FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Machine Learning Techniques for Source Code Pattern Recognition
Student:	Rudolf Raevskiy
Supervisor:	Mgr. Alexander Kovalenko, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2022/2023

Instructions

The task is to explore the possibilities of applying machine learning techniques to source code pattern recognition and matching.

1) Analyze the latest state-of-the-art approaches for source code classification using deep neural networks, including recurrent neural networks siamese neural networks, graph neural networks, and transformers.

2) Analyze and choose the optimal solution for the type of the model and input data (raw code, abstract syntax tree representation, dependence graph, etc.).

3) Define general and specific obstacles, constraints, and recommendations in the field of machine learning on source code.

4) Compare and suggest:
- various input preprocessing techniques;
- various neural network types;
- various neural network architecture;

5) Implement and test proposed applications.

Electronically approved by Ing. Karel Klouda, Ph.D. on 13 April 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Machine Learning Techniques for Source Code Pattern Recognition

Rudolf Raevskiy

Department of Applied Mathematics
Supervisor: Mgr. Alexander Kovalenko, Ph.D.

May 11, 2022

Acknowledgements

I'm extremely grateful to my supervisor, Mgr. Alexander Kovalenko, Ph.D., for all his help, support and patience during the writing of this thesis.

I would like to give a special thanks to my wife Valeria for her support, care, love and motivation. She was the key.

I am also grateful to my family and friends who always supported me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 11, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Raevskiy, Rudolf. *Machine Learning Techniques for Source Code Pattern Recognition*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Automatizované porozumění sémantice kódu je klíčové pro vývojáře při psaní spolehlivého a optimalizovaného kódu. V posledních letech roste zájem o aplikaci strojového učení ve zdrojovém kódu s cílem automaticky odhalovat chyby, komentovat kód nebo jej pochopit a vylepšit.

Tato práce uvádí techniky hlubokého učení aplikované na různé úrovně abstrakce zdrojového kódu. Experimentujeme se souborem dat, který se skládá ze zdrojového kódu jazyka R. Jazyk R má velkou komunitu převážně statistiků. Knihovny jazyka R však mají tendenci obsahovat neoptimální kód. Hlavním přínosem této práce je model natrénovaný na velkém souboru dat R, který je prvním krokem k automatizovanému nástroji pro psaní lepšího kódu R. Zaměřujeme se především na abstraktní syntaktické stromy (Abstract Syntax Trees, AST), ale uvažujeme i o jiných formách reprezentace. Různé abstrakce přidávají vstupním datům strukturu, a pomáhají tak lépe zobecňovat soubory dat.

Trénujeme a vyhodnocujeme několik modelů založených na různých reprezentacích kódu. Jako hlavní model pro tuto úlohu byla vybrána architektura založená na transformerech, protože v této oblasti dosahuje lepších výsledků než jiné modely. Trénování modelu na velkém souboru dat R je prvním krokem k automatizovanému nástroji pro psaní lepšího kódu R.

Výsledkem je *RASTaBERTa*, který je podle nás, nejmodernějším modelem založeným na transformátorech pro jazyk R a může být použit k dalšímu trénování pro specifické úlohy, jako je klasifikace, detekce chyb a anomálií, oprava chyb atd.

Klíčová slova Zdrojový Kód, Strojové Učení, Transformery, Grafy, Embedding, Podobnost

Abstract

The automated understanding of code semantics is crucial for helping developers write reliable and optimized code. In recent years, there has been a growing interest in applying machine learning to source code, with the aim of automatically discovering bugs, commenting or understanding and improving the code.

This work reports deep learning techniques applied to various levels of abstraction of the source code. We experiment with a dataset consisting of R language source code. R language has a large community of mostly statisticians. However, R libraries are prone to have suboptimal code. The main contribution of this work is a model trained on a large R dataset, which is the first step toward an automated tool to write a better R code. We primarily focus on Abstract Syntax Trees (AST), considering other representations forms just as well. Different abstractions add a structure to the input and therefore help to better generalize across the dataset.

We train and evaluate several models based on various code representations. The transformer-based architecture was chosen as a backbone model for the current task, as it outperforms its counterparts in this domain. Training a model on a large R dataset is the first step toward automatized tool to write a better R code.

As a result of *RASTaBERTa*, which is, to the best of our knowledge, the state-of-the-art transformer-based model for the R language and can be used for further training for specific tasks such as classification, bugs, and anomalies detection, bug-fix, etc.

Keywords Source Code, Machine Learning, Transformers, Graphs, Embedding, Similarity

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, we can hardly find a technology field that does not use a software. At the same time, the performance of the software directly depends on the quality of the related source code. Therefore the quality of a source code is of utmost importance as it can dramatically affect the whole domain's reliability and performance. One of the crucial aspects of helping developers write better code is an automated understanding of code semantics.

Semantic code understanding can be used for more efficient code completion, bug fixing, and detecting potential suboptimal patterns. Such suboptimal patterns can result in a slow program running and several types of errors.

Even though, early application of machine learning on source code, evolved from so-called "Naturalness Hypothesis" [1] that "Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools." Source code is much more structured than natural language.

While natural language emerges bottom-up, programming language (PL) is developed top-down and, in fact, is a formal language. Thus source code can be represented in many forms, such as sequence-based surface text representation (raw code), abstract syntax trees (AST), or any intermediate representations such as dependency-flow graphs, handcrafted syntactic features, or semantic analysis features.

Therefore, in the present work, we explore the possibilities of applying machine learning techniques to source code representations. As described above, the source code has several possible representations; thus, we explore how various models perform on various representations.

As a practical example, we explore the R-language dataset. R is a language developed by statisticians for statisticians, and despite a large number of users, it often contains suboptimal patterns as described above. The dataset and the model based on CodeBERT and fine-tuned on R, as well as the proposed model trained on R AST data, are open-sourced. Both models can be used as a foundation for a number of different tasks such as bug searching, clone detection, etc.

Chapter 2

Theory

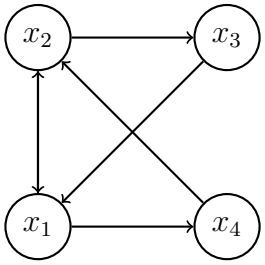
2.1 Graphs 101

Definition 1. ***Graph*** *is a pair (V, E), where V is a finite set of vertices and E a finite set of edges (paired vertices). Loop in a graph is an edge that joins a vertex to itself. Graph is called* ***directed****, when it has oriented edges,* ***undirected*** *otherwise. In this case we must redefine E as a set of ordered pairs of vertices as follows:*

$$E \subseteq \{(x, y) \,|\, (x, y) \in V^2 \quad and \quad x \neq y\}.$$

2.1.1 Adjacency Matrix

Definition 2. ***Graph Adjacency Matrix*** *is square $n \times n$ matrix A, that represents connections between the nodes in a finite graph. A_{ij} is equal to one when there is an edge between i-th and j-th nodes, zero - otherwise. In case of undirected graph the Adjacency Matrix is symmetric.*



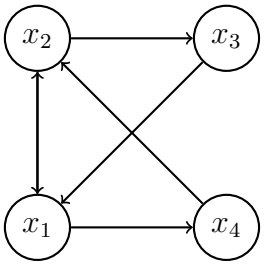
$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 2.1: Graph and It's Adjacency Matrix A.

In the case of large graphs, such an Adjacency Matrix may be vast and sparse. Therefore models are more challenging to learn.

2.1.2 Adjacency List

Definition 3. ***Adjacency List*** *is a collection of unordered lists, which represents a finite graph. Each unordered list within an Adjacency List contains a set of neighbors of particular node.*



node	neighbors
x_1	x_2, x_4
x_2	x_1, x_3
x_3	x_1
x_4	x_2

Figure 2.2: Graph and It's Adjacency List.

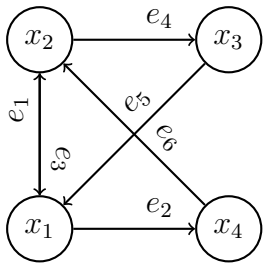
The main advantage of this representation is its small size, and therefore there is no need for a large amount of memory. Also, this version is more understandable and more transparent than the matrix. The only disadvantage of this option is the slow search for edges between the nodes, in fact, $O(|V|)$, while the Adjacency Matrix has a constant time.

2.1.3 Incidence Matrix

Definition 4. ***Incidence Matrix*** *is $n \times m$ matrix I, where n and m are the numbers of nodes and edges respectively.*

For undirected graph $I_{ij} = 1$ if the node x_i and edge e_j are incident, otherwise - 0. The sum for each column is 2.

For directed graph $I_{ij} = 1$ if the edge e_j enters vertex x_i, $I_{ij} = -1$ if it leaves vertex x_i, otherwise - 0. The sum of any column and row is 0.



	e_1	e_2	e_3	e_4	e_5	e_6
x_1	1	1	-1	0	-1	0
x_2	-1	0	1	1	0	-1
x_3	0	0	0	-1	1	0
x_4	0	-1	0	0	0	1

Figure 2.3: Graph and It's Incidence Matrix I.

2.1.4 Complexity Comparison

Each of the described graph representations has its trade-offs, so the choice depends on the concrete situation.

Operation	Adjacency Matrix	Adjacency List	Incidence Matrix	Incidence List
Add node	$O(\|V\|^2)$	$O(1)$	$O(\|V\| \cdot \|E\|)$	$O(1)$
Remove node	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$	$O(\|V\| \cdot \|E\|)$	$O(\|E\|)$
Add edge	$O(1)$	$O(1)$	$O(\|V\| \cdot \|E\|)$	$O(1)$
Remove edge	$O(1)$	$O(\|E\|)$	$O(\|V\| \cdot \|E\|)$	$O(\|E\|)$
Query edge	$O(1)$	$O(\|V\|)$	$O(\|E\|)$	$O(\|E\|)$

Table 2.1: Graph Operation Time Complexity.

As a rule, matrix representations take up much more memory but have an advantage in speed of operations, the lists vice versa.

Adjacency Matrix	Adjacency List	Incidence Matrix	Incidence List
$O(\|V\|^2)$	$O(\|V\| + \|E\|)$	$O(\|V\| \cdot \|E\|)$	$O(\|V\| + \|E\|)$

Table 2.2: Space Complexity (Average Case).

2.2 Source Code Representation

2.2.1 Tokens

During the *lexical analysis*, the lexer converts a sequence of characters into a sequence of tokens. Token, or in terms of PL, also called lexeme (often comparable to words in natural language), usually consists of its type or name and value. Common token types are:

Identifier Usually the name of variable/function/class

Keyword Programming Language reserved word (e.g. `for`, `function`, `while`)

Literal Numeric, logical, string, other literals (e.g. `TRUE`, `"Hello, World!"`, `1234.5`)

Operator Operation Symbol (e.g. `+`, `-`)

Separator Punctuation Character (e.g. `;`, `:`, `(`, `{`)

Comment Lines ignored by the compiler usually starting with symbols `//`, `#` or multi-line comments `/** */`

2.2.2 Abstract Syntax Tree

Abstract syntax tree (AST) is a data structure representing the syntactic structure and the source code in terms of its formal grammar. The compiler usually represents the code as ASTs at a particular stage of its syntax analysis (the step after lexical analysis), so this type of representation is essential.

Conversion of source code into AST is a reversible operation, so the order of expressions and operand positions must be explicit. Each node in the tree has its identifier, type, and value.

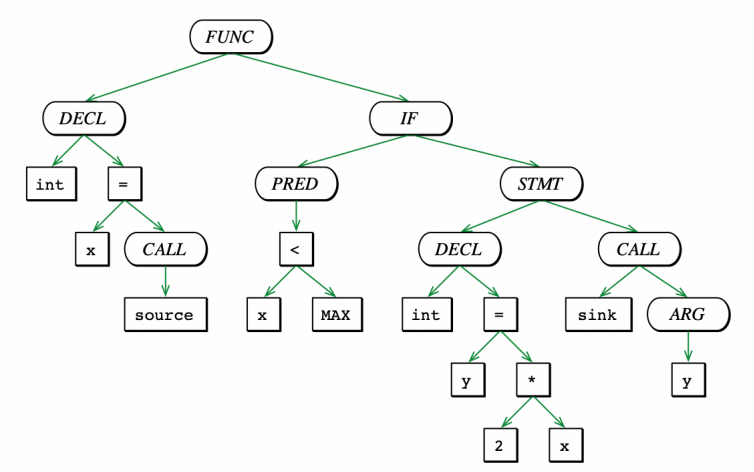


Figure 2.4: Abstract Syntax Tree Example [2].

2.2.3 Dependency Graph

A *Dependency Graph* is a directed graph that represents dependencies between objects. It has a widely used representation form in computer science. In programming languages, simple examples are data dependencies and control dependencies graphs. They can be used in compiler optimizations to find and eliminate the dead code (code that can not be executed and unused variables).

2.2.4 Program Dependence Graph

Program Dependence Graph (PDG, Figure 2.5) is an intermediate program representation that makes data and controls dependencies explicit [3]. It is used by most compilers in order to make transformations during optimization steps.

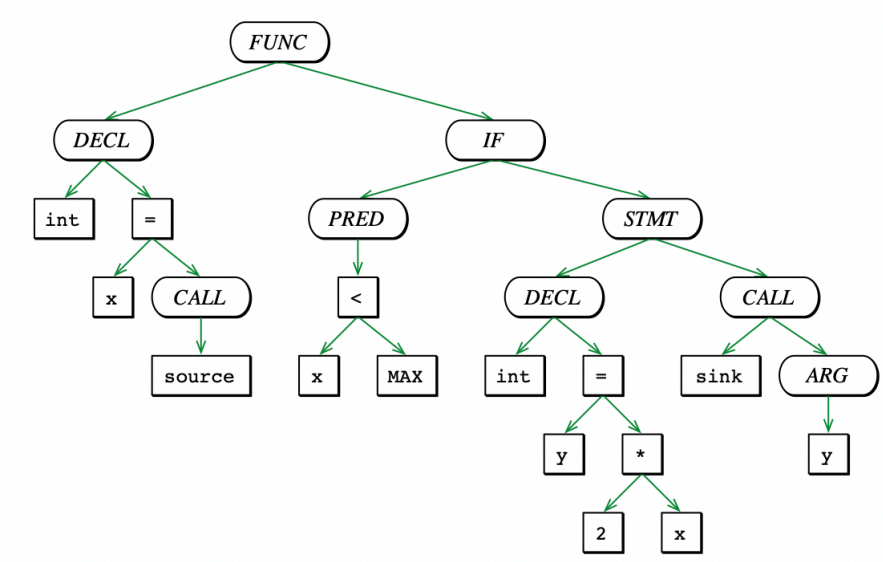


Figure 2.5: Program Dependence Graph Example [2].

2.2.5 Control Flow Graph

Control Flow Graph (CFG, Figure 2.6) represents the order of code execution and conditions for each node (single instruction) [4]. This type of graph is essential to many compiler optimizations and static analysis tools.

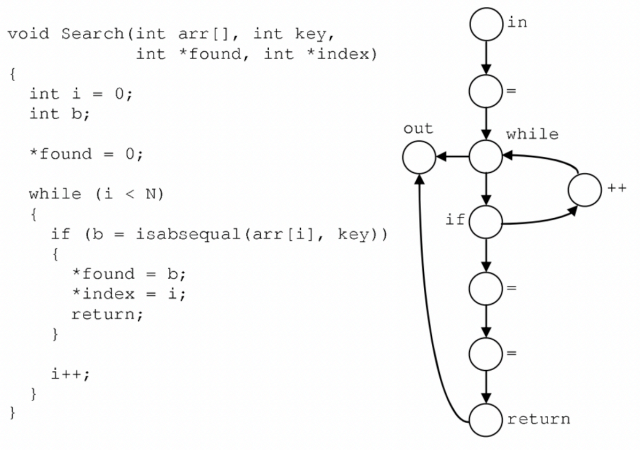


Figure 2.6: Control Flow Graph Example [5].

2.2.6 Code Property Graph

Code Property Graph (CPG, Figure 2.7) is a combination of AST, CFG, and DPG. Such intermediate representation is independent of the programming language. This robust and comprehensive data structure is widely used in source code analysis, searching for bugs and vulnerabilities.

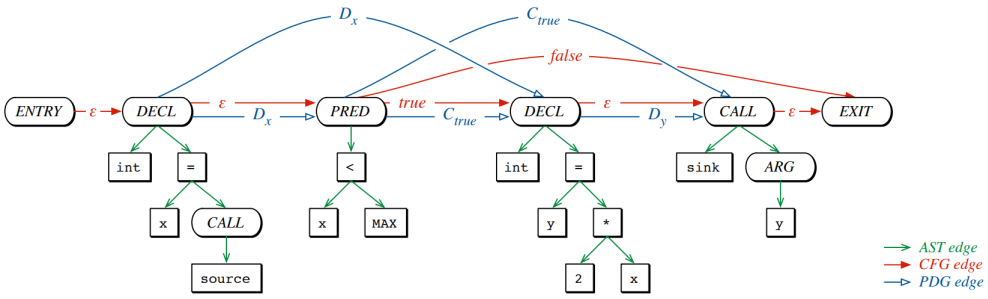


Figure 2.7: Code Property Graph Example.

2.3 Natural Language Processing

2.3.1 Word Embeddings

Definition 5. *Word embedding $e \in V$ is a learned distributed vector representation of a given word w, that maps it into Vector Space V.*

Embedding usually has a few hundred dimensions, unlike sparse methods, which have several dimensions proportional to the number of words in a corpus.

The Distributional Hypothesis proposed in paper *Distributional Structure* by Zellig S. Harris [6] says *"Words that occur in the same contexts tend to have similar meanings"*, so such words would have similar embeddings.

Methods, such as *word2vec* [7], Autoencoders, and embedding layers in Dense Neural Networks are typically used to learn real-valued embeddings for a predefined and fixed-sized vocabulary from a text corpus. However, there are also other approaches, which will be discussed later.

2.3.2 N-gram

Definition 6. *N-gram is a contiguous sequence of n tokens (words) from a given string.*

For example, *unigram* is just one word, *bigram* - two, and so on. *N--gram models* [8] use *Markov model* [9] as an approximation: we make an assumption, that each word depends only on the last n - 1 words.

2.3.3 Bag-of-Words

The first reference to this term in the NLP context was found in the already mentioned work of Zellig Harris [6]. BoW model is a bag (or multiset) representation of words that represents the word frequency within a document. Word order information is not used, making this method very simple, unlike pre-processing the data before using this method.

The algorithm looks as follows:

1. Create a vocabulary with unique words from all the documents. Each word will be associated with a specific position in future vectors.

2. The next step is to score each document, and there are several ways to do this.

    a) The first is to determine whether a word from vocabulary is in the document or not.

    b) Another way of doing this is to count the occurrences of each word.

    c) Calculate the word's frequency of occurrence in the document relative to all other words in the same document.

The most interesting part is pre-processing. Some words occur very frequently and do not affect semantics, so it is necessary to get rid of them. These words are called stop words (e.g. `"a"`, `"the"`, `"an"`, `"of"`). The lists of stop words for all major languages are freely available. Also, the same word can be used in different forms (e.g., words like `"drinking"` and `"drink"`), so you need to reduce them to their stem.

Pre-processing pipeline might look as follows:

1. Convert all words to the same case.
2. Remove punctuation.
3. Remove stop words.
4. Apply stemming algorithm.
5. Apply lemmatisation algorithm.

2.4 Graph Neural Networks

Graph Neural Network is designed to operate on graph data, preserving graph symmetries.

Graph level and node level embeddings are more complex than word embeddings due to structural information. The goal of a graph-level task is to predict the property of an entire graph. The node-level tasks focus on predicting the identity or role of each node in the graph.

When it comes to a graph, the graph's topology, the neighborhood of each node, node connections, and node features play a major role.

There are *spatial* algorithms that use all the information, including internal information from the nodes (e.g., embedding for a token that the node represents, or code token in our case), for example, *Graph2Vec* [10].

The other type is *spectral*, and it uses only general information about the graph and its structure (e.g., node degrees, clustering coefficient, adjacency matrix, etc.), for example, *FeatherGraph* [11].

Chapter 3

Related Work

The topic of applying machine learning techniques to source code has been developing very rapidly lately. Referring to *Papers With Code*[1] artificial intelligence in the field of computer code is used in 39 tasks, such as code generation, documentation generation, program repair, source code summarization, etc.

In the previous year, a model with the production name *Github Copilot* (which is actually a *Codex* [12]) became very popular. It's a GPT model fine-tuned on the source code from GitHub with 12B parameters. It can generate code based on a human description of the task in the form of a docstring and vice versa. The model weights and dataset are not publicly available.

This year, the first open-source model comparable in size to the GPT-3 was presented by *EleutherAI* [13]. *GPT-NeoX* is an autoregressive language model, which has 20B parameters. It has been trained on different types of data, including source code. The application of meta-learning techniques has recently become more popular. With using methods such as *Few-Shot Learning*, we can effectively use this and other models for source code tasks.

The team from *Carnegie Mellon University* has released open-source 2.7B model called *PolyCoder* [14], which was trained on data for 12 Programming Languages. *PolyCoder* outperforms GPT-3 in *C Lang*.

The authors of the previous two papers believe that open access to the data and source code of their projects is vital for developing this domain. Unfortunately, the field of machine learning is very costly because of the expensive hardware needed to train the models. Teams with access to such resources tend to achieve state-of-the-art, and this trend is rising.

[1] `https://paperswithcode.com`

Another transformer model called *AlphaCode* [15] raised the interest in solving the problem of competitive programming. The largest of the models presented by *DeepMind* has 41.1B parameters.

3.1 Source Code vs. Nature Language

Programming source code is basically a sequence of instructions or sequence of tokens, which is very similar to human language but uses certain rules and patterns which the compiler or interpreter will understand. Therefore, Nature Language Processing (NLP) tasks are comparable to the source code processing tasks (Table 3.1). In both domains, the same thing can be expressed in different ways.

character	character
token/lexeme	word
statement	sentence
code block	paragraph
source code file	document
corpus/repository	corpus

Table 3.1: Source Code vs. Nature Language Hierarchy.

3.2 The Naturalness Hypothesis

The Naturalness Hypothesis. *Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools.*

This hypothesis was introduced in the article "A survey of machine learning for big code and naturalness" [1].

Code is considered good if it optimally performs its task and is written in a clear and transparent way. Usually, this style is very similar to the way a person would try to explain something, but only using a specific syntax.

The meaning of program semantics and human semantics are different, so we need to make a clear distinction between them.

The variables and function names have no effect on the program performance and functionality, but they affect the readability of the code. The same thing can be elegantly written but work slowly, or incomprehensibly

and terribly written but work much faster. There are still problems that require expert knowledge, and no model can handle this and find the optimal solution for such a task.

3.3 Code Abstractions Diversity

There are several model types of models that work at different abstraction levels:

Token Working with token sequences (n-gram, Recurrent Neural Networks, Transformers).

Structural/Syntactic The input data are ASTs (e.g. *code2vec* [16]).

Semantic Generalization of two previous levels. This model type working with graph data (Graph Neural Networks).

Different models that work at these levels will be discussed in the following section. Then in Section 4.9, there will be proposed a model that works with AST at the token level. It combines two levels - Structural and Token levels.

3.4 Code Generating Models

3.4.1 N-gram model

N-gram[8] is a simple probabilistic language model predicting the next token by giving a sequence of $(n-1)$ tokens.

$$P(x_i \mid x_{i-(n-1)}, \ldots, x_{i-1}).$$

Before training, it is recommended to use an appropriate preprocessing pipeline. The next step is to count how many times each n-gram occurs. In the case of the bi-gram model (1-word context) and having these frequencies, it is easy to calculate which word occurs more often after our target word. For `(n > 2)`-grams the probability chain rule is used.

N-gram language model is very context-sensitive and is highly dependent on `n` parameter. The greater the value of `n`, the better the model, but it significantly increases the memory load. Also, code can be very comprehensive and detailed, so context gets lost very quickly.

Since all unique tokens cannot have connections to all others in a vocabulary, n-grams are sparse. This leads to the following problem the probability of a word that is not in the corpus is 0. The solution to this problem is a

technique called *smoothing*. The *Laplacian Smoothing* is a simple approach in which the 1 is added to every zero frequency.

In the code domain, such a model may produce syntactically incorrect results, which also depend on preprocessing step [1].

This model and its variations are widely used by source-code editors and plugins because it is a simple and lightweight model, but still, it is not effective as it only pays attention to the previous code tokens.

3.4.2 RNN

Recurrent Neural Networks [17] are networks with loops that are well suited for processing sequences (Figure 3.1).

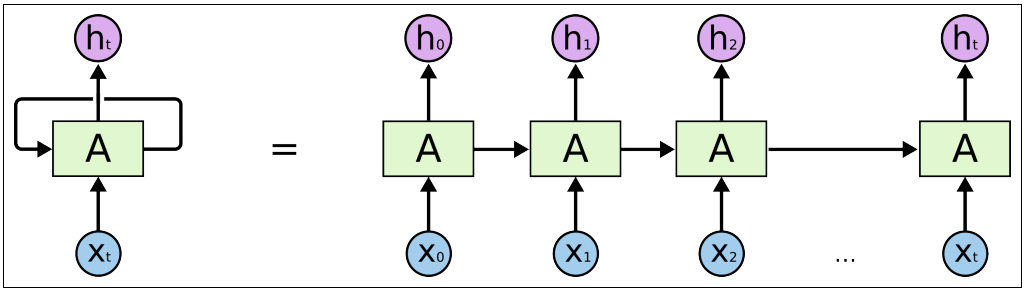


Figure 3.1: RNN Block.

RNN training is similar to training an ordinary neural network, but with a slight modification to the error backpropagation algorithm [18]. Since the same parameters are used at all temporal steps in the network, the gradient at each output depends not only on the computation of the current step but also on the previous temporal steps. Therefore, we need to keep track of the gradients not only at the current step but also at all previous steps. This algorithm is called Backpropagation Through Time or *BPTT* [19].

The *Long Term Memory Network* (LSTM) [20] is the most popular RNN architecture at the moment and is capable of learning long-term dependencies.

The state of an LSTM cell is similar to the hidden state of an RNN cell, but it is a much higher-dimensional vector. Each cell also has three gates: an input gate, a forget gate and an output gate. These gates control the flow of information into and out of the cell state, and they allow LSTM cells to effectively learn long-term dependencies.

RNNs are used when processing data that requires a consistent understanding, such as natural language. In the case of the code, things are more complicated because the interconnected tokens can be located at a large

distance. A single token can affect the change of the entire code block, its operability, and its semantics. This mechanism is flawed due to the *Vanishing Gradient Problem* [21], which leaves the model's state at the end of a long sentence without precise, extractable information about preceding tokens.

LSTM blocks are designed to improve information transfer from previous iterations of an RNN. However, the main problem with LSTM blocks is that the effect of previous states on the current state decreases exponentially with the distance between the words. The *Attention Mechanism* improves this factor to linear.

It is difficult to train RNNs efficiently because the dependency of token computations on results of previous token computations makes it hard to parallelize computation on modern deep learning hardware [22].

For these two reasons described above, RNNs are not the best option when working with code.

3.4.3 Transformer

The RNN's successor and outperforming replacement became an architecture called *Transformer* [22]. Transformers are designed to handle sequential input data (they do not use recurrent blocks and use the Attention Mechanism), but they don't necessarily process it in order as RNNs do. Therefore, it can more easily parallelize the training process than RNNs because it doesn't have to process the data in sequential order.

The original Transformer proposed in work "Attention Is All You Need" works on the encoder-decoder basis. Each encoder and decoder have self--attention. In addition, the decoder has another attention mechanism over the encoder layers. It also uses a multi-head attention mechanism to allow the model to focus on different parts of the input simultaneously. This is a generalization of the self-attention mechanism, where multiple attention heads are used. Each attention head calculates attention in a different space, and the results are concatenated and projected back to the original space [22].

The words of natural language in the sentence are connected to each other and can be represented as a fully connected graph, so the Transformer in this particular case can be called a Graph Neural Network with an attention mechanism [23]. Since the source code has the same properties and each token code has some relation and a context impact, this allows this to be applied to the code domain as well.

3.4.3.1 Bidirectional Encoder Representations from Transformer (BERT)

As the name implies, it is a model based on Transformer architecture that has been designed to pre-train the language representations in order to apply them further to various NLP tasks. It can be used for various natural language processing tasks such as text classification, question answering, and text generation.

Unlike other language models, *BERT* [24] trains context-dependent representations. For example, *word2vec* [7] always generates the same embedding for a word, even if the word is polysemous and its meaning depends on the context. BERT takes into account the context of a sentence and the meaning of the word in it, so the embedding may vary from sentence to sentence.

BERT is an *autoencoder* [25] trained simultaneously on two tasks: Next Sentence Prediction and Masked Language Modeling.

The backbone of BERT is the encoder stack. The encoder layers in this model use two-way attention, which allows them to take into account the context on both sides of the token being considered and, thus, more accurately determine the value of that token. By inputting tokenized pairs of sentences with some tokens hidden, BERT is able to learn a deep bi-directional representation of the language that allows it to better understand the context of a sentence.

The task of predicting the next sentence is a binary classification task. The network is trained to distinguish whether there is a connection between sentences in the text.

An example of a model working with code data based on the RoBERTa architecture (the successor of BERT) is CodeBERT . It has been trained on unimodal and bimodal data and achieves state-of-the-art results on downstream tasks, such as code search and documentation generation [26]. It will be mentioned in the perspective of fine-tuning in the Section 4.6.

3.5 Code Representation Models

3.5.1 Word2vec

One of the first successful and very important approaches is a group of algorithms called *word2vec* [7], that are used to produce distributed word embeddings.

Continuous Bag-of-Words and **continuous Skip-gram** architectures were introduced in *"Efficient Estimation of Word Representations in*

Vector Space" [7] by Tomas Mikolov (Figure 3.2). *CBOW* learns faster than *Skip-gram* and better represents more frequent words. At the same time *Skip-gram* have better embeddings for less frequent words and works good with small datasets [7]. *CBOW* approach is trying to predict the word by a given context words (in original paper the size of the context window for *Skip-gram* is 10 and 5 for *CBOW*), so the input vector consists of one-hot encodings of a context words, while *Skip-gram* predicts the context words by a given word.

To train such a model, we need to create a vocabulary of all unique words in our documents and then create pairs of all words in the context window for all the pre-processed documents. Taking the first sentence in this chapter as an example and given window size as 3, it might look like `(one, of)`, `(one, the)`, `(one, first)`, `(of, one)`, `(of, the)`, `(of, first)`, and so on. When training, the one-hot encoding of the first word in pair is input and the encoding of the second one is ground truth. This training step is made for every pair of words.

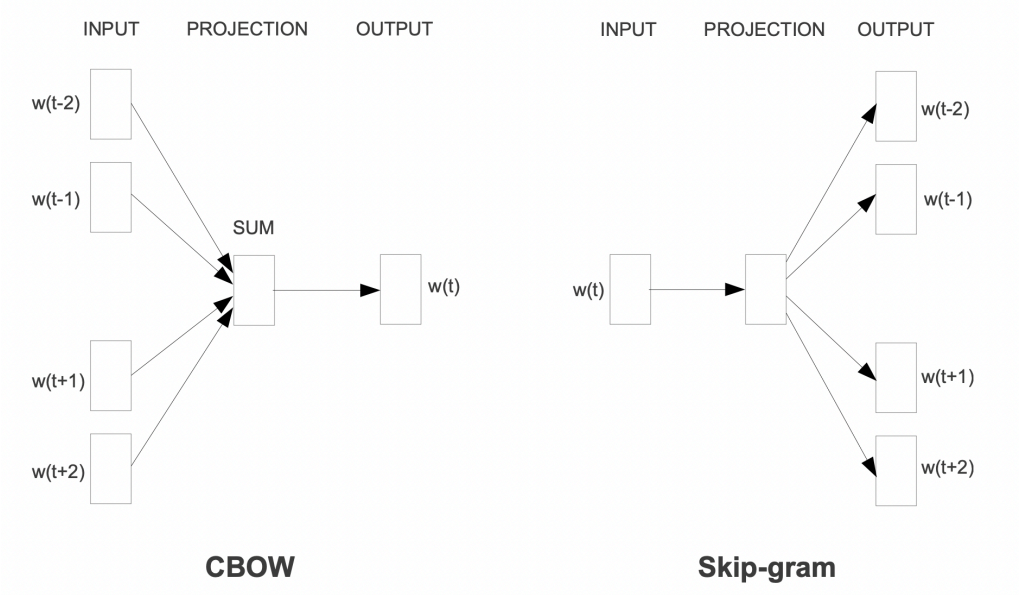


Figure 3.2: *CBOW* and *Skip-gram*Architectures [7].

The activation function is only used in the last layer. It's typically a *softmax* or another probability objective (loss) function, which returns the probability that the word will be in a selected context window. This task is called *fake* because the neural network will not be fully utilized with her outputs. The resulting embeddings we are interested in are contained in

the single hidden layer. The number of neurons in this layer determines the embedding length. The bigger it is, the more information it can contain, but it becomes more difficult to train such a model.

Also, the weights matrix grows linearly with increasing word count in the vocabulary. For example, in the case when embedding length equals 300 and vocabulary size is 50000, the matrix shape will be (300×50000). If the dataset is large, it becomes almost impossible to train such a model.

The first improving step is to delete all the frequent words like `the` because it does not affect paired word semantics and can be associated with almost any word.

The next possible upgrade is to use a *Sub-sampling*. At the step of vocabulary creation, the word will get there with a sample probability:

$$P(w) = (\sqrt{\frac{r(w)}{s}} + 1) \cdot \frac{s}{r(w)},$$

where w is a word, $r(w)$ is a frequency of w among all the documents in proportion to the number of all words, and s is a sub-sample parameter (default value is 0.001).

Sub-sample parameter affects the sub-sampling occurrence. The smaller the value of this parameter, the less chance that the word will get into the vocabulary.

Another important tweak is a *Negative Sampling* [7]. The point is to update only the weights for a specific number of randomly selected samples from some distribution (originally *Unigram*) [8], instead of all the words. A negative word in this context means a word for which the label is 0. It follows that rather than updating 15M weights, only $z * (k + 1)$ weights will be updated, where z is the embedding length and k is the number of negative samples. In addition, the weights for the input vector in the hidden layer are also updated, so k is incremented.

After this model appeared, it was used everywhere with any tokens, but there was no success with the code because of the reasons stated in Section 3.1. But such models, for example, help to represent information stored in AST nodes, and then it is possible to work with a more understandable structure and apply Graph models. This model provided the impetus and inspiration for the emergence of models such as *doc2vec* (a generalization of *word2vec*, which uses Distributed Bag of Words instead of CBOW and Distributed Memory [27]) and *graph2vec* [10] (which is very similar to *doc2vec*, but instead of sampling context words it samples subgraphs and uses Weisfeiler-Lehman kernel), with each subsequent model being the next level of abstraction and based on the previous ones.

3.5.2 Code2vec

Code2vec is a path-based attention model for learning distributed embeddings for code snippets. The model takes a code snippet and a corresponding label, caption, or name as input. A fully connected layer takes the embeddings of each path-context and combines them. The attention weights are learned using the combined context vectors and used to compute a code vector. After that code vector is used to predict the label (Figure 3.3) [16].

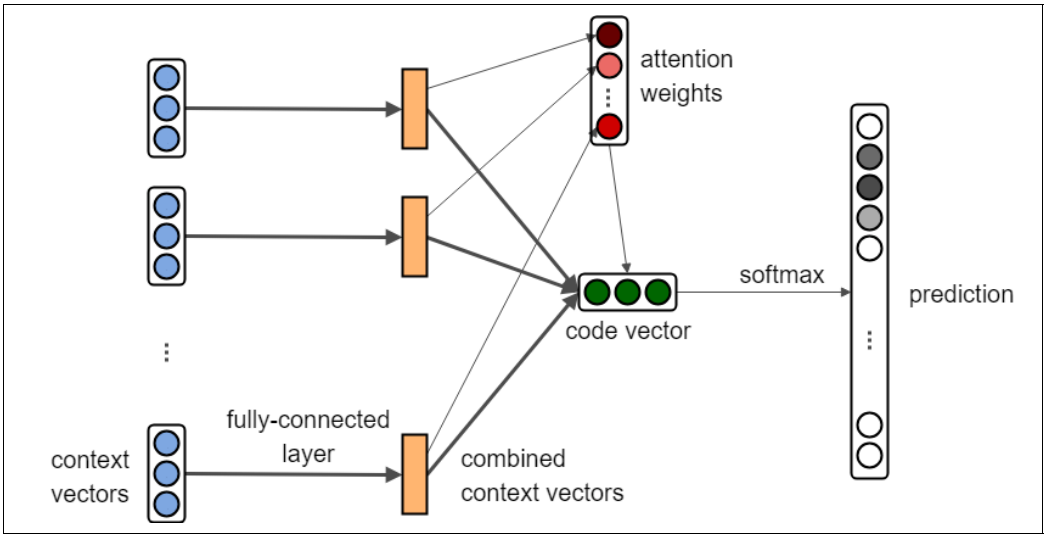


Figure 3.3: code2vec Architecture.

The main idea is that the distribution of labels can be inferred from syntactic paths. At the time of publication of this article, the model showed better results than previous works - `2x` better than *LSTM* and *CNN* with *Attention*.

The model has proven to be good at predicting the name of methods. And due to the attention mechanism, it is possible to understand which paths affected the prediction more.

Furthermore, I believe that the idea of applying syntactic paths can be developed further and can find its application with transformer models.

The model proposed in this paper also works with AST (Section 4.9), but in our case, the path is not extracted from the AST, which would probably simplify the transformer task but would change the preprocessing and learning algorithm.

Chapter 4

Current Approach

4.1 Setup

At the beginning of the work, the author used Google Colab Pro[2] (NVIDIA Tesla P100 16GB) and Apple M1 CPU. The *Programming Research Lab*[3] then provided the cluster (Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz, 64 cores, 126GB RAM, NVIDIA A100 40GB) for further experiments.

All the experiments in this work were conducted by using the languages R and Python.

To work with graphs and their processing were used libraries `pydot`[4], `networkx` [28], `karateclub` [29];

`numpy` [30], `pandas` [31], `matplotlib` [32], `scikit-learn` [33], `wandb` [34] were used to process the data and visualization;

`PyTorch` [35], `huggingface` [36], `gensim` [37], `umap` [38] were used for model training and clustering.

4.2 Data

4.2.1 R Code Dataset

The R programming language has been chosen for the experiments, as this work was created with the support of The *Programming Research Lab*, which is engaged in the development of the R.

[2] `https://colab.research.google.com`

[3] `https://prl-prg.github.io`

[4] Interface for Graphviz (`https://www.graphviz.org`)

R is an interpreted, dynamically typed language that is most often used for data analysis and modelling. The author created a dataset based on all the CRAN packages[5]. After downloading and parsing them, we received 281.3K Source Code files we can work with. Some of them may be written by inexperienced programmers since this language is mainly used by statisticians [39]. Since we cannot choose only well-written code from this or rewrite it because it must be done manually, this may affect the quality of further research and results.

It is assumed that all code is correct and can be run.

As an alternative source for our dataset, the author took 1000 of the most starred and open-source repositories from GitHub[6]. More than 50% of the files in these repositories are R source code. We recursively went through them and filtered the necessary data. As a result, we got 61.5 thousand R files. Some of these repositories are already from the CRAN, and after deleting duplicated data, 304.1K unique files turned out.

As a dataset for evaluation will be used, open data from the Kaggle[7]. This is meant R code and code from cells of `Rmd`/`irnb` notebooks. The size of this dataset is 10K files.

R is not as popular as other programming languages. Therefore there is significantly less open-source code. The comparison (Table 4.1) shows that this dataset is small compared to the other research datasets.

[5]Comprehensive R Archive Network (CRAN) is a repository of libraries and documentation for the R

[6]`https://github.com`

[7]`https://www.kaggle.com`

Dataset		# Files	Size
R Dataset	CRAN + GitHub	304.1K	
	Kaggle	10K	1.93GB
PolyCoder	Smallest (*Scala*)	245.1K	1.8GB
	⋮	⋮	⋮
	Largest (*Java*)	5.1KK	41GB
	Total	24.1KK	253.6GB
CodeBERT (Unimodal)	Smallest (*Ruby*)	164K	-
	⋮	⋮	⋮
	Largest (*JavaScript*)	1.8KK	-
	Total	6.4KK	-
AlphaCode	Smallest (*Rust*)	320K	2.8GB
	⋮	⋮	⋮
	Largest (*C++*)	21.5KK	290GB
	Total	86.3KK	715GB
Codex	Python	-	159GB

Table 4.1: Dataset Size Comparison.

The dataset will be placed in the public domain so that anyone can use it for their own purposes.

4.2.2 Preprocessing

This thesis does not address the *NL-PL*[8] problem, so the code is cleared of all the comments with human text and unused parts.

Since every sign and every word is important, we won't use NLP preprocessing steps such as removing stop words, removing punctuation, stemming, etc.

4.2.3 R ASTs Preprocessing

The following example demonstrates the output AST for the Min-Max Normalization function written in R (Listing 4.1 and Figure 4.1).

[8]NL stands for Natural Language, PL for Programming Language respectivly

```r
MinMaxNorm <- function(x) {
    return ((x - min(x)) / (max(x) - min(x)))
}
```

Listing 4.1: Min-Max Normalization Function.

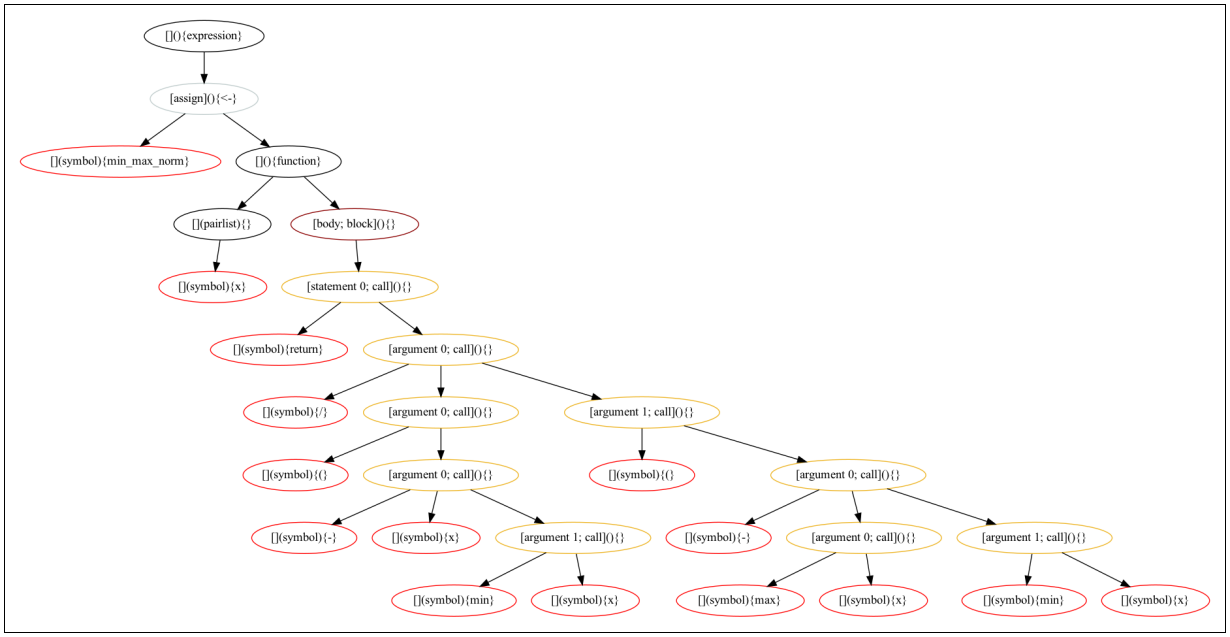


Figure 4.1: AST for Min-Max Normalization.

After extraction the ASTs with the help of *asttoolsr* [40] from all the source code it ended up with 15.43GB of *Graphviz* graphs. Every graph node has a **semantic information** (may be empty), **value** and **value type** (in case if the node value is a literal, otherwise empty) properties. It's necessary to vectorize it in order to pass those embeddings further.

Semantic information and value type can only have a predetermined and small number of values, which is known in advance:

```
- call name          - iteration variable  - integer
- body               - range               - pairlist
- false branch       - logical             - control flow
- true branch        - symbol              - str
```

- real	- argument	- assign
- block	- complex	- NULL
- variable	- call	- parameters
- statement	- condition	- litteral

Encoding them using the *MultiLabelBinarizer* [33] allows getting one-hot encoding of the same length for inputs with a different number of the predefined features. The result is two embeddings of length 24.

Next, pre-trained *doc2vec* (Section 4.10.1) model is used to get the 300-dimensions embedding for the last *value* feature.

The last step is to concatenate them into one feature vector with a fixed length of 348. It is important to note that these embeddings are not node embeddings because they do not contain information about the topology of the graph and neighbor nodes. These steps are repeated for all nodes in all graphs.

Multiprocessing is used to speed up the process by using 16 CPU workers. To process so many graphs, it was necessary to split the original ASTs into 40 chunks (our setup had 128 gigabytes of RAM). This processing took 24 hours and 50 minutes.

The total size of the processed graphs (pickled with LZMA compression, so the speed of backloading into memory will be slower) together with the embeddings is 216GB. Such a large size is a strong obstacle to learning the model and requires a large amount of RAM.

For the convenience of working with graphs, I will use the *NetworkX* [28] library and store graphs in this format. Other libraries, such as *StellarGraph* [41] have utilities for converting it to internal formats. Each graph is a code snippet. The code data in these graph nodes will be stored in the form of embeddings.

Also, it is necessary to create a disjoint union of AST graphs into one graph to work with (it is needed for Graph Models working in the Transductive model). By definition of disjoint union, the resulting graph is necessarily disconnected. By giving one node from this large graph, we can simply find the whole component it belongs to, so we can return to the source code snippet form. This operation is very time expensive, in fact it's $\mathcal{O}(\alpha(n))$, where the $\alpha(n)$ is inverse Ackermann function. To simplify this and use the usual graph union, each node must have a unique id.

Putting it all together, it turns out the following pipeline:

1. Source Code Files → ASTs.
2. Parse nodes information.
3. Nodes labeling (Unique ids).
4. Get feature embeddings.
5. Save it as *networkx* graphs.
6. Get graphs union.

4.3 Subword Tokenizers

Before training the Transformer model, it is necessary to prepare and train a tokenizer. Subwords help to counteract the tokenization of words that are not in the vocabulary. It can also be useful when dealing with tokens in which an error has occurred. Transformers also operate with special tokens. Their format may be different for each model.

Subword tokenization is very useful not only for NL, but also for PL, because function, variables and property names are agglutinative and as a rule, they are composed of multiple words (in case of code - *lexemes*, e.g. `var itemCollection`, `function arraySort()`, `function getNameById()`). This way, it increases the number of words recognized by the tokenizer and solves the problem of the appearance of new words that are not in the vocabulary. Tokenizers' vocabulary cannot be very large. Otherwise, the embedding matrix would be very large.

The tokenization process consists of several stages:

Pre-tokenizing The goal of this step is to split the input sequence into words (or tokens in people's understanding) via certain rules.

Normalizing This step can be described as a conversion to a certain format. For example, lowercasing, getting rid of diacritics, Unicode normalization, etc.

Post-processing During this stage the special tokens (e.g. `[CLS]`, `[SEP]`, `[UNK]`) are placed in the right places. This can be marking the beginning and the end of a sequence, punctuation marks, masks, replacing unknown tokens, and so forth.

Despite the fact that this work will mainly use the *BPE* tokenizer, it is worth noting the other algorithms for comparison.

4.3.1 Byte-Pair Encoding

After the pre-tokenization step BPE [42] creates a base vocabulary of all unique characters from all the words that appeared after the previous stage. Then, BPE applies merges frequently occurred tokens in order to concatenate them into a new one. This process continues until the value of the dictionary size parameter is reached. This tokenization algorithm is used by *GPT-2*, *RoBERTa*, etc.

The *Byte-Level BPE* tokenizer will split a string of text into tokens based on the number of bytes in the string. This can be useful for languages that use a lot of non-ASCII characters, such as Chinese or Japanese.

The *GPT-2* paper [43] discusses how to deal with the fact that the base vocabulary needs to include all base characters, which can be quite large if one allows for all Unicode characters. The paper introduces a clever trick of using bytes as the base vocabulary, which gives a size of 256. With some additional rules to deal with punctuation, this manages to be able to tokenize every text without needing an unknown token. For instance, the *GPT-2* model has a vocabulary size of 50257, which corresponds to the 256 bytes base tokens, a special `end-of-text` token, and the symbols learned with 50000 merges.

4.3.2 Wordpiece

WordPiece [44] is a subword tokenization algorithm that is used for *BERT*, *DistilBERT* and others. It is based on the same principles as *BPE*. The difference is that WordPiece does not choose the pair that is the most frequent but the one that will maximize the likelihood of the corpus once merged.

The algorithm uses the famous `##` prefix to identify tokens that are part of a word (i.e., not starting a word).

4.3.3 Unigram

Unigram [45] algorithm starts with a large vocabulary and gradually eliminates less common words and phrases. It is not used directly for any of the pre-trained models in the library but is usually used in combination with the *SentencePiece* [42] algorithm.

Unlike BPE, Unigram is not deterministic based on a set of rules applied sequentially. Instead, Unigram computes multiple ways of tokenizing and chooses the most probable one.

4.3.4 Training

We chose *Whitespace* pre-tokenizer for the simple separation of tokens by their boundaries.

A list of all special characters is as follows `[UNK]`, `[CLS]`, `[SEP]`, `[PAD]`, `[MASK]`.

Mask token is especially important for us because we will be training Masked Language Models.

The normalizer uses *NFD* unicode normalization, *StripAccents* to remove all accent symbols, and *Lowercase* normalization for conversion to lowercase.

As we prepare for BERT model training, the post-processor, in this case, adds special tokens `[CLS]` and `[SEP]`.

A large vocabulary size will result in a large embedding matrix, which can lead to memory issues. Most transformers models have a vocabulary size of 50000 or less, especially if they are trained in a single language.

For our task, the vocabulary size of 30000 was chosen, but to compare the speed of learning, the size of 50000 is also considered.

Tokenizer	Wall Time	CPU Total Time	Vocab Size
Byte-Level BPE	8min 35s	3h 58min 5s	30K
Byte-Level BPE	10min 17s	5h 20min 56s	50K
BPE	6min 48s	2h 31s	30K
BPE	9min 11s	2h 54min 22s	50K
WordPiece	6min 50s	2h 20min 18s	30K
WordPiece	9min	2h 59min 55s	50K
Unigram	1h 31min 29s	2h 40min 16s	30K
Unigram	1h 15min 52s	2h 33min 27s	50K

Table 4.2: Rough Benchmarks on Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz.

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
BPE	**Pre-tokenization**	do	.	dne	<-	function	(	X	,	label	,	ndim	=	2	,	numk	=	max	(	ceiling	(
	Tokenization	do	.	d	ne	<-	function	(	x	,	label	,	ndim	=	2	,	num	k	=	max	(
WordPiece	**Pre-tokenization**	do	.	dne	<-	function	(	X	,	label	,	ndim	=	2	,	numk	=	max	(	ceiling	(
	Tokenization	do	.	dn	##e	<-	function	(	x	,	label	,	ndim	=	2	,	num	##k	=	max	(
Unigram	**Pre-tokenization**	do	.	dne	<-	function	(	X	,	label	,	ndim	=	2	,	numk	=	max	(	ceiling	(
	Tokenization	do	.	d	n	e	<-	function	(	x	,	label	,	n	dim	=	2	,	num	k	=

Table 4.3: Tokenization Example.

4.4 Masked Language Modeling

Masked Language Modeling is the task where the model must predict the token to be instead of the mask token based on the context. Input data can contain several masks. It's also a self-supervised approach, so it does not require any label processing. During training, tokens are replaced with special mask tokens with a certain probability, and the model task is to predict them.

After training, we will not only get a useful model that can be applied to code auto-completion or other tasks, but we can also collect embeddings from the final hidden layer and use them for clustering or other similarity tasks.

4.5 BERT

The choice of this model was motivated by the presence of context-dependent embeddings. As with natural language, context is very important. The context in which the function is executed greatly affects its output.

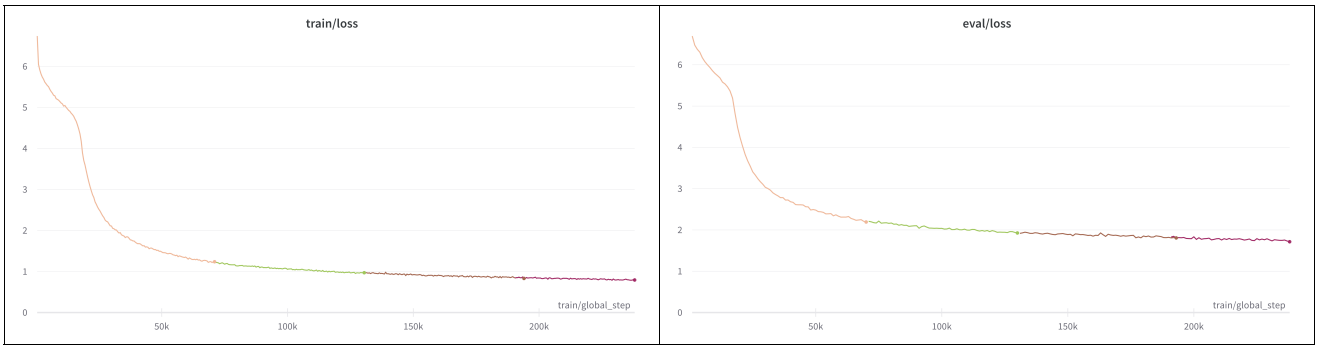
All the following models that we will be testing are improvements of this architecture.

4.5.1 Training

A standard model architecture (12 hidden layers and 12 attention heads) without any tweaks was chosen. It was found that even training a small model (compared to the rest) from scratch is almost impossible.

As BERT is one of the most simple models that show good results in NLP tasks, we further consider BERT trained on code as a baseline model.

More information and used parameters about this and other models can be found in the Table 4.12.



(a) Train Loss (b) Validation Loss

Figure 4.2: BERT Training.

4.6 CodeBERT

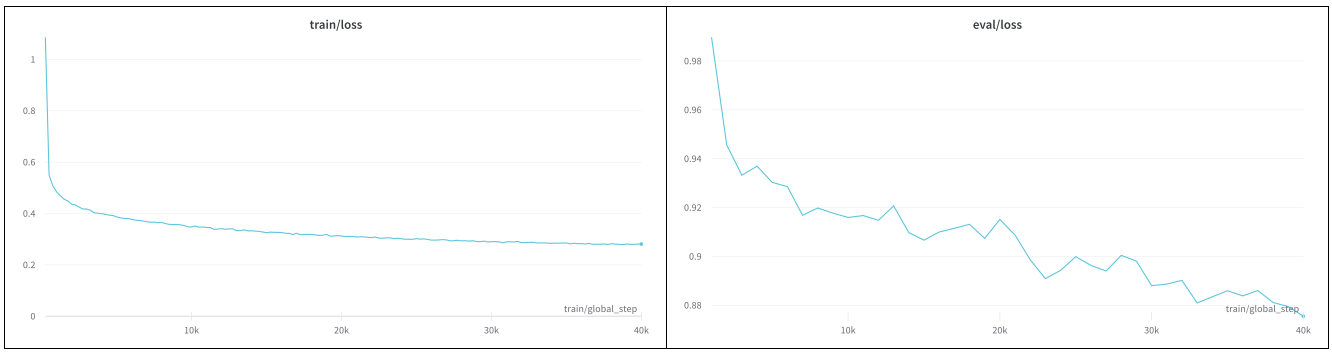
This model was pre-trained on the MLM task using the NL-PL dataset of six programming languages (Python, Java, JavaScript, PHP, Ruby, Go) [26]. These languages, especially Python and Go, are visually similar in their syntax and are often used in the same fields as R. A person who already knows several programming languages will not have difficulty learning a new one that is similar in syntax and application. Because of this analogy, this model is a useful starting point for the train it further on the R dataset.

4.6.1 Fine-tuning

In the first attempts to fine-tune, the loss had a very large variation, even larger than in the case of the BERT. To solve this problem, it was decided to increase the probability of dropout, add a small warmup for the learning rate and add weight decay.

This is the best result achieved on the raw code sequence dataset.

As will be seen further in Section 4.9.5, the result of all the models, including this one, could have been better if the sliding window split technique had been applied.



(a) Train Loss (b) Validation Loss

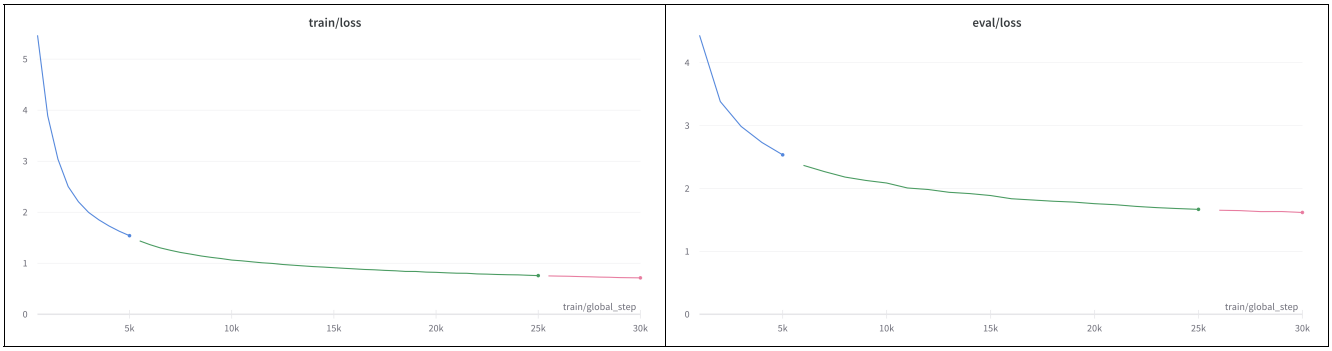
Figure 4.3: CodeBERT Fine-tuning.

4.7 ConvBERT

The authors of *ConvBERT* propose a new span-based dynamic convolution to replace self-attention heads in order to more efficiently local model dependencies. Convolution heads, together with the remaining self-attention heads, form a new mixed attention block that is more efficient at both global and local context learning. The paper demonstrates that the *ConvBERT* model outperforms BERT and its variants in various downstream tasks, with lower training cost and fewer model parameters [46].

4.7.1 Training

The above-described features of this architecture allowed us to increase the batch size and the embedding length to 1024, so due to the presence of convolution and a smaller number of network parameters, the model learns much faster.



(a) Train Loss (b) Validation Loss

Figure 4.4: ConvBERT Training (Kernel Size = 7, Batch Size = 32, Gradient Accumulation = 8).

This architecture is not performing at its best because most of the information in the dataset is not being used, and there are only a small number of training iterations. However, it has great potential to handle large datasets.

In further research, it would be interesting to compare it also with Roberta, which we use to work with AST in Section 4.9.3.

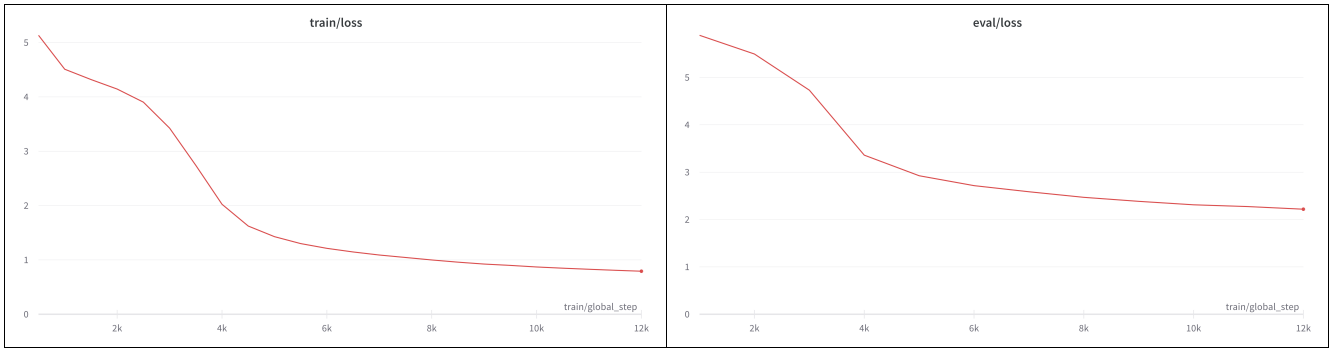
4.8 Longformer

Longformer is a transformer model with an attention mechanism that scales linearly with sequence length, making it easy to process documents of thousands of tokens or longer [47]. It employs a hybrid of local (sliding window) and global attention mechanisms, with the global attention layer being user-configurable according to the desired task in order to enable the model to learn task-specific representations.

The same dataset as before will be used, tokenized with a maximum length of 1024, so global attention is not used. As will be discussed further in Section 4.9.5, such a restriction can become problematic. But retraining the tokenizer and the tokenization of the whole dataset in our case is a task for which there are insufficient resources, which was the critical factor.

4.8.1 Training

Training such a model from scratch did not give super stunning results, and after comparing its results with other models, it was decided not to train this model to the end. The main assumption is that if we use the maximum possible length of the inputs, it will help to increase the context in both directions and improve the model's quality.

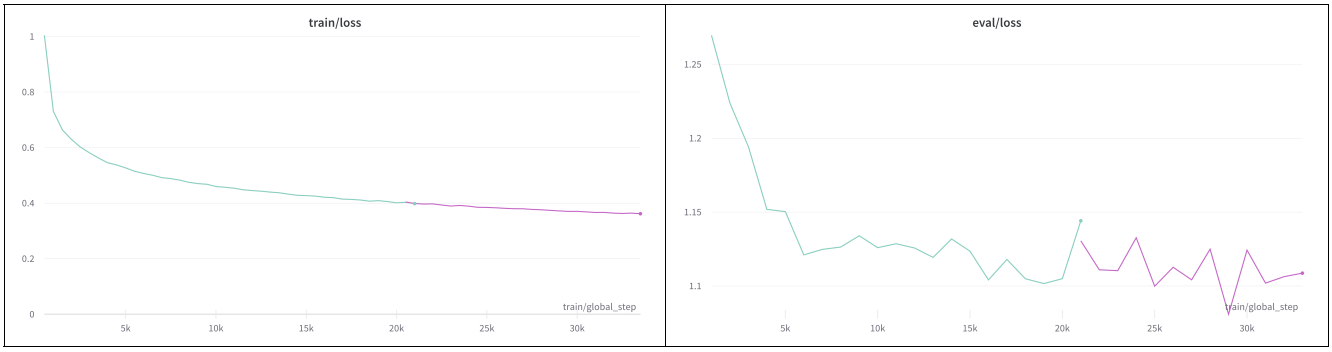


(a) Train Loss (b) Validation Loss

Figure 4.5: Longformer Training.

4.8.2 Fine-tuning

The model chosen for the finetuning was the one that was pre-trained by the original article's creators.



(a) Train Loss (b) Validation Loss

Figure 4.6: Longformer Fine-tuning.

A very interesting observation (Figure 4.5b, 4.6b) is that a model pre--trained on plain text performed better with code than a model that was trained on code from scratch.

One hypothesis is that a dataset for natural language work is ideal with respect to the very large amount of data and its variability. But not so ideal for this problem because a further error on validation may not decrease much, based on the comparison of loss on validation and the test.

4.9 R AST Transformer

The model was inspired by the *n-gram* model of Nguyen et al., which is the *n-gram* model that was extended by annotating code tokens with parse information that can be extracted from the sequence in order to increase the available context information [48] and UniXcoder [49] - a Transformer model that uses data from multiple sources (code comments and ASTs) to improve the accuracy of code representations.

The idea is to represent the information from AST in token sequencing format. This will allow the model to predict not only the next token but also semantic information and type.

4.9.1 AST-Sequence Mapping

The mapping function F maps the AST to a token sequence in a pre-order traversal way, adding a special token after the last leaf in each path.

F^{-1} is a reverse function that takes an AST token sequence as input and outputs the corresponding code or semantic information. It's very helpful

when reconstructing the AST with a newly predicted token (code token or semantic information).

4.9.2 Tokenizer

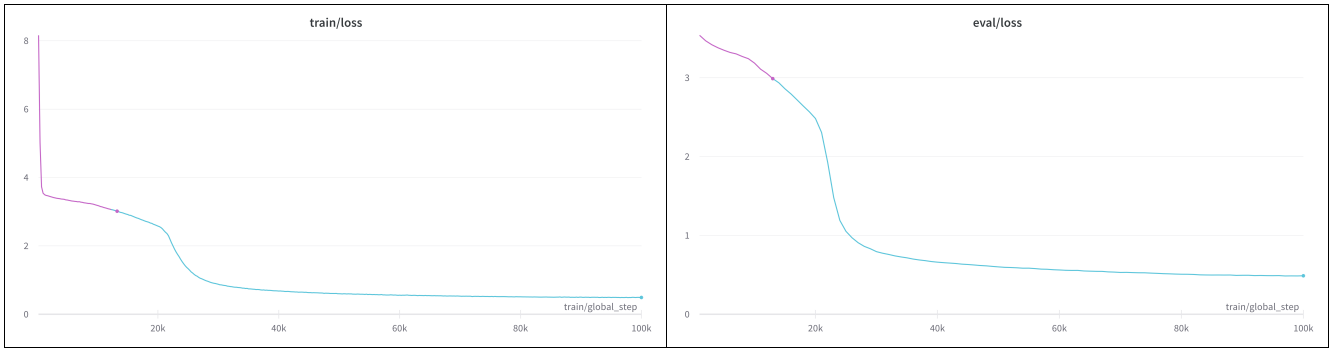
For this task, it was necessary to train the Byte-Level BPE. The tokenizer with a dictionary size of 50k was chosen, and the other parameters were kept the same.

The significant point is that the newly-trained tokenizer has learned our AST format. It is capable of identifying special tokens (`<LEND>`, `<LSEP>` which are used to mark the last leaf in paths and divide the nodes in our sequences) as one complete token. It is also able to detect brackets (and all possible combinations of them when there is no information in a specific position), which are used for a convenient format of parsing information from AST, for example `){`, `](`, `](){`, `](){}`, etc. This saves space for other more useful tokens in the model's input.

4.9.3 Training

We chose RoBERTa Architecture as the basis for our model based on the experience of *CodeBERT*.

Despite the fact that the dataset for validation is different from the previously used because AST sequencing is used here, the result is quite comparable, and it is much better compared to the fine-tuned CodeBERT that had the best result until then.



(a) Train Loss (b) Validation Loss

Figure 4.7: RASTaBERTa Training.

4.9.4 Crashtest

First, let's try to determine what the model is capable of and try it out with a few examples.

The following code snippet was obtained from the validation dataset.

```
library(ggplot2)
library(readr)

system("ls ../input")

data <- read.table(header=TRUE, text='
 subject sex size
       1   M    7
       2   F   NA
       3   F    9
       4   M   11
 ')

write.csv(data, "output.csv", row.names=FALSE)
```

Listing 4.2: Validation Set Sample.

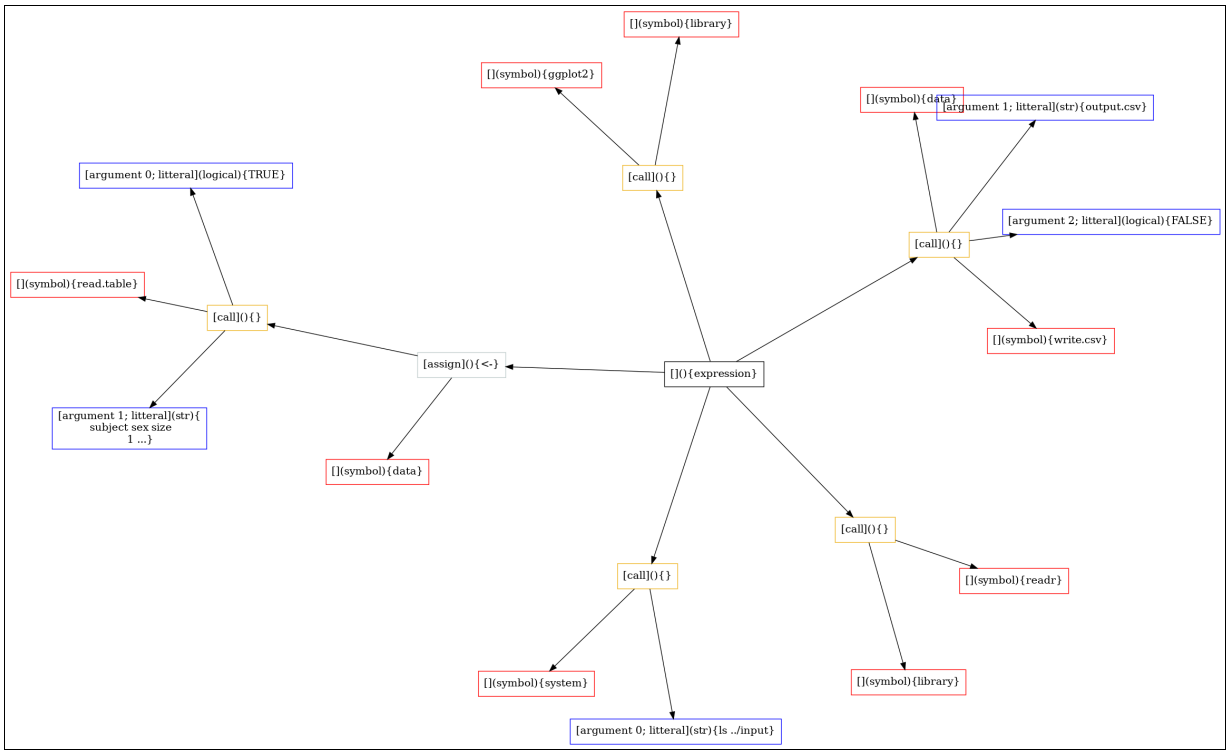


Figure 4.8: AST For R Code Sample.

The AST for this snippet was extracted in the same way as in the Section 4.2.3

This graph has a node with the following information: `[argument 2; litteral](logical){FALSE}`. The tokens we are interested in, are replaced with `<mask>`: `[<mask> <mask>; litteral](<mask>){FALSE}`.

Mask	Predicted Value	Probability
0	argument	0.9997
	litteral	0.0001
	false	0.0000
1	2	0.9790
	3	0.0116
	1	0.0043
2	logical	0.9939
	str	0.0060
	real	0.0001

Table 4.4: Prediction of semantic info and value type for boolean value.

With high accuracy, the model predicted the exact result of the semantic information and the type of this argument, even though neighboring tokens have also been masked.

The next example is a bit more complicated. Can the model understand from context the type of file extension into which the information will be written?

Raw input: `[argument 1; litteral](str){output.csv}`.

Masked input: `[argument 1; litteral](str){output.<mask>}`.

Mask	Predicted Value	Probability
0	csv	0.2166
	table	0.1700
	data	0.0632

Table 4.5: Prediction of filename extension - *AST RoBERTa*.

The predicted answer is correct, albeit with less confidence.

Now let's compare the result of the best pure source code model on the same task (prediction of `csv` extension), fine-tuned CodeBERT in our case.

Mask	Predicted Value	Probability
0	`unknown symbol`	0.2352
	data	0.1073
	turn	0.0407

Table 4.6: Prediction of filename extension - *Fine-tuned CodeBERT*.

The correct result is not even close in the model's field of view. Now let's look at another more familiar example (Listing 4.3)

```
recur_factorial <- function(n) {
    if  (n <= 1) {
        return(1)
    } else {
        return(n * recur_factorial(n - 1))
    }
}
```

Listing 4.3: Factorial via Recursion Function.

Both models have difficulty determining the assignment sign, but the AST model says with a full probability that the semantics of this token is assigned.

Mask	Predicted Value	Probability
0	n	0.4051
	print	0.0126
	r	0.0114

Table 4.7: Prediction of function name.

Mask	Predicted Value	Probability
0	function	0.9999
	n	0.0000
	call	0.0000

Table 4.8: Prediction of `function` keyword.

Mask	Predicted Value	Probability
0	n	0.3329
	sum	0.1751
	sqrt	0.0926

Table 4.9: Prediction of the recurrence call `recur_factorial`.

Mask	Predicted Value	Probability
0	n	0.8809
	k	0.0141
	r	0.0124

Table 4.10: Prediction of `n` in condition statement `n <= 1`.

Mask	Predicted Value	Probability
0	return	0.2207
	stop	0.2120
	n	0.1428
1	return	0.3483
	n	0.2621
	stopifnot	0.0574

Table 4.11: Prediction of both `return`.

After all the experiments, it turned out that this model predicts semantic information (Section 4.2.3), types, and keywords reserved by R with great accuracy. Problems mainly appear in cases of special characters, functions, and variable declarations.

The author believes that such neural networks can help not only in the area of smart type hinting and auto-completion but also in improving the performance of compiler optimizers.

4.9.5 Split-Augmentation Dataset

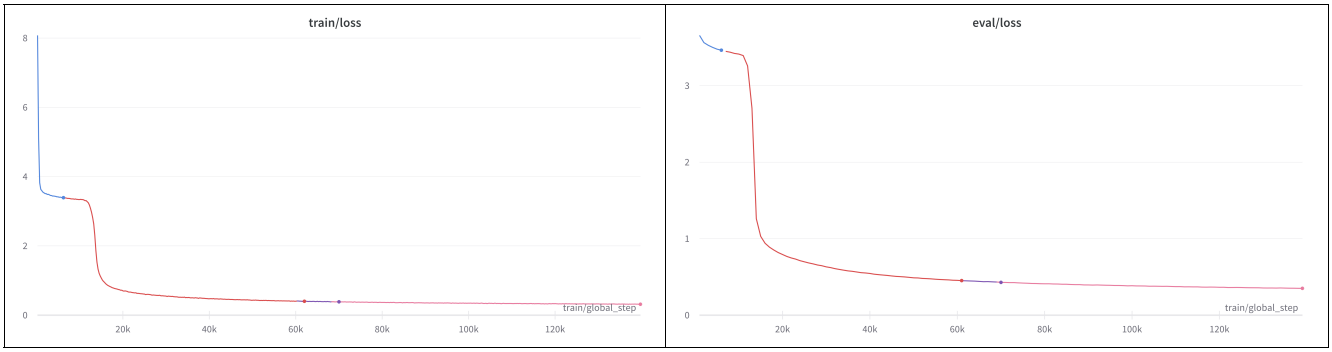
All of our models are limited in the length of the model's input. Previously we used vector lengths of 512 and 1024. 83.5% of all code files are longer than 1024, so much of the information is just truncated and lost in the tokenization phase. The idea of splitting sentences and using a sliding window arose, so that context would not be completely lost.

As a result of this simple augmented modification, the number of entries in the training dataset increased from 367326 to 11867395. This is a big increase of 32 times, which should improve the results of the model, but it will also increase its learning time.

Tokenization of such a large amount of data has taken 8 hours.

4.9.6 Training on Augmented Dataset

Comparing the graphs of the two models, it is clear that the improved model learns faster. It also has improved results in assignment character detection and other cases. This model shows the best results in token mask prediction compared to all tested transformer models.



(a) Train Loss (b) Validation Loss

Figure 4.9: RASTaBERTa Training On Augmented Data.

We introduced and named this model RASTaBERTa. For testing it, the demo has been prepared, which will be attached to this paper. This demo also includes code-to-ast transformation (our specific AST sequence format).

Model	**LR Scheduler**	**#Parameters**	**Batch Size**	**Grad Accumulation**	**Max Size**	**Tokenizer Vocab Size**	**#iter**	**Train Loss**	**Val Loss**
BERT Training	Linear	109.5M	8	1	512	30k	238k	0.79	1.71
CodeBERT Fine-tuning (Weight Decay 0.01 + Dropout 0.2)	Linear with Warmup	124.6M	24	8	512	50k	22k	0.32	0.91
CodeBERT Fine-tuning (Weight Decay 0.01 + Dropout 0.2)	Cosine with Warmup	124.6M	24	8	512	50k	14.25k	0.35	0.89
CodeBERT Fine-tuning (#Hidden Layers = 6)	Linear	82.2M	24	8	512	50k	29k	0.44	1.26
Longformer Training	Linear	148.7M	24	8	4098	50k	12k	0.71	2.21
Longformer Fine-tuning	Linear	148.7M	24	8	4098	50k	33.5k	0.36	1.1
ConvBERT Training (Conv Kernel Size = 7)	Linear	106.6M	8	1	1024	30k	92k	1.04	2.06
ConvBERT Training (Conv Kernel Size = 7)	Linear	106.6M	32	8	1024	30k	30k	0.71	1.61
ConvBERT Training (#Attention Heads = 8 , #Hidden Layers = 6 , Conv Kernel Size = 9)	Linear	65.3M	24	8	1024	30k	29k	0.85	1.78
R-AST-Roberta (Weight Decay 0.01)	Linear with Warmup	125M	10	4	1024	50k	100k	0.49	0.48
R-AST-Roberta (Weight Decay 0.01)	Linear with Warmup	125M	10	4	1024	50k	150k	0.31	0.34

Table 4.12: Transformer Experiments Parameters and Results.

4.10 Embeddings

In these experiments, the *K-Means* [50] algorithm is used for clustering, and the elbow method is used to find the optimal number of clusters.

For the visualization, it is necessary to reduce the number of dimensions. For this purpose *PCA* [50] (Principal Component Analysis) and *UMAP* [51] (Uniform Manifold Approximation and Projection for Dimension Reduction) will be used.

Several different techniques applicable to the task will be tested in the following sections.

Embeddings (10 for each model) for the manual evaluation and review were chosen randomly from the dataset.

4.10.1 doc2vec Embeddings

This model was trained on R Dataset in order to vectorize the internal graph information, such as value tokens. Also, we can compare the result of *doc2vec* with the embeddings of other models.

The optimal number of epochs was chosen from an original paper suggestion and equaled 20, other parameters such as embedding size and window size are equal to 300 and 10, respectively.

The training of the model was performed on the *Apple M1* CPU with eight cores and lasted 10 hours.

Figure 4.10: K-Means Clustering on doc2vec embeddings (PCA).

These embeddings and their similarity take almost no account of semantics, and their similarity is mainly influenced only by the presence of a strictly defined token order.

4.10.2 Transformer Embeddings

The output of these models contains high-dimensional context embeddings that take up a lot of space. In order to store all of the proposed embeddings, terabytes of physical memory would be required. Clustering algorithms struggle to handle embeddings that are very large. For example, fine-tuned CodeBERT embedding has the shape `[1, 512, 50265]`.

To reduce the dimensionality without losing all the information, the *Mean Pooling* operation was applied. As a result, the dimensionality has decreased to `[1, 512]`.

There is also a problem with files longer than the tokenizer max size of the input. In this case, it is possible to split the code into chunks of 1024 length and some stride (similarly to the case of an augmented dataset) and then get one embedding, for example, by using mean pooling. This paper does not address that due to the limitations of the hardware.

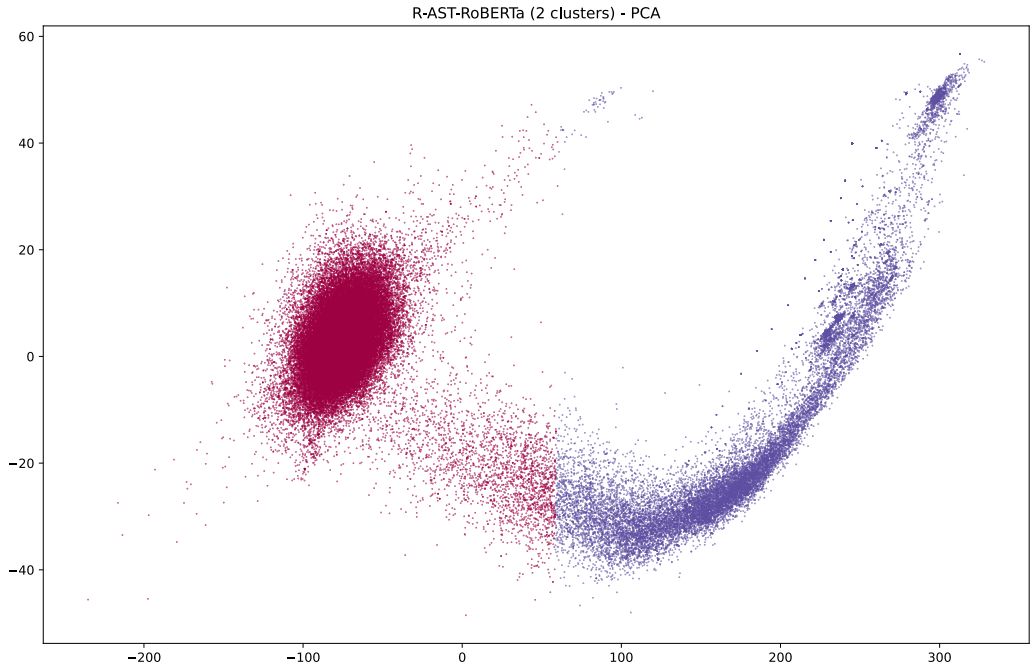


Figure 4.11: K-Means Clustering on RASTaBERTa embeddings (PCA).

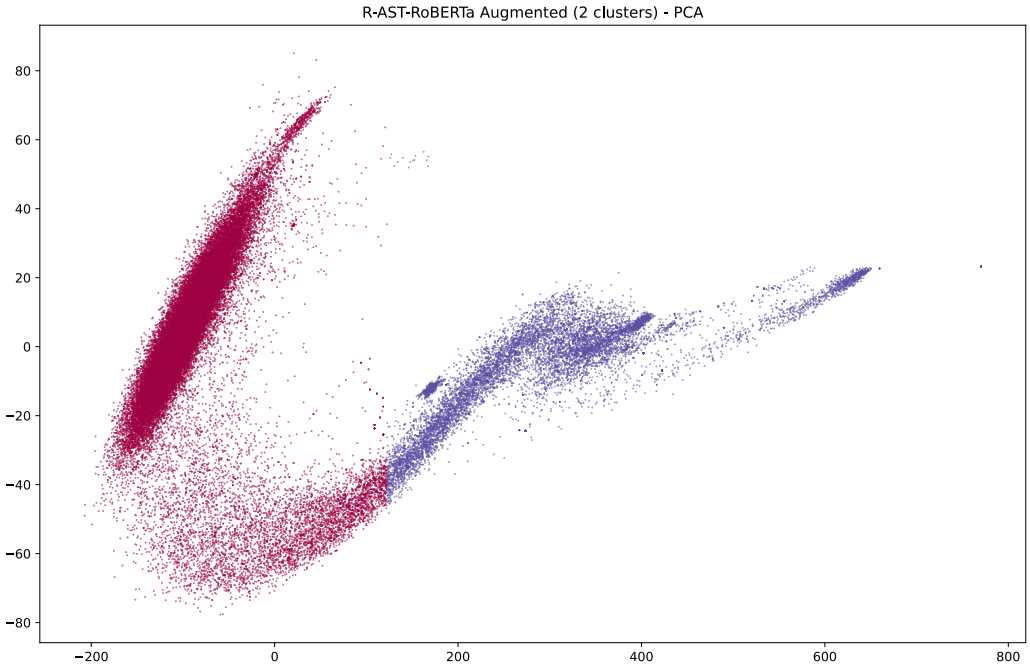


Figure 4.12: K-Means Clustering on RASTaBERTa embeddings (PCA).

The results of the two models trained on datasets with different types of data do not differ very much. Embeddings were calculated for roughly 20% of the training set. To speed up K-means in the case of a model trained on an augmented dataset (Figure 4.12), PCA was also applied to reduce the vector length to 512. Experiments with normalization and standardization did not lead to improved results and great acceleration.

This type of embedding is comparably better than *doc2vec*, and it takes into account the structure of the code and its semantics. Such embeddings can be extremely useful in the search for plagiarism and clone detection. Unfortunately, these embeddings can only be accurately assessed with either manually labeled data or by hand.

The success of this model lies in the increased context provided by AST. This can be compared to models trained in typed languages such as C and Java, which usually have better benchmarks than interpreted languages. Types and additional context simplify network learning.

4.10.3 Graph Embeddings

Due to the limited RAM, the number of processed graphs was limited to 18K. Also, due to hardware limitations, only two graph methods will be tested.

4.10.3.1 Feather

The algorithm uses the characteristic functions of node features with random walk weights to describe node neighborhoods. These node-level features are then pooled together by mean pooling to create graph-level embeddings [11]. Therefore, this algorithm does not use the internal features (e.g., manually parsed semantic information and values of the graph nodes).

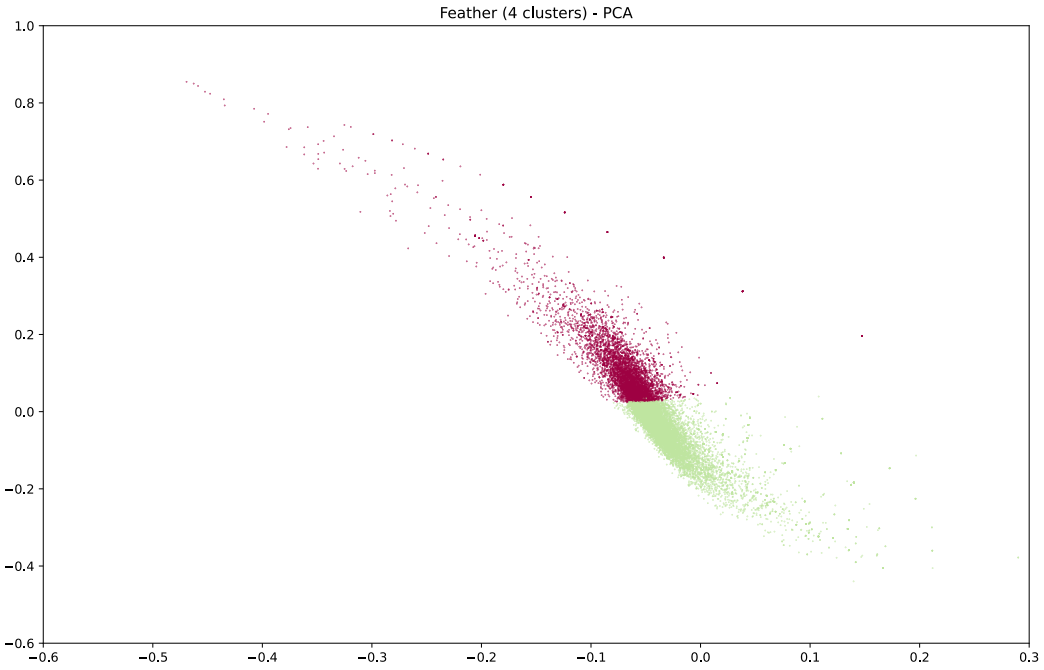


Figure 4.13: K-Means Clustering on Feather embeddings (PCA).

In this case, the result was also tested with UMAP, but this also did not yield a great improvement.

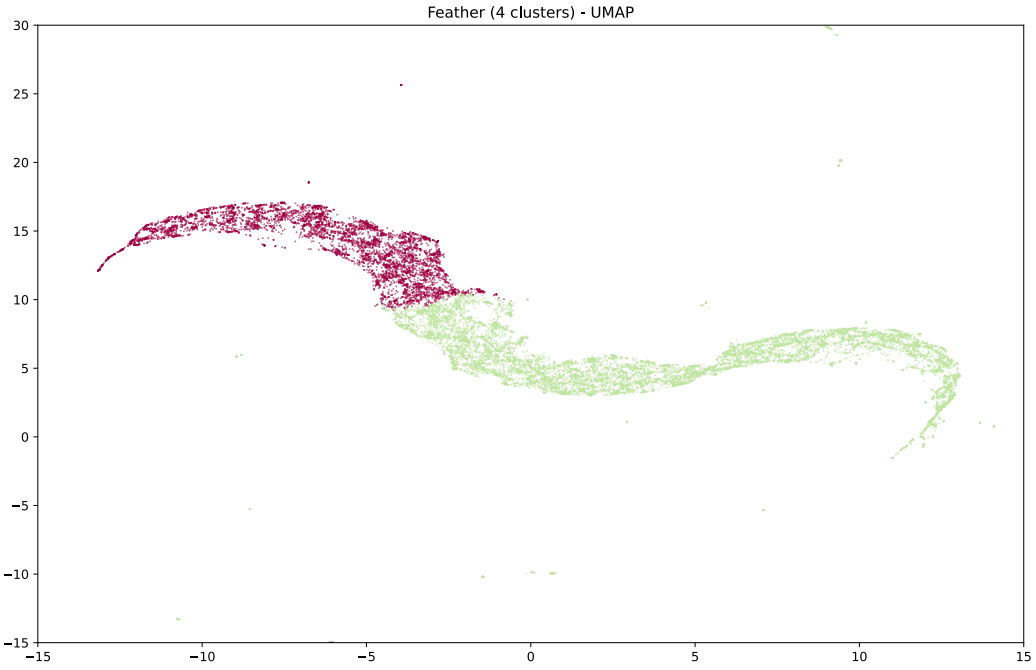


Figure 4.14: K-Means Clustering on Feather embeddings (UMAP).

Despite the fact that the elbow method gave a result of 4, both figures clearly show the division into 2 clusters.

According to our estimation, the code size has a very large influence on this model - graphs with similar length and structure have similar embeddings. Also, only the structure of the code graph without code tokens makes the embeddings whose code is very different close in this space.

4.10.3.2 GL2Vec

The algorithm first creates the line graph of each graph in the dataset. Next, it uses the *Weisfeiler-Lehman* tree features to create representations for the nodes in the graphs. Finally, it uses these representations to generate a co-occurrence matrix for the graphs. This matrix is the desired embedding. *GL2Vec* model uses internal features of the nodes in the graph [52].

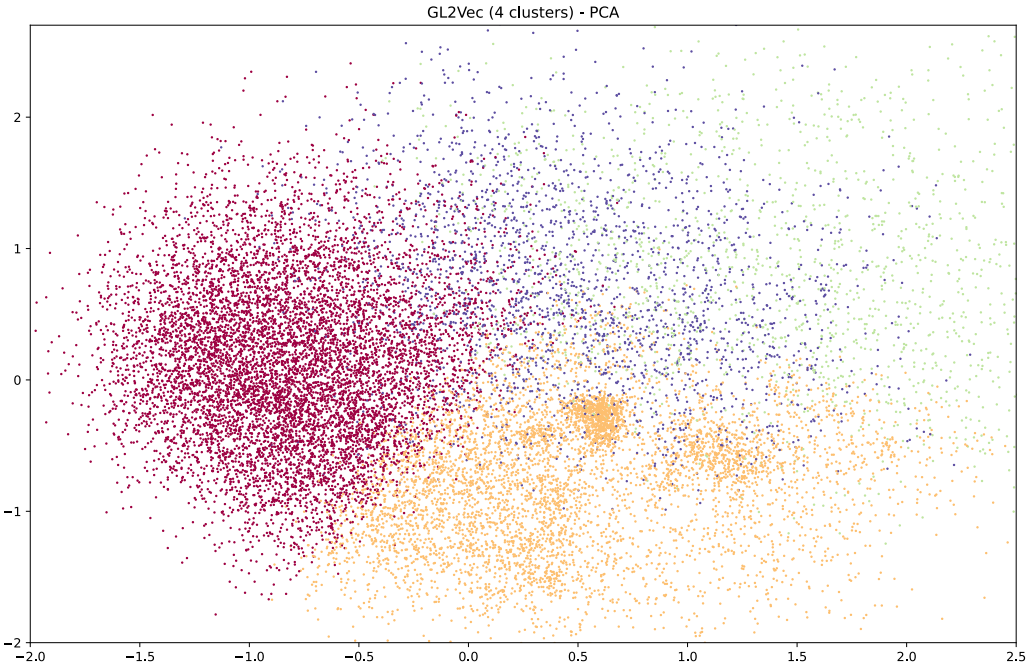


Figure 4.15: K-Means Clustering on GL2Vec embeddings (PCA).

In this space, it was not possible to find a clear distinction between clusters. Therefore, it is impossible to clearly state that there are 4 clusters.

This type of embedding is considered a middle ground between *doc2vec* and *Feather*, since it takes into account both the topology of the graph and the encoded information inside. In the matter of semantics, transformer embeddings proved to be better.

4.11 Hardware Constraints and Future Work

It takes a lot of computing power and memory to handle a large number of vectors with a large number of dimensions. This was a major stumbling block that prevented the proposed model from reaching its full potential.

In the case of graphs, the problem of hardware limitation is even worse because it is a more complex data structure that takes up more space and is more difficult to process. This prevented testing the most efficient graph

models. But due to the fact that we can draw a parallel between Transformer and GNN (Section 3.4.3), it can be interpreted as success.

Further work will be aimed at improving methods of preprocessing code and its representation (for example, using code paths, as in the case of *code2vec*), training models of other transformer architectures, such as ConvBERT and the application of these models in practice.

Chapter 5

Conclusion

As a result, we analyzed and passed through the complexities that can arise when working with the code. Different ways of representation and their pros and cons were reviewed. We used different techniques when training models to stabilize learning. A split augmentation technique with a sliding window was also proposed to increase context and reduce the impact of the model and tokenizer input limits. Many models have been implemented and tested, and obstacles associated with them were found and described. We examined why certain techniques are not suitable for source code or are no longer used with the emergence of new architectures. We have found that transformers are outperforming models with other architectures. We have trained the best transformer model so far that works with the R language and prepared a dataset for further research.

In summary, in this work the author presented:

- R Code and R AST Datasets
- Fine-tuned CodeBERT model for R
- Pre-trained RoBERTa-based model and tokenizer - *RASTaBERTa*
- Dataset Split-Augmentation Technique
- *RASTaBERTa* Application Demo

Bibliography

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, ``A survey of machine learning for big code and naturalness'', *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[2] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, ``Modeling and discovering vulnerabilities with code property graphs'', in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604. DOI: 10.1109/SP.2014.44.

[3] J. Ferrante, K. J. Ottenstein, and J. D. Warren, ``The program dependence graph and its use in optimization'', *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, 319–349, 1987, ISSN: 0164-0925. DOI: 10.1145/24039.24041. [Online]. Available: https://doi.org/10.1145/24039.24041.

[4] F. E. Allen, ``Control flow analysis'', in *Proceedings of a Symposium on Compiler Optimization*, Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, 1–19, ISBN: 9781450373869. DOI: 10.1145/800028.808479. [Online]. Available: https://doi.org/10.1145/800028.808479.

[5] R. Gold, ``Control flow graphs and code coverage'', *International Journal of Applied Mathematics and Computer Science*, vol. 20, no. 4, pp. 739–749, 2010. DOI: doi:10.2478/v10006-010-0056-9. [Online]. Available: https://doi.org/10.2478/v10006-010-0056-9.

[6] Z. S. Harris, ``Distributional structure'', *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[7] T. Mikolov, K. Chen, G. Corrado, and J. Dean, ``Efficient estimation of word representations in vector space'', *arXiv preprint arXiv:1301.3781*, 2013.

[8] P. F. Brown, V. J. Della Pietra, P. V. Desouza, J. C. Lai, and R. L. Mercer, ``Class-based n-gram models of natural language'', *Computational linguistics*, vol. 18, no. 4, pp. 467–480, 1992.

[9] P. A. Gagniuc, *Markov chains: from theory to implementation and experimentation*. John Wiley & Sons, 2017.

[10] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, ``Graph2vec: Learning distributed representations of graphs'', *arXiv preprint arXiv:1707.05005*, 2017.

[11] B. Rozemberczki and R. Sarkar, ``Characteristic functions on graphs: Birds of a feather, from statistical descriptors to parametric models'', ser. CIKM '20, Virtual Event, Ireland: Association for Computing Machinery, 2020, ISBN: 9781450368599. DOI: 10.1145/3340531.3411866. [Online]. Available: https://doi.org/10.1145/3340531.3411866.

[12] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P.d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, ``Evaluating large language models trained on code'', *arXiv preprint arXiv:2107.03374*, 2021.

[13] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang, M. Pieler, U. S. Prashanth, S. Purohit, L. Reynolds, J. Tow, B. Wang, and S. Weinbach, ``GPT-NeoX-20B: An open-source autoregressive language model'', 2022.

[14] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, ``A systematic evaluation of large language models of code'', *arXiv preprint arXiv:2202.13169*, 2022.

[15] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, *Competition-level code generation with alphacode*, 2022.

[16] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, ``Code2vec: Learning distributed representations of code'', *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 2019. DOI: 10.1145/3290353. [Online]. Available: https://doi.org/10.1145/3290353.

[17] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, ``Learning phrase representations using rnn encoder-decoder for statistical machine translation'', *arXiv preprint arXiv:1406.1078*, 2014.

[18] R. Hecht-Nielsen, ``Theory of the backpropagation neural network'', in *Neural networks for perception*, Elsevier, 1992, pp. 65–93.

[19] P. J. Werbos, ``Backpropagation through time: What it does and how to do it'', *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[20] S. Hochreiter and J. Schmidhuber, ``Long short-term memory'', *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[21] R. Pascanu, T. Mikolov, and Y. Bengio, ``On the difficulty of training recurrent neural networks'', in *International conference on machine learning*, PMLR, 2013, pp. 1310–1318.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, ``Attention is all you need'', *Advances in neural information processing systems*, vol. 30, 2017.

[23] C. Joshi, ``Transformers are graph neural networks'', *The Gradient*, 2020.

[24] 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.

[25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, pp. 499–523, `http://www.deeplearningbook.org`.

[26] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, *Codebert: A pre-trained model for programming and natural languages*, 2020. arXiv: `2002.08155 [cs.CL]`.

[27] Q. Le and T. Mikolov, ``Distributed representations of sentences and documents'', in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ser. ICML'14, Beijing, China: JMLR.org, 2014.

[28] P. S. Aric Hagber Dan Schult, *Networkx*, `https://styler.r-lib.org`, 2021.

[29] B. Rozemberczki, O. Kiss, and R. Sarkar, ``Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs'', in *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*, ACM, 2020, 3125–3132.

[30] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, ``Array programming with NumPy'', *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[31] T. pandas development team, *Pandas-dev/pandas: Pandas*, version latest, Feb. 2020. DOI: 10.5281/zenodo.3509134. [Online]. Available: https://doi.org/10.5281/zenodo.3509134.

[32] J. D. Hunter, ``Matplotlib: A 2d graphics environment'', *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, ``Scikit-learn: Machine learning in Python'', *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[34] *Weights and biases*. [Online]. Available: https://wandb.ai/site.

[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, ``Pytorch: An imperative style, high-performance deep learning library'', in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d"0027 Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[36] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S.

Gugger, M. Drame, Q. Lhoest, and A. M. Rush, ``Transformers: State-of-the-art natural language processing'', in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[37] R. Rehurek and P. Sojka, ``Gensim--python framework for vector space modelling'', *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, vol. 3, no. 2, 2011.

[38] L. McInnes, J. Healy, N. Saul, and L. Grossberger, ``Umap: Uniform manifold approximation and projection'', *The Journal of Open Source Software*, vol. 3, no. 29, p. 861, 2018.

[39] A. Goel, P. Donat-Bouillud, C. Kirsch, and J. Vitek, ``Why We Eval in the Shadows'', in *2021 PACMPL Proceedings of the ACM on Programming Languages*, ACM, 2021.

[40] P. Donat-Bouillud, *asttoolsr*, version 1.0.0, Apr. 2022. [Online]. Available: https://github.com/programLyrique/asttoolsr.

[41] C. Data61, *Stellargraph machine learning library*, https://github.com/stellargraph/stellargraph, 2018.

[42] 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.

[43] V. Cohen and A. Gokaslan, ``Opengpt-2: Open language models and implications of generated text'', *XRDS*, vol. 27, no. 1, pp. 26–30, 2020, ISSN: 1528-4972. DOI: 10.1145/3416063. [Online]. Available: https://doi.org/10.1145/3416063.

[44] X. Song, A. Salcianu, Y. Song, D. Dopson, and D. Zhou, ``Fast wordpiece tokenization'', *arXiv preprint arXiv:2012.15524*, 2020.

[45] T. Kudo, ``Subword regularization: Improving neural network translation models with multiple subword candidates'', *arXiv preprint arXiv:1804.10959*, 2018.

[46] Z.-H. Jiang, W. Yu, D. Zhou, Y. Chen, J. Feng, and S. Yan, ``Convbert: Improving bert with span-based dynamic convolution'', in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 12 837–12 848. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/96da2f590cd7246bbde0051047b0d6f7-Paper.pdf.

[47] I. Beltagy, M. E. Peters, and A. Cohan, ``Longformer: The long-document transformer'', *arXiv:2004.05150*, 2020.

[48] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, ``A statistical semantic language model for source code'', in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 532–542.

[49] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, ``Unixcoder: Unified cross-modal pre-training for code representation'', *arXiv preprint arXiv:2203.03850*, 2022.

[50] C. Ding and X. He, ``K-means clustering via principal component analysis'', in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 29.

[51] L. McInnes, J. Healy, and J. Melville, ``Umap: Uniform manifold approximation and projection for dimension reduction'', 2020.

[52] H. Chen and H. Koga, ``Gl2vec: Graph embedding enriched by line graphs with edge features'', in *International Conference on Neural Information Processing*, Springer, 2019, pp. 3–14.

Chapter 6

Supplement Structure

- `README.md` ..... description
- `src` ..... source code
  - `R` ..... source code and data related to R
  - `experiments-on-cluster` ..... code executed on cluster
  - `latex` ..... the directory of LaTeX source codes of the thesis
- `thesis.pdf` ..... the thesis text in PDF format