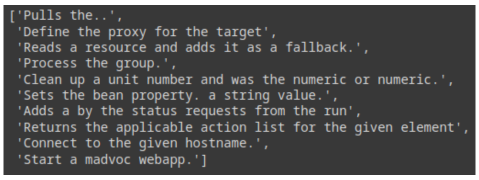
Figure 4.7: First 10 of the testing dataset Java code summaries. At the left is the produced summary by GraphCode and at the right is the respective ground truth.

4.4.3 Ensemble Models and their Results

For making the aggregation of the softmax tensors possible, the respective produced sentences should have the same size. To this end, the produced (sentence_length x vocabulary_size)-tensors were padded with zero-tensors of size equal to the vocabulary_size to obtain sentences of size 128 x vocabulary_size. After this, the different aggregation techniques followed. For all of the four aforementioned ensemble techniques, the appropriate to each occasion aggregation of two sentences was firstly computed and followed by an argmax-function that would identify the tokens belonging to the vocabulary with the highest probability to be the next token in the desired summary. The token-IDs obtained from this argmax-function were eventually decoded to form the final output. Below we present some interesting results from our experimentation with this procedure.

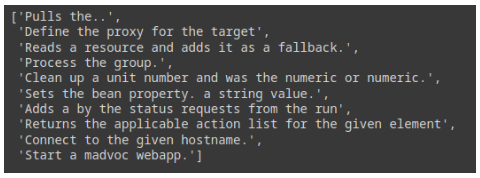
Mean/Max

The simplest methods that were initially tested were the mean and maximum aggregation techniques that are described in 2.7. Their results were almost identical, achieving the scores of 13.49 and 13.46 respectively. Their produced summaries can be seen in Figure 4.8. For conciseness purposes we limited the illustrated results to the first ten of the used testing dataset.



```
['Pulls the..',
 'Define the proxy for the target',
 'Reads a resource and adds it as a fallback.',
 'Process the group.',
 'Clean up a unit number and was the numeric or numeric.',
 'Sets the bean property. a string value.',
 'Adds a by the status requests from the run',
 'Returns the applicable action list for the given element',
 'Connect to the given hostname.',
 'Start a madvoc webapp.']
```

(a) Summaries produced from mean-aggregation



```
['Pulls the..',
 'Define the proxy for the target',
 'Reads a resource and adds it as a fallback.',
 'Process the group.',
 'Clean up a unit number and was the numeric or numeric.',
 'Sets the bean property. a string value.',
 'Adds a by the status requests from the run',
 'Returns the applicable action list for the given element',
 'Connect to the given hostname.',
 'Start a madvoc webapp.']
```

(b) Summaries produced from max-aggregation

Figure 4.8: First 10 of the testing dataset Java code summaries with the mean and max aggregation techniques. The reference outcomes are illustrated in Figure 4.9

Weighted Mean/Weighted Max

For the weighted mean and max, there were several weights tested, either leveraging PLBart or GraphCodeBERT's probabilities. These weights ranged from one to a hundred but here only the most interesting results are presented. The BLEU-scores of the following summaries varied from 12.35 to 15.52. Detailed results can be seen in Tables 4.3 and 4.4. Besides the PLBart and GraphCodeBERT's softmax tensors, some of the already aggregated tensors were also tested since during the experimentation it was noticed that they were presenting some interesting characteristics that could add value to this procedure. Specifically, it was tested whether a model that gives more weight to the seemingly weaker model, here GraphCodeBERT, and enhances its performance, can be combined again with the other model, to produce an even better result. The enhanced GraphCodeBERT model's probabilities, in this case, are obtained after using the weighted max technique on PLBart and GraphCodeBERT, with weights one to PLBart and eight to GraphCodeBERT, aiming this way the dominance of GraphCodeBERT but also the contribution of PLBart. The new tensor is then the first of the inputs in a weighted mean aggregation, while the regular PLBart remains the second input. For clarification purposes, we are going to be referring for the rest of this thesis to this model as DoubleEnsemble. Illustrations of these results can be seen in Figures 4.10,4.11,4.12 and 4.13.

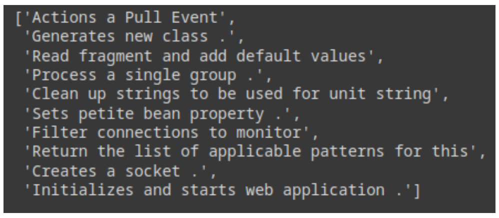
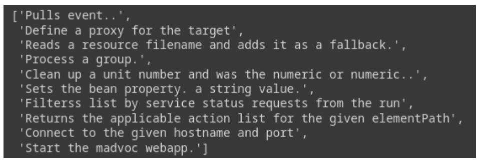
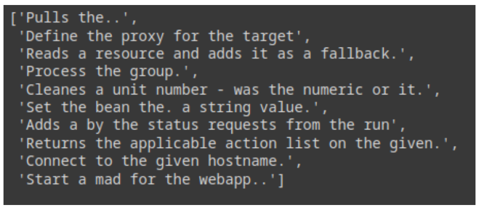

```
['Actions a Pull Event',
 'Generates new class .',
 'Read fragment and add default values',
 'Process a single group .',
 'Clean up strings to be used for unit string',
 'Sets petite bean property .',
 'Filter connections to monitor',
 'Return the list of applicable patterns for this',
 'Creates a socket .',
 'Initializes and starts web application .']
```

Figure 4.9: Reference outputs of the first ten code-snippets of our testing dataset. It is used as reference for all the tested approaches.

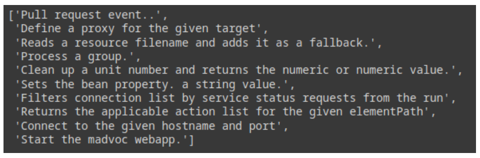

```
['Pulls event..',
 'Define a proxy for the target',
 'Reads a resource filename and adds it as a fallback.',
 'Process a group.',
 'Clean up a unit number and was the numeric or numeric..',
 'Sets the bean property. a string value.',
 'Filterss list by service status requests from the run',
 'Returns the applicable action list for the given elementPath',
 'Connect to the given hostname and port',
 'Start the madvoc webapp.']
```

(a) Weighted mean with weight of PLBart=2 and weight of GraphCode-BERT=1. BLEU-4 score: 14.65.

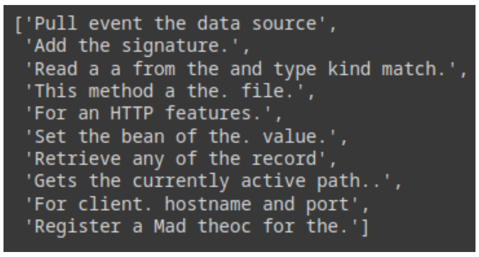

```
['Pulls the..',
 'Define the proxy for the target',
 'Reads a resource and adds it as a fallback.',
 'Process the group.',
 'Cleanes a unit number - was the numeric or it.',
 'Set the bean the. a string value.',
 'Adds a by the status requests from the run',
 'Returns the applicable action list on the given.',
 'Connect to the given hostname.',
 'Start a mad for the webapp..']
```

(b) Weighted mean with weight of PLBart=1 and weight of GraphCode-BERT=2. BLEU-4 score: 12.79.

Figure 4.10: Summaries produced with the aggregation technique of weighted mean.

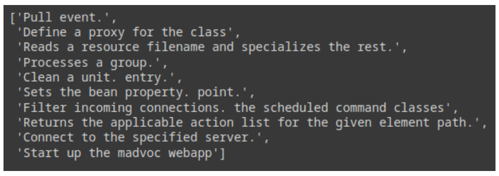

```
['Pull request event..',
 'Define a proxy for the given target',
 'Reads a resource filename and adds it as a fallback.',
 'Process a group.',
 'Clean up a unit number and returns the numeric or numeric value.',
 'Sets the bean property. a string value.',
 'Filters connection list by service status requests from the run',
 'Returns the applicable action list for the given elementPath',
 'Connect to the given hostname and port',
 'Start the madvoc webapp.']
```

(a) Weighted max with weight of PLBart=10 and weight of GraphCodeBERT=1. BLEU-4 score: 14.52.


```
['Pull event the data source',
 'Add the signature.',
 'Read a a from the and type kind match.',
 'This method a the. file.',
 'For an HTTP features.',
 'Set the bean of the. value.',
 'Retrieve any of the record',
 'Gets the currently active path..',
 'For client. hostname and port',
 'Register a Mad theoc for the.']
```

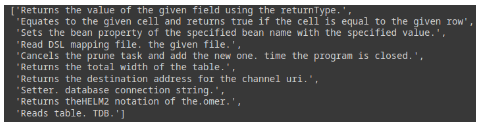
(b) Weighted max with weight of PLBart=1 and weight of GraphCodeBERT=8. BLEU-4 score: 12.35.

Figure 4.11: Summaries produced with the aggregation technique of weighted max.



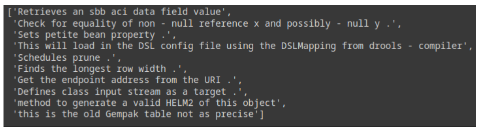
```
['Pull event.',
 'Define a proxy for the class',
 'Reads a resource filename and specializes the rest.',
 'Processes a group.',
 'Clean a unit. entry.',
 'Sets the bean property. point.',
 'Filter incoming connections. the scheduled command classes',
 'Returns the applicable action list for the given element path.',
 'Connect to the specified server.',
 'Start up the madvoc webapp']
```

Figure 4.12: DoubleEnsemble 4.4.3(Mean/Max) results with weights 1 to the enhanced GraphCodeBERT and 10 to PLBart. BLEU-4 score: 15.52



```
['Returns the value of the given field using the returnType.',
 'Equates to the given cell and returns true if the cell is equal to the given row',
 'Sets the bean property of the specified bean name with the specified value.',
 'Read DSL mapping file. the given file.',
 'Cancels the prune task and add the new one. time the program is closed.',
 'Returns the total width of the table.',
 'Returns the destination address for the channel uri.',
 'Setter. database connection string.',
 'Returns theHELM2 notation of the.omer.',
 'Reads table. TDB.']
```

(a) Produced summaries of DoubleEnsemble 4.4.3(Mean/Max)



```
['Retrieves an sbb aci data field value',
 'Check for equality of non - null reference x and possibly - null y .',
 'Sets petite bean property .',
 'This will load in the DSL config file using the DSLMapping from drools - compiler',
 'Schedules prune .',
 'Finds the longest row width .',
 'Get the endpoint address from the URI .',
 'Defines class input stream as a target .',
 'method to generate a valid HELM2 of this object',
 'this is the old Gempak table not as precise']
```

(b) Reference output of this subset of the testing dataset.

Figure 4.13: Summaries of model on a different subset of the testing dataset.

4.4.4 Summary

Below we present a table summarizing the BLEU-4 scores of the tested models. However, even if they can give an indication about a model's performance, it is advised not to be seen individually but in parallel to the produced summaries, since the BLEU scores refer to the lexical similarity, which is not the only objective in our case.

BLEU-4 scores	
Aggregation	**BLEU-4**
PLBart	15.49
GraphCodeBERT	12.05
mean (PLBart, GraphCodeBERT)	13.49
max (PLBart, GraphCodeBERT)	13.46
weighted mean (10 x PLBart, GraphCodeBERT)	14.55
weighted mean (2 x PLBart, GraphCodeBERT)	14.65
weighted mean (PLBart, 2 x GraphCodeBERT)	12.79
weighted mean (PLBart, 1.5 x GraphCodeBERT)	12.87

Table 4.3: Summary of BLEU-4 scores of the tested models and their ensemble.

BLEU-4 scores (continued)	
Aggregation	**BLEU-4**
weighted max (10 x PLBart, GraphCodeBERT)	14.52
weighted max (2 x PLBart, GraphCodeBERT)	14.42
weighted max (PLBart, 2 x GraphCodeBERT)	12.56
weighted max (PLBart, 8 x GraphCodeBERT)	12.35
weighted mean (50 x PLBart, weighted max (PLBart, 8 x GraphCodeBERT)) 4.4.3(Mean/Max)	15.30
weighted mean (10 x PLBart, weighted max (PLBart, 8 x GraphCodeBERT)) 4.4.3(Mean/Max)	**15.52**
weighted mean (10 x PLBart, weighted mean (PLBart, 10 x GraphCodeBERT))	15.42

Table 4.4: Summary of BLEU-4 scores of the tested models and their ensemble.

Chapter 5

Discussion

In this chapter, we analyze and interpret the results presented in 4.4. The conclusions coming from these findings are then presented in the next chapter 6 accompanied by their limitations and suggestions for future work.

5.1 Baseline Models' Results

In this thesis, PLBart and GraphCodeBERT are used as the base models of the final ensemble model but also as the baselines that make the comparison of the new and previous results possible. The purpose of this is to investigate whether the combination of these models can produce better results compared to their individual ones.

PLBart performed better compared to GraphCodeBERT in our implementation. Its BLEU-4 score was around 15-16 and its summaries were semantically accurate. On a lexical level, the similarity was not that significant, which is also illustrated by the relatively low BLEU-score compared to its original, by its creators, implementation. This could be because of various reasons. One of them is the used dataset. It was noticed that in some cases the provided as ground truth comment was not that accurate, but noisy or too general 6.3. Another factor is the size of the training dataset which was quite small. The training dataset size together with the non-extensive fine-tuning of the model led to a slightly over-fitting model that produces though informative summaries. From Figure 4.4 it can be seen that the over-fittingness is not strong but exists if we also consider the low training loss. However, PLBart's architecture, because of its attention mechanisms, is able to get the important information out of the code and translate it into natural language, providing this way more detailed summaries than the ground truth, without losing the overall semantic informativeness.

GraphCodeBERT performed slightly worse than PLBart. This can be seen in both its BLEU-score and its produced summaries 4.7. Its produced summaries are even more detailed and specific compared to the ones of PLBart, but still relate to the ground truth and can capture the functionality of the at the time code snippet. This is once again because of the not extensive fine-tuning of the model but in this case, it is more significant due to the changes that occurred in the configuration of the model. Like we mentioned in 3.2.3 and 4.4.2 GraphCodeBERT was compromised to make the aggregation possible. Both the tokenizer and the vocabulary size were adjusted to the PLBart's standards. Concerning the tokenizer, the pre-trained on PLBart tokenizer was used while concerning the vocabulary size, it was changed to 50005. This way, the produced probabilities of our NMT-models are in alignment and in position to produce the final summarized sequence. However, the model had to be trained from scratch since the weights produced by the GraphCodeBERT's pre-training could not be transferred. Therefore, the direct training on the task of code summarization, without the exploitation of any transfer learning, was more demanding and time-consuming. The model used in this thesis is not ideal but can capture the main concepts of the code snippets and provide some additional to the PLBart information.

We would like to clarify that since the objective of this thesis was not to produce a SOTA model but to test the efficiency of ensemble learning and show the way towards it for this specific task, the used PLBart and GraphCodeBERT implementations are not the best we could get. They are though, sufficient for the purposes of our work since they are of a similar level of performance and at the same time produce quite different results that would add their own value in a stacking-based model.

5.2 Ensemble Models

From the results presented in 4.3, several observations can be made referring to our approach.

At first, the simplest aggregation techniques, such as the mean and max, produce the less improved or less significant summaries. The BLEU-score ranges around the mean of the BLEU-scores of the two respective models and their summaries are relative to that. This means that they take almost equal information from both the models leading to a mixed summary that could be confusing. Some sentences 4.8 are cut in half (1st), in some cases one of the two models prevails (2nd), others combine non-related information from the two models (5th) and there are the occasions where the aggregation is actually beneficial (6th).

Moving to the more complex aggregations, where weights are put to the models to increase the influence of the one and decrease the influence of the other, the results are more promising and clear. In both cases, either where PLBart or GraphCodeBERT dominate the aggregation, the output summary is informative, meaningful and more complete. This is because one out of two models works as the base model that is being enhanced by the other one. It is obvious that when PLBart is used as a base model, the performance is better, but this is anticipated because of the better individual performance of the model compared to GraphCodeBERT. This can be seen both in the BLEU-scores illustrated in 4.4 but also in the Figures 4.10,4.11. In these cases where PLBart dominates, the produced summary is an enhanced PLBart summary with some details added by GraphCodeBERT. In the comparison of them with the outputs of the PLBart itself 4.4, the resemblance is apparent, something that makes the distinction of the GraphCodeBERT's contribution easier. The BLEU-scores though remain lower than the baseline model. This is not necessarily bad because of mainly two reasons. Firstly, as mentioned before, the ground truth comments are not very detailed and hence a deviation from a general comment would harm the BLEU-score. Secondly, the BLEU-score itself is focusing on lexical similarity. This similarity is also influenced by the size interval between the produced and the target sentence. Therefore, more information, besides the more specific information that was mentioned before, can also decrease the achieved score. On the other hand, when GraphCodeBERT dominates, the produced sentences are focusing on the details of the code snippets, especially when its weight is significant. In Figure 4.10b, where its given weight is just the double of PLBart's, the influence of PLBart is still significant and GraphCodeBERT is the one that plays the role of the helper. However, in Figure 4.11b the dominance of GraphCodeBERT is clear. The results, in both cases where GraphCodeBERT prevails, are of lower BLEU-score because of the aforementioned deviation from the ground truth but contain information that could be used in the documentation of a code snippet. Finally, concerning the aggregation techniques, weighted mean and weighted max, both perform more or less the same, with a small advantage given to weighted mean.

The final model that was tested, was inspired by the previous observations. Specifically, the results coming from a GraphCodeBERT-dominated ensemble model were similar to the original GraphCodeBERT's results, but were brought closer to PLBart's format after its influence. Therefore, the difference between these two models was decreased, but without loosing each other's identity and main characteristics. To this end, the enhanced GraphCodeBERT was also put into the equation in this aggregation procedure. The summaries produced by the DoubleEnsemble 4.4.3(Mean/Max) show good results both BLEU-score and human evaluation-wise. BLEU-score-wise, it produced the highest score among all the ensemble models that were tested (15.52) which was slightly higher than the maximum score among the baseline models (15.46). The produced summaries are following the PLBart's format because of its weight dominance, but at the same time introduce some new information provided by GraphCodeBERT. We also present a few summaries on a different subset of the testing dataset to illustrate the efficiency of this approach 4.13.

Chapter 6

Conclusions and Future work

This chapter is for presenting the conclusions of this thesis, spotting its limitations and suggesting ways that other researchers could use to tackle the same task in future works. At the end we also reflect on the outcomes from other levels than just answering the addressed research questions such as contributions on an economical, social, environmental and ethical aspect.

6.1 Conclusions

The goal of this thesis was to test whether the combination of two powerful models that tackle the same task from different angles, can occur efficiently, in a complementary manner and eventually produce meaningful, informative and improved summaries. During this testing, the necessary procedures to be considered are explored since such an implementation requires consistency in the outputs that comes out of the configurations of the used base models. These procedures are thoroughly described in 3.2.1 and 4.4.3 and show that special attention should be paid since these steps are not trivial.

As far as the performance of an ensemble learning-based model on source code description generation is concerned, there is no clear advantage in one of the developed models within our work but indications of promising outcomes and useful observations that could lead future work in this direction.

In our case, the base models produced significantly different next-token probabilities and consequently different summaries. They were both capturing the core idea of the code but eventually, they were focusing on different parts. This made the simple aggregation techniques, where the same or similar weights would be assigned to both models, less effective and more confusing. Such techniques would be fruitful in cases where the outputs of the used models do not present significant differences and hence one model could correct the other model's deviations by adding its "opinion" and influencing this way the final output. LeClair [17] validated this in their work when the AST-Flat+AST-Flat-FC ensemble was proven superior to others, less similar to each other models (e.g. Transformer + AST-Flat-FC), after using the simple mean aggregation. This did not occur in our case, making the models where one of the two takes the role of the dominant model, more effective. In this case, the dominant model's output prevails while the second model contributes by adding some extra information. This information is mostly added at the end of the dominant model's sentence because of our implementation of the aggregation where we pad the output probabilities tensor with zeros after the appearance of the end-of-sentence token. However, this is not a certain outcome since a model might have strong confidence about a token or sequence of tokens and hence dominate the aggregation even if it is the "helper" model. Similarly, the combination of the two models could bring into the spotlight tokens that did not appear in either of the models' summaries. An example illustrating the dominance of the "helper" model is the first summary of 4.11b where the part "pull event" had not appeared in GraphCodeBERT and another example referring to the appearance of new tokens is the ninth summary of 4.12 where the part "specified server" had not appeared in neither of the two model's outputs. In both cases, these tokens or sequences of tokens, had a high probability of appearing in that place of the respective sequence but this probability was not the highest at the time. It could be because of the greedy search's limitation that might assign a smaller probability to a token because of the prior probabilities or just because of the similarity of the appeared and not appeared tokens.

Another observation that was made after the interpretation of our results, was the significance of human evaluation. Metrics such as BLEU, offer an evaluation on the lexical level. However, in source code summarization the objective is not the production of a specific sequence such as in NL-to-NL machine translation. BLEU-score provides an indication of the performance of the developed model but it should be combined with human evaluation to a certain degree. Moreover, it should be used in a dataset of high-quality reference summaries, which is not always available. The docstrings that are used as references in this dataset, are drawn from different projects and developers and were not meant for the creation of an appropriate dataset in the first place. This could lead to confusing results, such as in weighted mean results 4.10a, where the summaries are very similar to PLBart's 4.4, but contain some additional information that is not necessarily wrong, yet decreases the BLEU-score anyway.

In conclusion, we continued LeClair's [17] work on ensemble learning for source code summarization. We tested more complex models and aggregation techniques than theirs to validate that their results were not incidental and to explore more ensemble learning aspects. The outcome was that such ensemble learning-based models can improve both simple and powerful models. Such procedure though requires attention in order to bring in alignment the base models' outputs. Once this is done, extensive weight tuning should follow. If time is not an issue, a meta-learner could be proven even more efficient. Besides these, the evaluation of the produced summary should be a combination of metrics like BLEU, CIDEr, METEOR, RIBES or ROUGE and human evaluation for a better and more representative interpretation of the summaries.

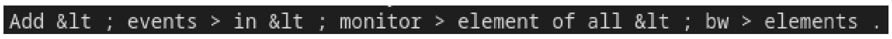
6.2 Limitations

The main limitations in this project are related to the available resources and time. Transformers are already heavy architectures [125] and the transformer-based models that were used are as well. The pre-training of such a model would require multiple GPUs to be running continuously for several days which could exceed the ten days. Therefore, the fine-tuning of a pre-trained model or the direct usage of one without any pre-training were the only choices in a transformer-based approach. The size of transformer-based models also influences our results, since further fine-tuning and experimentation would most likely lead to better performances. However, the objective of the project was covered with the available resources, thus these limitations were not eventually of a damaging level. Another limitation refers to the lack of human evaluation that would add a different, more complete interpretation of our results. The BLEU score, as mentioned in 6.1 is not always representative of the produced summary. It provides information only on a lexical level, which is not the objective of the developed models. An appropriate human evaluation would require five to fifteen specialized participants with at least three years of experience on the at the time programming language that are not part of the research in any capacity for providing unbiased evaluations. This would require more time and special attention to the selection of the participants in order to have informative and reliable feedback from them.

6.3 Future Work

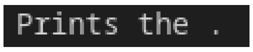
Several approaches concerning the task of source code description generation have been reviewed and considered. However, due to limitations on resources and time, only the most promising, according to our personal judgment, and also feasible approach was followed and tested. To the best of our knowledge, ensemble learning for the task of summarization, has been only used by [17]. However, it was tested on simple base models, so its contribution was not the presentation of a SOTA model, but the validation that ensemble learning can be also used for this task. The purpose of this thesis is similar and specifically aims at taking this one step further. Since ensemble learning looked promising, the objective here was to check if it also works for more complex and powerful models that can individually tackle this task from specific perspectives. If so, the next step would be to highlight what are the aspects that someone should consider when doing that. Therefore, since this was the focus, some issues were not considered but should be in the future. In this section, we will focus on these remaining issues that should be addressed in future work. These issues concern our implementation but also the general task of code summarization.

Concerning our implementation, the limitations we put on ourselves and could be easily fixed in future works, refer mostly to the used models. Firstly, we used pre-trained models provided from the hugging face library [107], something that limited us in the modification of the models. Personally developed models could offer some liberty that would also allow the relevant configuration that would make the models to be combined more compatible with each other. For example in our case, PLBart and GraphCodeBERT were pre-trained with different tokenizers and on different vocabularies. The fact that we modified these on fine-tuning could have decreased the performance of the respective model. Moving to the used models, there is a lot of room for improvement in their tuning. In this thesis, the necessary steps that would allow the aggregation of the models were taken, but no significant attention was paid to the full exploitation of these models. Well fine-tuned PLBart and GraphCodeBERT for the task of code summarization could had lead to a SOTA ensemble model, something that was not examined here. Specifically for GraphCodeBERT, which was not able to carry the weights from its pre-training because of its configuration modification 4.4.2, more data and more epochs accompanied by the necessary hyper-parameter fine-tuning would be also beneficial. Moreover, concerning the aggregation, more complex techniques could be tested, and specifically the usage of a meta-



```
Add &lt ; events > in &lt ; monitor > element of all &lt ; bw > elements .
```

Figure 6.1: Noisy ground truth.



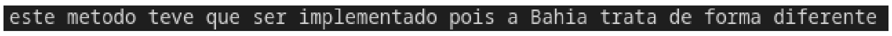
```
Prints the .
```

Figure 6.2: Missing important information from the ground truth code's comment.

learner. If resources and time are available, the training of a meta-learner, which is expensive, could lead to a more robust and reliable predictions' aggregation. In continuation of this, more than these two models could be used for testing our hypothesis or similar in the future. The addition of more models could lead to the discovery of a more appropriate combination of models but also would provide more information about the performance of the ensemble model compared to other already developed models that have shown interesting results. Finally, more languages could be tested or even a language-agnostic model, since the programming language of the snippet to be summarized is a significant factor in such problems.

The models though are not the only parts to which more attention can be paid on. While testing the results, it was discovered that the dataset itself was not ideal. Specifically, there were code comments that were noisy and would not provide any information, such as in Figure 6.1, and others that were too general or that were not deleted in the only English comment filtering, like in Figures 6.26.3. This decreases the representativeness of the BLEU-scores and makes the need for human evaluation even higher. Therefore, a more well-tended dataset could also improve the performance and the interpretability of the developed models.

As far as the general task of source code summarization is concerned, there are also alternative approaches that future researchers could follow to tackle the problems in the existing models that are mentioned in 1.2. Initially, there could be some fundamental changes in the architecture of some powerful models. These changes could include a different attention



```
este metodo teve que ser implementado pois a Bahia trata de forma diferente
```

Figure 6.3: Non-English comment. Pre-processing did not exclude all non-English docstrings from the testing dataset.

mechanism. As mentioned in [5] and validated by NeuralCodeSum [2] and CodeTransformer [14], the usage of relative positional attention has been proven efficient and effective with transformers. Therefore, also other, more powerful, transformer-based models that carry additional, useful characteristics could be pre-trained again with this modification to test this claim on more advanced models. The size of these models though, makes this try expensive time and resource-wise. Other ideas that came up after the literature review are the usage of contrastive learning or reinforcement learning. The contrastive learning objective could be added to the pre-training of a model with the scope of obtaining useful, informative code representations that could be eventually used for any downstream task. ContraCode [92] and [75] have shown promising results by exploiting similar and dissimilar code snippets and training their respective models with the objective of minimizing and maximizing respectively their representations' distance. Contrastive learning would be promising because of its non-dependency on extremely large datasets since similar and dissimilar code snippets can be created out of the existing data and hence enhance this way the initial dataset. On the other hand, in Reinforcement Learning, the at the time seq2seq model could act as the actor while a second critic-network would update the actor according to the loss induced from the ground truth in an actor-critic network manner. This is just another way to enhance the learning but does not specifically refer to any of our discovered problems. However, its robustness and efficiency accompanied by the increase in computational power make this approach worth trying.

6.4 Reflections

The contributing parties to this project are KTH Royal Institute of Technology (KTH) and Research Institutes of Sweden (RISE)[126]. There is scientific interest from both sides but also the outcome of the project could have positive influence in several levels. On an economic level, a reliable source code description generator could take that responsibility from a developer and at the same time produce informative comments that would clarify someone's code to another. The fact that the developers are liberated from the task of commenting sufficiently their code, gives them more time to focus on more productive tasks that would benefit them but also the company that they are working for. Additionally, as mentioned in 1, code comprehension takes a significant amount of time if there is no appropriate documentation [127]. These non-productive working hours are costly and affect organizations economically. The cost though is not only economical but psychological since according to [1, 128] code comprehension can take up to 58% of developers' time, and this could lead to frustration. So an improvement on that could also be considered a contribution on a social level because of its effect on the mental health of a large part of the society. This thesis does not benefit only the economy or part of the society, but also the environment. The usage of transfer learning, when possible, can reduce the time required for a model's training significantly. This means a limitation on GPU usage and hence less CO_2 emissions [129]. Finally, ethically wise, there was no human participant or data in our project so there are no concerns about any human-right violation. Concerning intellectual property rights, the used code contains parts of others' work that were always under the MIT License [130]. Moreover, the used Figures were all personal, made with the help of Miro [131] and diagrams.net [132], besides 2.1 that belongs to the freely-licensed images of Wikipedia [133].

References

[1] S. Haiduc, J. Aponte, and A. Marcus, “Supporting program comprehension with source code summarization,” *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE 10*, 2010. doi: 10.1145/1810295.1810335 [Pages 1 and 71.]

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in neural information processing systems*, 2017, pp. 5998–6008. [Pages 1, 2, 13, 36, 39, and 70.]

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016. [Page 2.]

[4] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *arXiv preprint arXiv:1412.3555*, 2014. [Page 2.]

[5] N. Chirkova and S. Troshin, “Empirical study of transformers for source code,” *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021. doi: 10.1145/3468264.3468611 [Pages 3, 4, 7, 16, 27, and 70.]

[6] Z. Tang, C. Li, J. Ge, X. Shen, Z. Zhu, and B. Luo, “Ast-transformer: Encoding abstract syntax trees efficiently for code summarization,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1193–1195. [Pages 3, 4, 7, 26, and 93.]

[7] I. Neamtiu, J. S. Foster, and M. Hicks, “Understanding source code evolution using abstract syntax tree matching,” in *Proceedings of the 2005 international workshop on Mining software repositories*, 2005, pp. 1–5. [Pages 3, 8, and 22.]

[8] F. E. Allen and J. Cocke, “A program data flow analysis procedure,” *Communications of the ACM*, vol. 19, no. 3, p. 137, 1976. [Pages 3 and 9.]

[9] R. Polikar, *Ensemble Learning*. Boston, MA: Springer US, 2012, pp. 1–34. ISBN 978-1-4419-9326-7. [Online]. Available: https://doi.org/10.1007/978-1-4419-9326-7_1 [Page 3.]

[10] O. Sagi and L. Rokach, “Ensemble learning: A survey,” *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, p. e1249, 2018. [Pages 3 and 18.]

[11] S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, and X. Xia, “Code structure guided transformer for source code summarization,” *arXiv preprint arXiv:2104.09340*, 2021. [Pages 4, 26, and 93.]

[12] H. Wu, H. Zhao, and M. Zhang, “Code summarization with structure-induced transformer,” *arXiv preprint arXiv:2012.14710*, 2020. [Pages 4, 7, 16, 26, and 93.]

[13] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “A transformer-based approach for source code summarization,” *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020. doi: 10.18653/v1/2020.acl-main.449 [Pages 4, 24, and 90.]

[14] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, “Language-agnostic representation learning of source code from structure and context,” *arXiv preprint arXiv:2103.11318*, 2021. [Pages 4, 25, 46, 70, and 92.]

[15] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “Graphcodebert: Pre-training code representations with data flow,” *CoRR*, vol. abs/2009.08366, 2020. [Online]. Available: https://arxiv.org/abs/2009.08366 [Pages 4, 5, 7, 16, 26, 29, 46, and 91.]

[16] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, “Unified pre-training for program understanding and generation,” *CoRR*, vol. abs/2103.06333, 2021. [Online]. Available: https://arxiv.org/abs/2103.06333 [Pages 4, 15, 28, 29, 46, and 94.]

[17] A. LeClair, A. Bansal, and C. McMillan, "Ensemble models for neural source code summarization of subroutines," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 286–297. [Pages 5, 18, 31, 34, 66, 67, 68, and 97.]

[18] A. Bansal, S. Haque, and C. McMillan, "Project-level encoding for neural source code summarization of subroutines," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 253–264. [Pages 5 and 18.]

[19] S. Ravichandiran, *Getting Started with Google BERT Build and Train State-Of-the-art Natural Language Processing Models Using BERT*. Packt Publishing, 2021. [Pages 5, 25, 39, and 41.]

[20] J. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "Editsum: A retrieve-and-edit framework for source code summarization," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 155–166. [Pages 5, 29, and 96.]

[21] D. A. Schmidt, "Programming language semantics," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 265–267, 1996. [Page 7.]

[22] Y. Wang, M. Huang, X. Zhu, and L. Zhao, "Attention-based lstm for aspect-level sentiment classification," in *Proceedings of the 2016 conference on empirical methods in natural language processing*, 2016, pp. 606–615. [Page 7.]

[23] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, "A multi-modal transformer-based code summarization approach for smart contracts," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 1–12. [Pages 7, 27, and 94.]

[24] X. Zhang, S. Yang, L. Duan, Z. Lang, Z. Shi, and L. Sun, "Transformer-xl with graph neural network for source code summarization," in *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2021, pp. 3436–3441. [Pages 7, 27, and 93.]

[25] S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, and X. Xia, "Code structure guided transformer for source code summarization," *arXiv preprint arXiv:2104.09340*, 2021. [Pages 7, 27, 46, and 93.]

[26] S. Kim, J. Zhao, Y. Tian, and S. Chandra, “Code prediction by feeding trees to transformers,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162. [Pages 7 and 27.]

[27] R. Dutta, *Encoding Abstract Syntax Trees (AST) via distance based self-attention mechanism*. University of California, Los Angeles, 2021. [Page 8.]

[28] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa, “Byte pair encoding: A text compression scheme that accelerates pattern matching,” 1999. [Page 10.]

[29] T. Kudo, “Subword regularization: Improving neural network translation models with multiple subword candidates,” *arXiv preprint arXiv:1804.10959*, 2018. [Page 10.]

[30] M. Schuster and K. Nakajima, “Japanese and korean voice search,” in *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2012, pp. 5149–5152. [Page 10.]

[31] T. Kudo and J. Richardson, “Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing,” *arXiv preprint arXiv:1808.06226*, 2018. [Page 10.]

[32] B. Wu, C. Xu, X. Dai, A. Wan, P. Zhang, Z. Yan, M. Tomizuka, J. Gonzalez, K. Keutzer, and P. Vajda, “Visual transformers: Token-based image representation and processing for computer vision,” *arXiv preprint arXiv:2006.03677*, 2020. [Page 11.]

[33] K. W. Church, “Word2vec,” *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017. [Page 11.]

[34] J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation,” in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543. [Page 11.]

[35] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching word vectors with subword information,” *Transactions of the association for computational linguistics*, vol. 5, pp. 135–146, 2017. [Page 11.]

[36] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014. [Page 12.]

[37] M. Geva, R. Schuster, J. Berant, and O. Levy, "Transformer feed-forward layers are key-value memories," *arXiv preprint arXiv:2012.14913*, 2020. [Page 14.]

[38] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019. [Pages 15 and 28.]

[39] Z. Xie, "Neural text generation: A practical guide," *arXiv preprint arXiv:1711.09534*, 2017. [Page 17.]

[40] S. Welleck, I. Kulikov, S. Roller, E. Dinan, K. Cho, and J. Weston, "Neural text generation with unlikelihood training," *arXiv preprint arXiv:1908.04319*, 2019. [Page 17.]

[41] S. Wiseman and A. M. Rush, "Sequence-to-sequence learning as beam-search optimization," *arXiv preprint arXiv:1606.02960*, 2016. [Page 17.]

[42] E. Garmash and C. Monz, "Ensemble learning for multi-source neural machine translation," in *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, 2016, pp. 1409–1418. [Page 18.]

[43] Y. Freund, R. E. Schapire *et al.*, "Experiments with a new boosting algorithm," in *icml*, vol. 96. Citeseer, 1996, pp. 148–156. [Page 19.]

[44] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997. [Page 19.]

[45] A. Natekin and A. Knoll, "Gradient boosting machines, a tutorial," *Frontiers in neurorobotics*, vol. 7, p. 21, 2013. [Page 19.]

[46] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996. [Page 19.]

[47] ——, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001. [Page 19.]

[48] B. Pavlyshenko, “Using stacking approaches for machine learning models,” in *2018 IEEE Second International Conference on Data Stream Mining & Processing (DSMP)*. IEEE, 2018, pp. 255–258. [Page 19.]

[49] M. P. Sesmero, A. I. Ledezma, and A. Sanchis, “Generating ensembles of heterogeneous classifiers using stacked generalization,” *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 5, no. 1, pp. 21–34, 2015. [Page 19.]

[50] B. Linghu and B. Sun, “Constructing effective svm ensembles for image classification,” in *2010 Third International Symposium on Knowledge Acquisition and Modeling*. IEEE, 2010, pp. 80–83. [Page 19.]

[51] P. M. Granitto, P. F. Verdes, and H. A. Ceccatto, “Neural network ensembles: evaluation of aggregation algorithms,” *Artificial Intelligence*, vol. 163, no. 2, pp. 139–162, 2005. [Page 19.]

[52] D. Movshovitz-Attias and W. Cohen, “Natural language models for predicting programming comments,” vol. 2, 08 2013, pp. 35–40. [Pages 21 and 87.]

[53] M. Allamanis, H. Peng, and C. Sutton, “A convolutional attention network for extreme summarization of source code,” in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2091–2100. [Online]. Available: https://proceedings.mlr.press/v48/allamanis16.html [Pages 22 and 87.]

[54] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Summarizing source code using a neural attention model,” *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016. doi: 10.18653/v1/p16-1195 [Pages 22, 23, and 87.]

[55] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, pp. 1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735 [Page 22.]

[56] X. Hu, Y. Wei, G. Li, and Z. Jin, “Codesum: Translate program language to natural language,” 08 2017. [Pages 22, 23, and 88.]

[57] U. Alon, S. Brody, O. Levy, and E. Yahav, “code2seq: Generating sequences from structured representations of code,” *arXiv preprint arXiv:1808.01400*, 2018. [Pages 22, 50, and 88.]

[58] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, “Improving automatic source code summarization via deep reinforcement learning,” *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018. doi: 10.1145/3238147.3238206 [Pages 22 and 88.]

[59] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura, “Automatic source code summarization with extended tree-lstm,” *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019. doi: 10.1109/ijcnn.2019.8851751 [Pages 23 and 88.]

[60] K. S. Tai, R. Socher, and C. D. Manning, “Improved semantic representations from tree-structured long short-term memory networks,” *arXiv preprint arXiv:1503.00075*, 2015. [Page 23.]

[61] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “Deep code comment generation,” *Proceedings of the 26th Conference on Program Comprehension*, 2018. doi: 10.1145/3196321.3196334 [Pages 23 and 29.]

[62] ——, “Deep code comment generation with hybrid lexical and syntactical information,” *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, 2020. [Pages 23 and 89.]

[63] A. LeClair, S. Jiang, and C. McMillan, “A neural model for generating natural language summaries of program subroutines,” *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019. doi: 10.1109/icse.2019.00087 [Pages 23 and 89.]

[64] Z. Zhou, H. Yu, G. Fan, Z. Huang, and X. Yang, “Summarizing source code with hierarchical code representation,” *Information and Software Technology*, vol. 143, p. 106761, 2022. [Pages 23 and 90.]

[65] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang, “Leveraging code generation to improve code retrieval and summarization via

dual learning,” *Proceedings of The Web Conference 2020*, 2020. doi: 10.1145/3366423.3380295 [Pages 24 and 89.]

[66] Y. Xia, T. Qin, W. Chen, J. Bian, N. Yu, and T.-Y. Liu, “Dual supervised learning,” in *International Conference on Machine Learning*. PMLR, 2017, pp. 3789–3798. [Page 24.]

[67] A. H. Payberah, “Recurrent neural networks,” Dec 2021. [Pages ix and 24.]

[68] P. Shaw, J. Uszkoreit, and A. Vaswani, “Self-attention with relative position representations,” *arXiv preprint arXiv:1803.02155*, 2018. [Page 25.]

[69] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and et al., “Codebert: A pre-trained model for programming and natural languages,” *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020. doi: 10.18653/v1/2020.findings-emnlp.139 [Pages 25, 29, 41, 46, 48, and 90.]

[70] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Learning and evaluating contextual embedding of source code,” in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 5110–5121. [Online]. Available: https://proceedings.mlr.press/v119/kanade20a.html [Pages 25, 41, and 91.]

[71] A. A. Ishtiaq, M. Hasan, M. M. A. Haque, K. S. Mehrab, T. Muttaqueen, T. Hasan, A. Iqbal, and R. Shahriyar, “Bert2code: Can pretrained language models be leveraged for code search?” *CoRR*, vol. abs/2104.08017, 2021. [Online]. Available: https://arxiv.org/abs/2104.08017 [Page 25.]

[72] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019. [Pages 25 and 41.]

[73] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018. [Pages 25 and 36.]

[74] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," *arXiv preprint arXiv:2003.10555*, 2020. [Page 25.]

[75] M. Hägglund, F. J. Peña, S. Pashami, A. Al-Shishtawy, and A. H. Payberah, "Coclubert: Clustering machine learning source code," in *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2021, pp. 151–158. [Pages 25, 70, and 91.]

[76] W. Wang, Y. Zhang, Z. Zeng, and G. Xu, "Trans^3: A transformer-based framework for unifying code summarization and code search," *CoRR*, vol. abs/2003.03238, 2020. [Online]. Available: https://arxiv.org/abs/2003.03238 [Pages 25 and 92.]

[77] C. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "Pymt5: Multi-mode translation of natural language and python code with transformers," *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020. doi: 10.18653/v1/2020.emnlp-main.728 [Pages 25 and 92.]

[78] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation," *arXiv preprint arXiv:2108.04556*, 2021. [Pages 26, 46, and 95.]

[79] F. Zhang, B. Chen, R. Li, and X. Peng, "A hybrid code representation learning approach for predicting method names," *Journal of Systems and Software*, vol. 180, p. 111011, 2021. [Pages 27 and 94.]

[80] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," *Proceedings of the 28th International Conference on Program Comprehension*, 2020. doi: 10.1145/3387904.3389268 [Pages 27, 30, and 97.]

[81] G. Yang, X. Chen, J. Cao, S. Xu, Z. Cui, C. Yu, and K. Liu, "Comformer: Code comment generation via transformer and fusion method-based hybrid code representation," in *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2021, pp. 30–41. [Pages 27 and 95.]

[82] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: Sequence-to-sequence pre-training for learning the representation of source code," *arXiv preprint arXiv:2201.01549*, 2022. [Pages 28, 46, and 94.]

[83] B. Wei, "Retrieve and refine: Exemplar-based neural comment generation," *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019. doi: 10.1109/ase.2019.00152 [Pages 28, 29, and 95.]

[84] S. Robertson and H. Zaragoza, *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009. [Page 28.]

[85] "Welcome to apache lucene." [Online]. Available: https://lucene.apache.org/ [Page 28.]

[86] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020. doi: 10.1145/3377811.3380383 [Pages 28, 29, and 95.]

[87] M. R. Parvez, W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," *Findings of the Association for Computational Linguistics: EMNLP 2021*, 2021. doi: 10.18653/v1/2021.findings-emnlp.232 [Pages 29 and 96.]

[88] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," *arXiv preprint arXiv:2004.04906*, 2020. [Page 29.]

[89] Y. Huang, M. Wei, S. Wang, J. Wang, and Q. Wang, "Yet another combination of IR- and neural-based comment generation," *CoRR*, vol. abs/2107.12938, 2021. [Online]. Available: https://arxiv.org/abs/2107.12938 [Pages 29 and 96.]

[90] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Automatic code summarization via multi-dimensional semantic fusing in GNN," *CoRR*, vol. abs/2006.05405, 2020. [Online]. Available: https://arxiv.org/abs/2006.05405 [Pages 29 and 96.]

[91] A. Y. Wang, D. Wang, J. Drozdal, M. Muller, S. Park, J. D. Weisz, X. Liu, L. Wu, and C. Dugan, "Documentation matters: Human-centered ai system to assist data science code documentation in

computational notebooks," *ACM Transactions on Computer-Human Interaction*, vol. 29, no. 2, pp. 1–33, 2022. [Page 30.]

[92] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. Gonzalez, and I. Stoica, "Contrastive code representation learning," *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021. doi: 10.18653/v1/2021.emnlp-main.482 [Pages 30, 70, and 97.]

[93] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2. IEEE, 2006, pp. 1735–1742. [Page 30.]

[94] F. Ye, S. Zhou, A. Venkat, R. Marcus, N. Tatbul, J. J. Tithi, P. Petersen, T. G. Mattson, T. Kraska, P. Dubey, V. Sarkar, and J. Gottschlich, "MISIM: an end-to-end neural code similarity system," *CoRR*, vol. abs/2006.05265, 2020. [Online]. Available: https://arxiv.org/abs/2006.05265 [Page 30.]

[95] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *arXiv preprint arXiv:1806.07336*, 2018. [Page 30.]

[96] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019. [Page 30.]

[97] J. Gu, P. Salza, and H. C. Gall, "Assemble foundation models for automatic code summarization," *arXiv preprint arXiv:2201.05222*, 2022. [Pages 31 and 97.]

[98] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, "Neural code summarization: How far are we?" *CoRR*, vol. abs/2107.07112, 2021. [Online]. Available: https://arxiv.org/abs/2107.07112 [Page 31.]

[99] L. Rokach, "Ensemble-based classifiers," *Artificial intelligence review*, vol. 33, no. 1, pp. 1–39, 2010. [Page 35.]

[100] "Where the world builds software." [Online]. Available: https://github.com/ [Page 35.]

[101] “Where developers learn, share, amp; build careers.” [Online]. Available: https://stackoverflow.com/ [Page 35.]

[102] “Your machine learning and data science community.” [Online]. Available: https://www.kaggle.com/ [Page 35.]

[103] S. J. Pan and Q. Yang, “A survey on transfer learning,” *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009. [Page 35.]

[104] Y. Liu, J. Gu, N. Goyal, X. Li, S. Edunov, M. Ghazvininejad, M. Lewis, and L. Zettlemoyer, “Multilingual denoising pre-training for neural machine translation,” *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 726–742, 2020. [Page 36.]

[105] D. Hendrycks and K. Gimpel, “Gaussian error linear units (gelus),” *arXiv preprint arXiv:1606.08415*, 2016. [Page 36.]

[106] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, “fairseq: A fast, extensible toolkit for sequence modeling,” in *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019. [Page 38.]

[107] “Hugging face – the ai community building the future.” [Online]. Available: https://huggingface.co/ [Pages 38 and 68.]

[108] “Pytorch.” [Online]. Available: https://pytorch.org/ [Page 38.]

[109] “Tensorflow.” [Online]. Available: https://www.tensorflow.org/ [Page 38.]

[110] “Jax quickstart.” [Online]. Available: https://jax.readthedocs.io/en/latest/notebooks/quickstart.html [Page 38.]

[111] “Introduction.” [Online]. Available: https://tree-sitter.github.io/tree-sitter/ [Page 43.]

[112] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Codesearchnet challenge: Evaluating the state of semantic code search,” *arXiv preprint arXiv:1909.09436*, 2019. [Page 46.]

[113] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” *arXiv preprint arXiv:1711.00740*, 2017. [Page 50.]

[114] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “A general path-based representation for predicting program properties,” *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018. [Page 50.]

[115] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318. [Page 50.]

[116] R. Vedantam, C. Lawrence Zitnick, and D. Parikh, “Cider: Consensus-based image description evaluation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 4566–4575. [Page 50.]

[117] A. Aizawa, “An information-theoretic perspective of tf–idf measures,” *Information Processing & Management*, vol. 39, no. 1, pp. 45–65, 2003. [Page 50.]

[118] A. Lavie and A. Agarwal, “Meteor: An automatic metric for mt evaluation with high levels of correlation with human judgments,” in *Proceedings of the second workshop on statistical machine translation*, 2007, pp. 228–231. [Page 50.]

[119] H. Isozaki, T. Hirao, K. Duh, K. Sudoh, and H. Tsukada, “Automatic evaluation of translation quality for distant language pairs,” in *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, 2010, pp. 944–952. [Page 50.]

[120] C.-Y. Lin, “Rouge: A package for automatic evaluation of summaries,” in *Text summarization branches out*, 2004, pp. 74–81. [Page 50.]

[121] J. Mahmud, F. Faisal, R. I. Arnob, A. Anastasopoulos, and K. Moran, “Code to comment translation: A comparative study on model effectiveness amp; errors,” *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, 2021. doi: 10.18653/v1/2021.nlp4prog-1.1 [Page 50.]

[122] A. Stent, M. Marge, and M. Singhai, “Evaluating evaluation methods for generation in the presence of variation,” in *international conference on intelligent text processing and computational linguistics*. Springer, 2005, pp. 341–351. [Page 50.]

[123] B. Chen and C. Cherry, "A systematic comparison of smoothing techniques for sentence-level bleu," in *Proceedings of the ninth workshop on statistical machine translation*, 2014, pp. 362–367. [Page 51.]

[124] T.-H. Jung, "Commitbert: Commit message generation using pre-trained programming language model," *arXiv preprint arXiv:2105.14242*, 2021. [Page 52.]

[125] S. Singh and A. Mahmood, "The nlp cookbook: Modern recipes for transformer based deep learning architectures," *IEEE Access*, vol. 9, pp. 68 675–68 702, 2021. [Page 67.]

[126] "Vi är sveriges forskningsinstitut." [Online]. Available: https://www.ri.se/sv [Page 70.]

[127] G. Sridhara, E. Hill, D. Muppaneni, L. 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*, 2010, pp. 43–52. [Page 70.]

[128] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017. [Page 71.]

[129] P. Henderson, J. Hu, J. Romoff, E. Brunskill, D. Jurafsky, and J. Pineau, "Towards the systematic reporting of the energy and carbon footprints of machine learning," *Journal of Machine Learning Research*, vol. 21, no. 248, pp. 1–43, 2020. [Page 71.]

[130] "The mit license." [Online]. Available: https://opensource.org/licenses/MIT [Page 71.]

[131] "The visual collaboration platform for every team: Miro." [Online]. Available: https://miro.com/ [Page 71.]

[132] "Diagrams.net - free flowchart maker and diagrams online." [Online]. Available: https://app.diagrams.net/ [Page 71.]

[133] "Free licenses," Jun 2012. [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia:File_copyright_tags/Free_licenses [Page 71.]

Appendix A

Tabularized previous work on Source Code Summarization

Investigated Models							
Name	Goal	Technique	Fine-tuned for	Pre-training	Tokenization	PL	Limitations
Statistical Language Model [52]	Comment Completion	n-grams and conditional probabilities	-	-	n-grams	Java	Expensive (O(exp n)), can not handle unseen data
Convolutional Attention Network [53]	Method-like Naming	CNN with attention and copy mechanism	-	-	Sequence of code-tokens	Java	Not a code description generator
CodeNN [54]	Code Summarization	Encoder-encoder LSTM with attention	Can be used also for the inverse task of code retrieval	(TRAIN) Supervised end-to-end training with backpropagation, predicting next word conditioned on the context	One-hot encoding	C#, SQL	LSTM is not parallelizable, no structural information, one-hot vector is too large

Table A.1: All the investigated in the literature models with their basic attributes summarized and their main limitations.

Investigated Models (continued)							
Name	Goal	Technique	Fine-tuned for	Pre-training	Tokenization	PL	Limitations
CodeSum [56]	Code Summarization	Encoder-encoder LSTM with attention on ASTs	-	(TRAIN) Supervised end-to-end training with backpropagation, predicting next word conditioned on the context	AST sequences with SBT linearization	Java, C#, SQL	LSTM is not parallelizable, sensitive to identifiers and different programming styles
Code2Seq [57]	Code Summarization	Encoder-Decoder LSTM with attention on ASTs treated as paths between terminal nodes (identifiers)	-	(TRAIN) Supervised end-to-end training with backpropagation, predicting next word conditioned on the context (use average of paths at the decoder)	Paths and paths' nodes encoding from ASTs	Java, C#	LSTM is not parallelizable, the order of input paths is random
Hybrid2Seq + Deep Reinforcement Learning [58]	Code Summarization	Encoder-Decoder, AST-based LSTM (for code strucure) + LSTM (for code content) with Hybrid attention, DRL (actor-critic NN)	-	Pre-train actor network with standard supervised learning with cross-entropy and critic with mean square loss. (TRAIN) Update both according to BLEU metric via gradient policy	Code and comment-token seqiences, ASTs	Python	LSTM is not parallelizable, uses tree-LSTM which cannot handle ASTs (trees) containing nodes with arbitrary number of ordered children, evaluation based on BLEU metric
Extended Tree-LSTM [59]	Code Summarization	Encoder-Decoder, multi-way Tree-LSTM (bi-directional LSTM) on encoding phase with attention	-	(TRAIN) regular seq2seq training	ASTs and NL-token sequences	Java	LSTM is not parallelizable, not significant improvement (indication of potential better, different approach)

Table A.1: All the investigated in the literature models with their basic attributes summarized and their main limitations (Part2).

Investigated Models (continued)							
Name	Goal	Technique	Fine-tuned for	Pre-training	Tokenization	PL	Limitations
ast-attendgru[63]	Code Summarization	Encoder-Decoder, 2 GRUs (code structure, code text separately), SBT-AO (replace all words from AST with OTHER), 2 attention mechanisms (text/comment and ast/comment)	-	(TRAIN) regular seq2seq training	ASTs with SBT-AO, split dataset by project not method	Java	LSTM is not parallelizable, performs better only when methods' names do not clearly state what the method does, no human evaluation
Hybrid-DeepCom [62]	Code Summarization	Encoder-Decoder, 2 GRUs (code structure, code text separately), SBT, hybrid-attention mechanism (fuse lexical and syntactical information), introduced Beam search instead of greedy(decoder)	-	(TRAIN) regular seq2seq training	AST by SBT, split identifiers in source code according to camel case (vacabulary is not enough otherwise), code and comments parsed into tokens with javalang and NLTK repsectively	Java	LSTM is not parallelizable, unknown word when identifier is mentioned in the comment (since there was no spliting there)
CO3 [65]	Code Summarization and Code Retrieval	Dual Learning, Encoder-Decoder for code summarization and code generation (simoultaneously), use hidden states of both for code retrieval task. Source code, Bi-LSTM with sharing of parameters	-	(TRAIN) regular seq2seq training for the 3 objectives	Code/text-token sequences + embedding matrix to map those into higher dimension vectors	SQL, Python	LSTM is not parallelizable, no syntax considered, python and SQL are closer to natural language

Table A.1: All the investigated in the literature models with their basic attributes summarized and their main limitations (Part3).

Investigated Models (continued)							
Name	Goal	Technique	Fine-tuned for	Pre-training	Tokenization	PL	Limitations
HACS [64]	Code Summarization	Hierarchical code representation, Encoder-Decoder LSTM	-	-	Token sequences, ASTs	Java	The longer the code the worse the summary (because of AST), fail to generate out of the given code words, LSTM-based
NeuralCode Sum [13]	Code Summarization	Transformers, copy mechanism, relative position representation	-	-	Camel and snake case split	Python, Java	code as plain text, standard transformer architecture
CodeBERT [69]	General purpose code representations	Transformers encoder, BERT, RoBERTa-base	Code search, code documentation generation	MLM (on bimodal data), Replaced Token Detection (on bimodal and unimodal data)	WordPiece (code as plain text)	Ruby, JavaScript, Go, Python, Java, PHP (training and testing) C# (only on testing)	No structure considered (worse than Neural-CodeSUM because there is no decoder and no relative positioning), no generation-based pre-training, good with languages that were not in the pre-training dataset but not as good as code2seq which uses ASTs, small-scale pre-training dataset

Table A.1: All the investigated in the literature models with their basic attributes summarized and their main limitations (Part4).

Investigated Models (continued)							
Name	Goal	Technique	Fine-tuned for	Pre-training	Tokenization	PL	Limitations
GraphCode BERT [15]	General purpose code representations	Transformers encoder (not only on code but on code-comment pairs to facilitate the learning on relative tasks) + Special position embeddings + Graph-guided Masked Attention	Code search, clone detection, code translation and code refinement	MLM, Edge Prediction, Node Alignment	Data flow graph (shallower and gets semantics instead of syntactics since programmers have different styles	Ruby, JavaScript, Go, Python, Java, PHP	Small-scale pre-training dataset, no generative pre-training tasks
CuBERT [70]	General purpose code representations	Transformers encoder, BERT-base (on code, specifically code-comment pairs when there exist a comment)	Variable-misuse classification, Wrong binary operator, Swapped operand, Function-docstring mismatch, Exception type, Variable-misuse localization and repair	MLM and Next Sentence Prediction	Python package-tokenize, split identifiers (snake and camel case) and strings on space and special characters (preserve semantically meaningful boundaries of tokens such as distinction between operators and operands), compress vocabulary into a subword vocabulary	Python	No structure considered/No code-based pre-training, no generation-based pre-training or fine-tuning, small-scale pre-training dataset
CoCluBERT [75]	General purpose code representations based on functionality	CuBERT, Contrastive Learning: Triplet/DR-C/Unsupervised	Code clustering	CuBERT	Code-token sequences	Python	few functionalities, method names are not always that informative about method's functionality; positive and negative to the anchor method could confuse the model

Table A.1: All the investigated in the literature models with their basic attributes summarized and their main limitations (Part5).

Investigated Models (continued)							
Name	Goal	Technique	Fine-tuned for	Pre-training	Tokenization	PL	Limitations
Code Transformer [14]	General purpose code representations	Transformers encoder, Context and Structure, relative position, multilingual	Method name prediction	Method name prediction, without pretraining in monolingual but with the options of taking method name prediction as pretraining task OR pretraining on the MLM task	Code token sequences, ASTs	Python, Javascript, Ruby, Go, Java	AST used as is; could contain noise, too long for a Transformer
TranS^3 [76]	Code Summarization and Unification of Summarization with Search	Deep Reinforcement Learning where actor network is a Transformer encoder-decoder. Uses the generated comment to enchance code search by enabling this new mapping	-	-	Natural language based on intervals and code based on a set of symbols. For code use Tree-Transformer encoder which incorporates indent based semantics and traverses the tree with pre-order traversal	Python	Only indents considered structure-wise which are not that informative for all languages, Tree-transformer relies a lot on quality of program forms which enhances the Python specification threat
PyMT5 [77]	Docstring generation (docstrings are different than comment, more technical than informative)	Transformer Encoder-Decoder	-	Span Masking Objective of T5	AST, ignore comments because they are trivial and not part of the normal language syntax	Python	Python specific, AST as is; increases transformers' complexity and decreases performance because of its size, also AST can be syntactically dense which can lead to similarity systems into memorizing syntax than learning semantics

Table A.1: All the investigated in the literature models with their basic attributes summarized and their main limitations (Part6).

Investigated Models (continued)							
Name	Goal	Technique	Fine-tuned for	Pre-training	Tokenization	PL	Limitations
AST-Transformer [6]	Code Summarization	Efficient encoding of AST for Transformers with ancestor-descendant and siblings relation matrices and tree-structure attention	-	-	AST and Pre-order traversal	Java, Python	AST can be syntactically dense which can lead to similarity systems into memorizing syntax than learning semantics, also AST may contain noise
SG-Trans[11]	Code Summarization	Transformer with structure-guided self-attention and hierarchical structure variant attention	-	-	Pair-wise relationships: (i) token (ii) statement (iii) dataflow (for global structure)	Java, Python	Expensive and not much better results than basic models
SiT [12]	Code Summarization	Tranformer on multi-view AST network, SiAttention	-	-	AST, control flow and data dependencies as adjacency matrices	Java, Python	Baseline models have been outperfromed by other approaches and here the improvement is not that significant
TPTrans-α[25]	General purpose code representations	Transformer using relative and absolute path of tokens across AST (GRU encoders and then integrated into self-attention)	Code Summarization	(TRAIN) transformer for method name prediction	ASTs by Tree-Sitter, camel case split	Python, Javascript, Go, Ruby	Large paths are causing performance reduction (due to AST size/noise). Absolute and relative paths overlap so better use just relative paths
Transformer-XL [24]	Code Summarization	ASTs' encoding via ConvGNN, fed into Transformer-XL (no decoder just Softmax(linear))	-	-	ASTs encoded via ConvGNN	Java	Only structural information as input (ConvGNN captures local semantic information)

Table A.1: All the investigated in the literature models with their basic attributes summarized and their main limitations (Part7).

Investigated Models (continued)							
Name	Goal	Technique	Fine-tuned for	Pre-training	Tokenization	PL	Limitations
MMTrans [23]	Code Summarization	Transformer on AST encoded as SBT sequence and ConvGNN	-	-	SBT, Graph (adjacency matrix), camel and snake case split and other preprocessing	Ethereum Smart Contracts	Same as Transformer-XL, large dataset for smart contract standards but limited due to its (method, comment) format
PLBart [16]	General purpose code and natural language representations	BART_base (encoder-decoder)	Code summarization, code generation, code translation, code classification	Pre-trained on unlabeled code and natural language (no bimodal-pairs) hence large-scale via denoising auto-encoding (generative task thanks to decoder)	code and text token sequences	Ruby, Javascript, Go, Python, Java, PHP, C#	No structural information and code and text are handled the same
SPT-Code [82]	General purpose code representations	Encoder (CodeBERT) - Decoder (GraphCodeBERT)	Code summarization, code completion, bug fixing, code translation, code search	Only on code (no need of bilingual corpus/no limitation) via Masked Sequence to Sequence (MASS), Code-AST Prediction (CAP), Method Name Generation (MNG)	Lexical analyzer to tokenize source code and further editing, simplified-SBT traversal for AST (X-SBT)	Python, Javascript, Go, Ruby, Java, PHP	No Classification pre-training (only generation but still performs good on classification tasks), simplified-AST could still be large
Meth2Seq [79]	General purpose code representations	Transformer Encoder-Decoder (decoder for the specific task)	Method name prediction	Method name prediction	control/data flow, path representation, intermediate representations (what each statement does), NL sequence	Python, Java	Has not been checked if it can generalize, just one training task, compared with no State Of The Art (SOTA) techniques, not that good in get-set methods

Table A.1: All the investigated in the literature models with their basic attributes summarized and their main limitations (Part8).

Investigated Models (continued)							
Name	Goal	Technique	Fine-tuned for	Pre-training	Tokenization	PL	Limitations
SynCoBERT [78]	General purpose code representations	Transformer's encoder	Natural language code search, code clone detection, code defect detection, program translation	Multi-modal Masked Language Modeling (MMLM), Identifier Prediction (IP), AST Edge Prediction (TEP), Multi-modal Contrastive Learning (MCL)	NL and code token sequences, AST with depth-first traversal	Python, Javascript, Ruby, Go, Java, PHP	Standard AST, no other structure methods that they are mentioning in introduction, a lot of elements taken from GraphCode-BERT, no generative tasks
ComFormer [81]	Code Summarization	Transformers, Multi-Modal (Jointly, Shared, Single encoder), Beam Search	-	-	Byte-BPE, further split for OOV words, SimSBT	Java	linearized AST directly into Transformer,no proper comparisons
Re^{2}Com [83]	Code Summarization	IR-based: Retrive (BM25, Lucene), Refine (BiLSTM encoder, LSTM decoder on input representation and exemplar)	-	-	Code token sequence, AST traversed by SBT-AO	Java	Lexical similarity; semantic similarity is considered afterwards but just to determine whether or on what degree the retrieved information would be used, LSTM (sequential input and no long range dependencies)
Rencos [86]	Code Summarization	IR-based: pre-trained attentional encoder-decoder (LSTM), syntactic and semantic similarity	-	-	Code token sequence (tokenize, javalang), AST, summaries (NLTK), snake-camel case split	Python, Java	LSTM-based, online code retrieval is expensive, non similar retrieved snippets could confuse the model

Table A.1: All the investigated in the literature models with their basic attributes summarized and their main limitations (Part9).

Investigated Models (continued)

Name	Goal	Technique	Fine-tuned for	Pre-training	Tokenization	PL	Limitations
REDCODER [87]	Code generation and summariization	Retrievers (CodeBERT, GraphCodeBERT), Generator (PLBART)	-	CodeBERT, GraphCodeBERT, PLBART	Code and NL token sequences	Java	Online code retrieval is expensive, non similar retrieved snippets could confuse the model (there might be semantic inconsistencies between input and retrieved code)
Hybrid-GNN [90]	Code summarization	BiLSTM (encoder), Lucene, Edit distance, global attention based dynamic graph + retrieval augmented static graph, GRU(fusion), attention-based LSTM decoder	-	-	Code Property Graph (AST with different type of edges)	C, tested on Python	Online code retrieval is expensive, non similar retrieved snippets could confuse the model, no semantics, LSTM-based
Dynamic IR-NN [89]	Code Summarization	Re^{2}Com's retriever, Cross-encoder (CodeBERT for semantic similarity), DeepCom	-	-	Code sequences, AST by SBT, identifier split, lower case	Java	Lexical similarity; if eventually not used we just have a simple DeepCom
EditSum [20]	Code Summarization	Jaccard, BM25, Lucene, Bi-LSTM (encoder), edit vector (semantic similarity based on differences after attention), LSTM (edit vector, prototype)	-	-	Code and NL token sequences, camel case-underscore split, lower case	Java	Lexical similarity, low usefulness score; possible redundant or not completely correct information

Table A.1: All the investigated in the literature models with their basic attributes summarized and their main limitations (Part10).

Investigated Models (continued)							
Name	Goal	Technique	Fine-tuned for	Pre-training	Tokenization	PL	Limitations
ConvGNN [80]	Code Summarization	seq2seq encoder-decoder with attention, Conv-GNN (for encoding AST), GRU (for encoding code token sequences)	-	-	Code and NL token sequences, AST by SrcML library	Java	No global dependencies, GRUs are not parallelizable
ContraCode [92]	General purpose code representations	Self-supervised contrastive learning; source-to-source compiler transfomation(generate semantically similar code on unlabeled data), encoders: BiLSTM and Transformer	Code clone detection, extreme code summarization, type inference	RoBERTa with MLM objective; maximize the similarity between positives without collapsing onto a single representation and minimize between negatives	AST, transformations (code compression, identifier modification, regularization)	JavaScript, TypeScript	Naming inconsistency from programmers does not allow ContraCode to perform significantly better than a transformer trained from scratch on extreme code summarization task
Ensemble [17]	Code Summarization	Bagging and stacking	-	-	AST, AST-FC (FC: uses other methods of the file to facilitate the learning of words out of the method itself), GNN, code sequences	Java	No real control, a lot of choices
ADAMO [97]	Code Summarization	Transfer Learning, CodeBERT, Refiner (AWGN), GTP-2, continuous pre-training, intermediate fine-tuning	-	CodeBERT, GTP-2, MLM (continuous for encoder), Casual Language Modeling (continuous for decoder), Concept Interpolation/Extrapolation/Annotation (intermediate fine-tuning)	Code and NL token sequences	Java, Python	Refiner's influence is not obvious even when it is well configured, pre-training encoder and decoder does not perform better than pre-training just decoder

Table A.1: All the investigated in the literature models with their basic attributes summarized and their main limitations (Part11).

TRITA-EECS-EX- 2022:00