DEGREE PROJECT IN TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

Transformer-based Multistage Architectures for Code Search

KTH Thesis Report

Angel L. González

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Authors

Angel L González <algon@kth.se>
Information and Communication Technology
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden
RISE Research Institutes of Sweden

Examiner

Amir H. Payberah, KTH Royal Institute of Technology

Supervisors

Francisco J. Peña, KTH Royal Institute of Technology

Sepideh Pashami and Ahmad Al-Shishtawy, RISE Research Institutes of Sweden

Abstract

Code Search is one of the most common tasks for developers. The open-source software movement and the rise of social media have made this process easier thanks to the vast public software repositories available to everyone and the Q&A sites where individuals can resolve their doubts. However, in the case of poorly documented code that is difficult to search in a repository, or in the case of private enterprise frameworks that are not publicly available, so there is not a community on Q&A sites to answer questions, searching for code snippets to solve doubts or learn how to use an API becomes very complicated. In order to solve this problem, this thesis studies the use of natural language in code retrieval. In particular, it studies transformer-based models, such as Bidirectional Encoder Representations from Transformers (BERT), which are currently state of the art in natural language processing but present high latency in information retrieval tasks. That is why this project proposes a multi-stage architecture that seeks to maintain the performance of standard BERT-based models while reducing the high latency usually associated with the use of this type of framework. Experiments show that this architecture outperforms previous non-BERT-based models by +0.17 on the Top 1 (or Recall@1) metric and reduces latency with inference times 5% of those of standard BERT models.

Keywords

Code Search, Natural Language Processing, BERT, Information Retrieval

Abstract

Kodsökning är en av de vanligaste uppgifterna för utvecklare. Rörelsen för öppen källkod och de sociala medierna har gjort denna process enklare tack vare de stora offentliga programvaruupplagorna som är tillgängliga för alla och de Q&A-webbplatser där enskilda personer kan lösa sina tvivel. När det gäller dåligt dokumenterad kod som är svår att söka i ett arkiv, eller när det gäller ramverk för privata företag som inte är offentligt tillgängliga, så att det inte finns någon gemenskap på Q&AA-webbplatser för att besvara frågor, blir det dock mycket komplicerat att söka efter kodstycken för att lösa tvivel eller lära sig hur man använder ett API. För att lösa detta problem studeras i denna avhandling användningen av naturligt språk för att hitta kod. I synnerhet studeras transformatorbaserade modeller, såsom BERT, som för närvarande är den senaste tekniken inom behandling av naturliga språk men som har hög latenstid vid informationssökning. Därför föreslås i detta projekt en arkitektur i flera steg som syftar till att bibehålla prestandan hos standard BERT-baserade modeller samtidigt som den höga latenstiden som vanligtvis är förknippad med användningen av denna typ av ramverk minskas. Experiment visar att denna arkitektur överträffar tidigare icke-BERT-baserade modeller med +0,17 på Top 1 (eller Recall@1) och minskar latensen, med en inferenstid som är 5% av den för standard BERT-modeller.

Nyckelord

Kodsökning, behandling av naturligt språk, BERT, informationssökning

Acknowledgements

This project has been carried out in collaboration with the Software and Computer Systems (SCS) department of the KTH Royal Institute of Technology and the RISE Research Institutes of Sweden.

I would like to express my gratitude to my examiner, Amir H. Payberah, for his guidance, patience and lessons, and to my supervisor, Francisco J. Peña, for his support, help and advice that helped me to become a better researcher.

I would also like to thank my RISE supervisors Sepideh Pashami and Ahmad Al-Shishtawy, for their help, advice and support throughout the project.

I express my sincere thanks to the EIT community, especially to my colleagues in Madrid and Stockholm, by far the best of this master.

I also want to thank my family and friends, that gave me all the support I needed to carry out this adventure.

Acronyms

DL Deep Learning
ML Machine Learning
NLP Natural Language Processing
DCS Deep Code Search
NCS Neural Code Search
BERT Bidirectional Encoder Representations from Transformers
API Application Programming Interface

Contents

1 Introduction **2**

1.1 Background 2
1.2 Problem 2
1.3 Research Question and Contributions 3
1.4 Outline 3

2 Theoretical Background **4**

2.1 Machine Learning and Deep Learning 4
2.1.1 Siamese Neural Networks 5
2.1.2 Embeddings 5
2.1.3 Attention mechanism and Transformers 6
2.1.4 BERT and Sentence-BERT 8
2.2 Code Search 9
2.3 Related work 10

3 Research Methodology **11**

3.1 Models 11
3.1.1 UNIF 11
3.1.2 UNIF SNN 12
3.1.3 Sentence-BERT and monoBERT 13
3.1.4 Multistage architecture 13
3.2 Dataset 14
3.3 Training 15
3.3.1 Triplet network framework 15
3.3.2 MonoBERT training 16
3.4 Tests 17
3.4.1 Metrics 17
3.4.2 Rephrasing test 17
3.4.3 Search examples 18

4 Experiments and Results **19**

4.1 Experiments 19
4.2 Code Search examples 21

5 Conclusions and Future Work **29**

5.1 Conclusions 29
5.1.1 Future Work 29

References **31**

List of Figures

1.2.1 Example of a question on StackOverflow and GitHub 3

2.1.1 Neural network example 5
2.1.2 Siamese neural network for signature forgeries detection. 6
2.1.3 Self-attention mechanism 7
2.1.4 Transformer encoder architecture 8
2.1.5 BERT vs Sentence-BERT 9

3.1.1 UNIF model 12
3.1.2 UNIF SNN model 13
3.1.3 Two stages architecture. 14
3.3.1 Triplet networks architecture. 15
3.3.2 MonoBERT architecture. 16

List of Tables

3.2.1 Deep Code Search dataset summary. 14
3.3.1 MonoBERT dataset format 16
3.4.1 Top-N example 17

4.1.1 Main results 20
4.1.2 Rephrasing results 21
4.1.3 Rephrased examples 21
4.2.1 Search example 1 22
4.2.2 Search example 2 23
4.2.3 Search example 3 24
4.2.4 Search example 4 25
4.2.5 Search example 5 26
4.2.6 Search example 6 27
4.2.7 Search example 7 28

Chapter 1

Introduction

1.1 Background

A developer's workday is much more than coding. There are numerous studies [25, 28, 30] on the wide variety of tasks that developers perform, such as reading and writing documentation, reviewing code, and debugging. Among all of them, code search, either to understand what a program does, to find out why it has behaved in a particular way or to learn how to use a framework, among other reasons [32], is one of the most frequent. Back in 1997, it was already one of the most common tasks in a developer's day-to-day life [33]. At this time, searching for code was done with basic tools such as the command line utility *grep*, or the editor's or company's search engines and the only way to resolve more complex questions was to consult with colleagues.

Fortunately, software development has evolved a lot since then, providing programmers with tools that make many tasks like this one easier and faster. Firstly, the rise of software engineering social media has led to the emergence of Q&A sites where developers solve their doubts and share their knowledge, being Stack Overflow [20] the most popular of them. Secondly, the open source software movement has given rise to huge repositories of code available to consult for examples of Application Programming Interface (API) usage, frameworks and algorithm implementations among other reasons.

1.2 Problem

Consider the following query *"How do I create a file and write to it?"*. As shown in Figure 1.2.1, the first answer for this question in Stack Overflow contains a small and specific snippet that solves the problem, but the same search on GitHub leads to an file that contains some of the words from the question but does not solve the problem. Therefore, a user could solve a programming question on a Q&A site if he or she can formulate it in a similar way to the title of an existing question. However, in order to do so in a code repository, he or she will have to use words present in the code or in its comments. The problem here is to be able to find the corresponding answer when the question does not match the keywords in the code fragment.

Code search is a problem that has been approached as a text retrieval task, where an input query is used

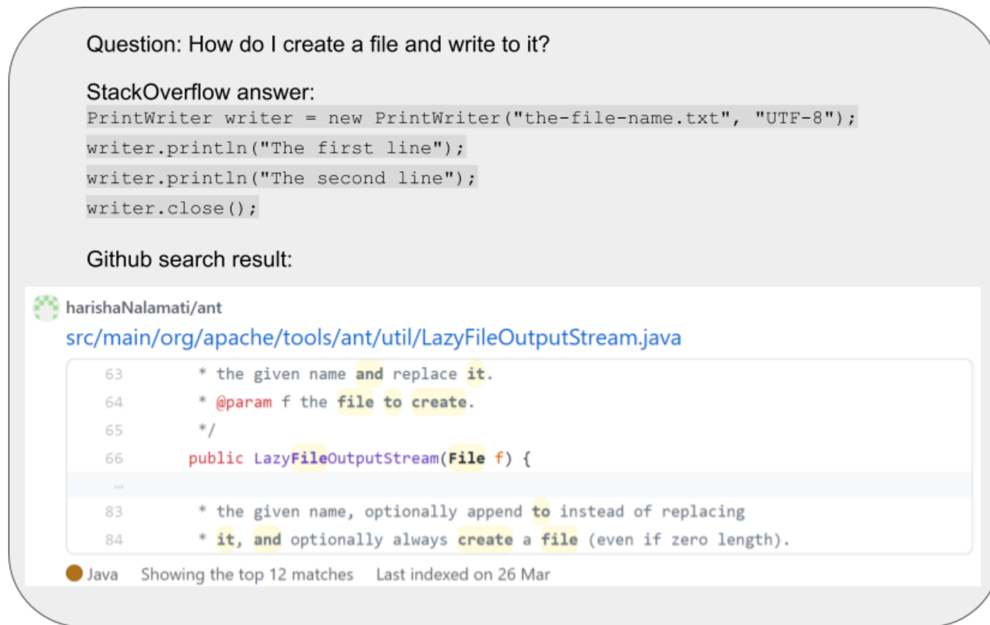


Figure 1.2.1: Example of a question on StackOverflow, the code of its accepted answer, and the result of searching for that sentence on Github. The latter references code unrelated to the answer but containing some of the words in the query.

to retrieve the most relevant documents from an existing corpus. In this type of task, Natural Language Processing models such as BERT have become popular for their accuracy. However, in cases of large document corpora, which is common, these models have long runtimes that make them unfeasible for production use.

1.3 Research Question and Contributions

The research question of this project will be: *How can we use BERT-based approaches to find code snippets given a query with a reasonable low latency?*

The main objective of this project is to provide new tools to help improve the state of the art of code search. Therefore, the main contributions of this project are:

- We propose a new model for code search that outperforms the state of the art with low latency.
- We evaluate our results with state of the art techniques.

1.4 Outline

This thesis is structured as follows. In Chapter 2, the theoretical background necessary to understand the techniques and technologies involved in the project is presented, as well as the related work. In chapter 3, all the assets involved in the development of the project (datasets, models, training frameworks and tests) are presented. In Chapter 4, the results of the tests applied to the models presented in the previous chapter are shown. In Chapter 5, conclusions and proposals for future steps are presented.

Chapter 2

Theoretical Background

In this section we define the concepts required to understand and follow the rest of the project. We start with Machine Learning and Deep Learning, two of the most popular trends in the world of artificial intelligence today, which form the technological basis of this project. We continue to define Siamese neural networks, attention mechanisms and Transformers, tools that we have used to build the models of this project. We end this chapter by talking about code search and mentioning related work in the field.

2.1 Machine Learning and Deep Learning

Machine Learning (ML) is a field at the intersection of computer science and mathematics that aims to solve problems by using historical data from the past to identify its patterns. For example, a simple regression algorithm can learn the relation between house features (such as size, number of bedrooms and construction year) to be able to figure out (to predict) its price by analysing existing data from other houses. These techniques can solve tasks that are difficult to define as a computer program but that a human could solve intuitively. For example, it is easy for humans to identify a car in a picture by, for example, looking for wheels on it, but it is complex to define a wheel in terms of pixels.

However, when the variability of the features of the data is so high that it changes for every single element of the dataset and requires nearly human-level understanding for solving the task, ML is not enough. In the example of car recognition, each example of our historical data would contain pictures with cars with different angle, color, illumination, visibility, shape or background.

Deep Learning (DL) [12] solves this problem by the concatenation of small functions that maps an input (pixels of a picture) with and output (presence or not of a car). The goal of a DL model is to define a function $f(x, \theta) = y$ where x are the inputs (pixels of a picture), y is the output (presence 1 or absence 0 of a car) and θ are training parameters. During the training phase, the DL will use the existing data to learn the θ that makes f to better map the defined inputs with the outputs. This function is composed by simpler functions (called neurons) that are stacked in layers. The bigger the number of layers, the more complex patterns the model will be able to learn. Each of the layers receives the data from the previous layer multiplied the training parameters θ_n for that layer. An activation function is applied to

the resulting value, that can be sigmoid, ReLU and tanh among others.

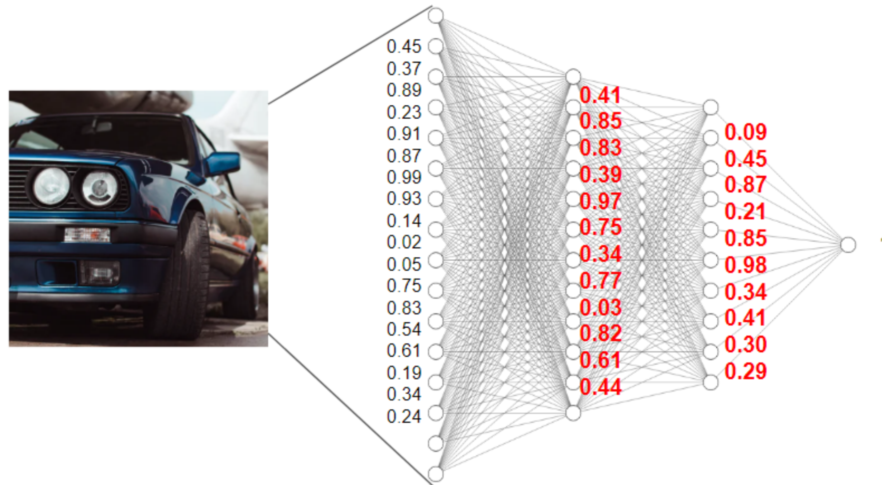


Figure 2.1.1: In our example with the car, the numeric value of the pixels is feed to the first layer of neurons. This values are multiplied by training parameters and feeded to the second layer neurons, that also will apply an activation function. This process continues until the last layer, that will return 1 if the image contains a car or 0 if it does not. The goal of training this neural network is to find the training parameters that will keep this behaviour with new and unknown images.

2.1.1 Siamese Neural Networks

The first time we heard about Siamese Neural Networks was back in 1993, when Bromley et al. [2] proposed this architecture to detect signature forgeries. The idea of this project was to feed two identical neural networks with the information of two signatures (obtained from a digitalizer tablet) and to try to discover if the second signature was a forgery attempt. This two submodels shared the training parameters so the training process had to be modified for constraining the weights to be identical in both neural networks. The cosine of the angle of the *numeric vector* obtained for each of these models represented the similarity between both signatures and the likelihood that one of the signatures was a forgery of the other.

This architecture has been used successfully in many areas, being Image Analysis the one with more applications [5], and in most of the cases the idea is always the same, obtain the similarity of two different elements such as fingerprints in images [1], questions [7] and cars in videos [18].

2.1.2 Embeddings

In the previous section we studied a model that uses a siamese neural network to generate a *numeric vector* used to be compared in order to get the similarity degree between two elements, in this case, signatures. These are feature vectors that allow a discrete object to be transformed into a distributed representation, so that information about the original entity is retained. For example, in image processing, the intermediate layers of a convolutional network can be used as a representation of an image [12]. The most popular case of embeddings and the one that is relevant to this work is that of words and phrases. Some popular embedding techniques, such as CBOW and Skip-gram [26], use DL models to

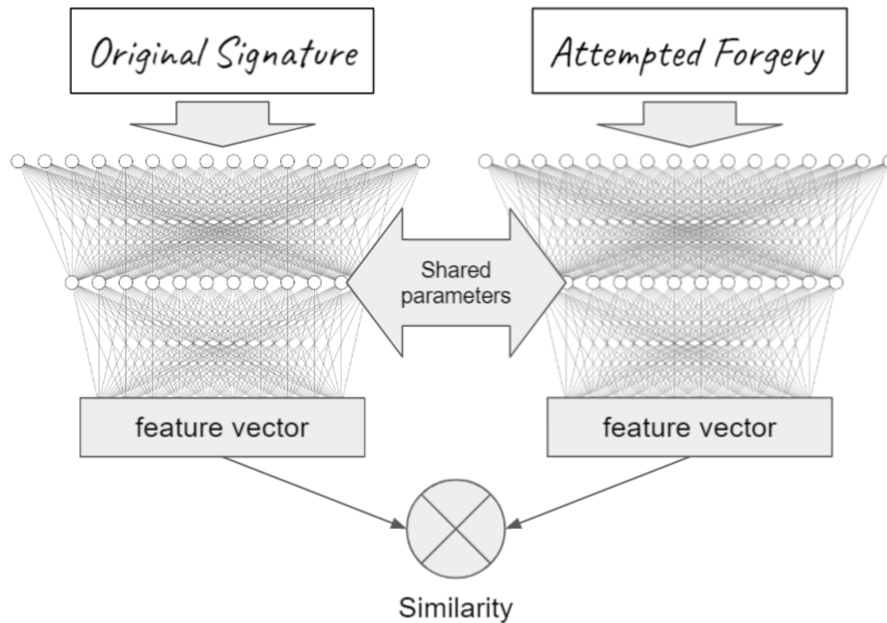


Figure 2.1.2: Siamese neural network for signature forgeries detection.

transform words into numeric vectors containing their semantic meaning. The possibility of transforming words into numbers makes it possible to compare them and operate with them mathematically (as in the classic example of [27] where they subtract the vector Man from the vector King and add the vector Woman, resulting in the word Queen).

2.1.3 Attention mechanism and Transformers

Natural Language Processing (NLP) is one of the most popular fields where DL is being successfully applied. Traditionally, sequence-to-sequence models based on Recurrent Neural Networks and Long Short-Term Memory models were used in text translation tasks. In this area, *attention techniques* appeared, mechanisms aimed to make the models to focus on specific parts of the input data. In image recognition [8], models can pay attention to specific parts of the picture to recognize figures and make predictions. In text translation tasks, a neural network can know which words are relevant (context) for each word of the input sentence, and it will be used to predict the associated word in the target language. The improvement compared to sequence-to-sequence models is that there is no loss of information in long sentences.

Dot-Product Attention is a simple attention technique that enriches word embeddings with information about their context. Given a sentence ($s = v_1, v_2, \cdots v_n$), for each of the word embeddings (v_i) is multiplied by the embeddings of rest the words and normalize the result. The obtained scores ($p_{i1}, p_{i2}, \cdots p_{in}$) are multiplied by the original embeddings ($s = v_1, v_2, \cdots v_n$) and the sum of the returned vectors will be the enriched embedding (y_i). This new embedding will contain information about the word and surrounding words. The following is the formula for applying the attention mechanism to the word v_i of the sentence $s = v_1, v_2, \cdots v_n$:

$y_i = sum(softmax(\sum_{j=1}^{n} v_i * v_j) * s)$

The word *mouse*, for example, can refer to a small animal or a computer peripheral. However, when

processed by an *Dot-Product Attention* module in the sentence *Click with the mouse on the trash can icon*, its embedding will lose its *animal meaning*, and reinforce its meaning on computer science.

Self-attention is an evolution of this approach. The basic idea is to add weights to the Dot-Product technique, allowing a neural network to learn more complex attention patterns and obtain better performance. In the previous example, each word embedding in the first step of the algorithm (v_i) is multiplied by a weight matrix called *query*, all the words embeddings are multiplied by *key* weight matrices before the dot-product and before being multiplied by the scores the word embeddings are multiplied by *value* weight matrices. As shown in Figure 2.1.3, with just three matrices, converted to linear layers, and very simple operations, the attention module obtained is very easy to implement and train using neural networks.

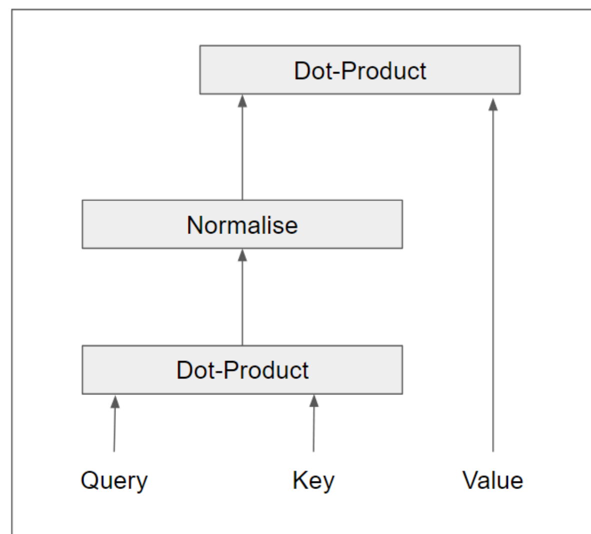


Figure 2.1.3: Self-attention mechanism

But in some cases, some words require paying attention to several words in the sentence. For example, in the sentence ”I give food to my dog”, the verb ”give”, should pay attention to ”I”, to ”food” and to ”dog”. Then, to scale the number of words to which attention is paid, it is possible to add in parallel more matrices (or linear layers) to ”query”, ”key” and ”value”. As result, we multiply the number of ”scores” before the final dot-product operation, and, instead of one final vector for each word, we will have multiple, one per matrix put in parallel. These final vectors are concatenated and fed to a dense layer. The result is a new embedding vector, that contains information about its context. This architecture is called *multi-headed attention*, where each of these new matrices is considered a ”head”.

The last evolution of this mechanism is an architecture presented in the paper *”Attention Is All You Need”* [34] that was a revolution in this area. The authors proposed an architecture called *Transformers* that consists of stacking encoders composed of a multi-headed attention module and dense layers. After both of these components, their output and inputs are combined in a ”add and norm” layer, that will improve training, avoid gradient vanishing and prevent drastic changes in weights.

The Transformer architecture, unlike recurrent approaches, feeds the entire sentence in parallel and ignores the position of each word. In some cases, the order of the words could be important. For example, the meaning of the sentence *”It’s raining today, that’s why I’m not sad”* changes completely

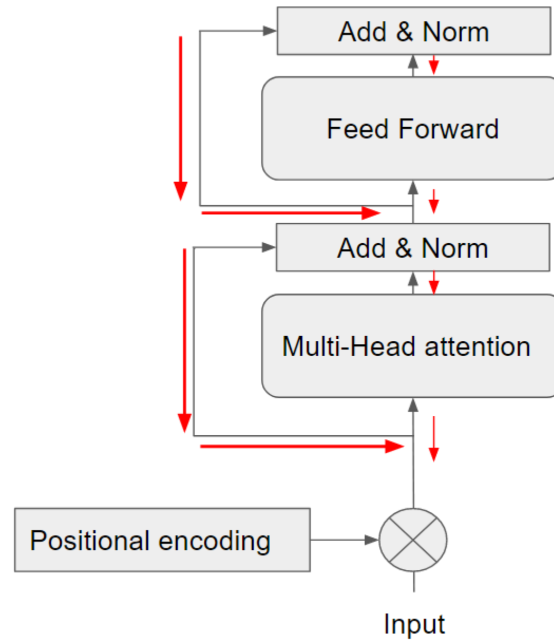


Figure 2.1.4: Transformer encoder architecture. During the training, the gradient signal (in red) splits: one of the branches goes through the Feed Forward and Attention modules, but the other avoids them. This technique prevents the gradient vanishing when multiple encoders are stacked.

if we modify the position of the word *not* as in *"It's not raining today, that's why I'm sad"*. That is why the authors of the project proposed a *positional encoding*: a fixed matrix that contains a notion of the position of each word is added to the input embedding at the beginning of the entire module. This positional encoding matrix is calculated with the following rules:

$$P(pos, 2i) = sin(\frac{pos}{1000^{2i/d_{model}}})$$

$$P(pos, 2i+1) = cos(\frac{pos}{1000^{2i/d_{model}}})$$

Where pos refers to the position in the sentence, i refers to the embedding position and d_{model} refers to size of the embedding. The authors stated that the sinusoidal wavelengths generated for each encoding dimension allow the transformer to learn from the relative positions, as well as to apply this behavior to longer sentences than those found during training.

Transformers outperformed many state-of-the-art NLP models in several tasks[34], being the base of some popular architectures like BERT[9] and GPT-3[3].

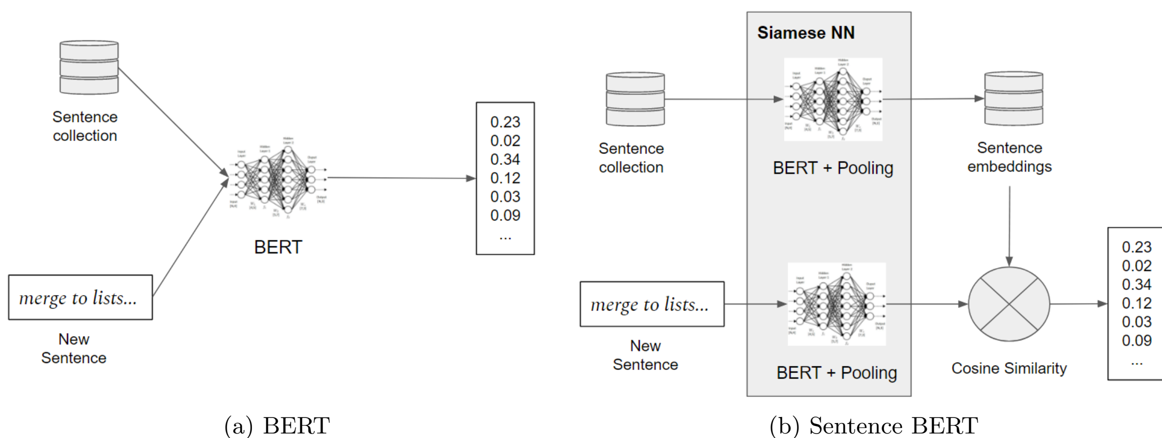
2.1.4 BERT and Sentence-BERT

BERT [9] is the state-of-the-art model for NLP developed by Google. The authors emphasize the importance of bidirectional approaches in language models and, for this purpose, BERT is based on Transformers and trained with for solving Masked Language Modeling tasks. This task consist of masking some of the words in the input and try to predict it in the output, using the left and the right context. In addition to this task, BERT is trained to predict when a sentence B follows another sentence A. This Next Sentence Prediction task makes BERT solve Question Answering and Natural Language Inference tasks. With this pretraining, BERT is able to generate robust embeddings, where each of the words

contains information of its context. The key to the success of such pretrained models is that with simple modifications they can be fine tuned to solve tasks for which they were not trained.

The success of BERT and its good results applied in various fields of NLP is sometimes overshadowed when used in text extraction tasks, especially when applied to large corpora. The main problem lies in the fact that BERT's base architecture (mono) requires embedding each document in the corpus for each query performed, generating performance problems[17]. Over the last few years many variants of BERT have emerged that have tried to improve its performance, but Sentence-BERT [29] is one of those that addresses its latency problems. This branch of the BERT family generates a single embedding vector for an entire sentence, by applying a max or mean pooling to the embedding of all the words in the sentence. Unlike BERT, which is set up to receive two sentences for tasks such as sentence similarity, Sentence-BERT uses a Siamese neural network architecture to receive, in each BERT module, one sentence individually. This difference in architecture results in a significant improvement in performance at the inference stage.

Lets say we have a group of sentences, and for each new sentence, we want to get the most similar one in our collection. With BERT, we create embeddings that contains both sentences to compare (the new one and each element of our collection). Therefore, we need to compute an embedding for each of the elements of our dataset everytime a new sentence is provided. However, with Sentence-BERT, we can have stored the embeddings of all our sentences of our collection and, at inference time, we only need to embed the new sentence and calculate its similarity with the embeddings of our collection.



(a) BERT (b) Sentence BERT

Figure 2.1.5: In sentence pair tasks, BERT generates its embeddings with each pair of sentences. With Sentence-BERT, each embedding is computed individually so only the embedding of the new phrase and the cosine similarity has to be executed at inference time.

Although the improvement in inference time (efficiency) is remarkable when using statement-based architectures with respect to single models (mono), there is a significant loss in output quality levels (effectiveness) [17].

2.2 Code Search

Searching for code to solve bugs, implement an algorithm, refactor code, understand how an API works and solve technical doubts, is a common practice in the day to day life of software developers.

These queries can be made on Q&A sites such as StackOverflow, one of the most popular computer programming questions sites. When a developer has a query, she/he has to create a new question on this site, or search among the existing questions one that resembles the user's one. The search engine will return questions that contains words that are also in the in the user's query, but it is not capable to find the patterns that relate the question with the snippet of code that solves it.

Code Search is a subarea of text retrieval applied in Software engineering that aims to find this relation by transforming both code and description in representation (numeric vector) that can be compared. More information about the state-of-the-art approaches in this area can found in Section 2.3.

2.3 Related work

Code search has been a much studied topic in the last decade, due to the popularity of opensource, code reuse and software engineering QA sites. Initially, most of the approaches were based on information retrieval techniques. For example CodeHow [21] attempted to match a user's query to an API by comparing it to its description and name, creating vectors based on term frequency and inverse document frequency.

With the rise in popularity of ML and DL in recent years, new projects using this technology have emerged, surpassing previous techniques and leading the way for future research. These new proposals diverge in different aspects, such as architecture, training technique or tokenization, but they tend to have one thing in common: the core idea is to generate a numeric vector (embedding) for code snippets and for its description or query and try to make these two vectors as close as possible. This ”closeness” is commonly measured with cosine similarity.

With LSTM architectures reaping success in the field of artificial intelligence in different sectors, one of the first code search models based on DL had this recurrent neural networks as a fundamental part. Deep Code Search [13] generates embeddings of code snippets feeding the tokens, the API sequences and the method names to bi-directional LSTM layers, and combining them in a maxpooling layer.

Despite the good results of this project, it has the disadvantage of being a slightly complex architecture and depends on the availability of the method name and API sequence for each code snippet. UNIF [4] is a proposal that claims that smaller and simpler approaches should be tested first, before trying more complex architectures. This model uses only the code and description tokens, adding an attention layer to the former and an averaging layer to the latter. This model outperformed previous proposals without using the method name, allowing it to be used in cases where this is not available, such as in datasets taken from QA sites.

Many other projects start from this baseline, varying the way in which the embedding is generated. For example CoNCRA [22] generate the code token embeddings using convolutional layers, and CQIL [16] apply them to both tokens and method name. But the embedding is not the only differentiating factor in the different code search approaches. CoaCor [36] uses a Reinforcement Learning component to generate additional descriptions for a given code snippet, and during inference stage, each query is compared with both code and the additional description embeddings, combining both similarities in a single value.

Chapter 3

Research Methodology

In this chapter we define the assets involved in the experiments performed, including the dataset, models, training techniques, metrics and tests.

3.1 Models

In this section we describe the models involved in this project, starting with UNIF, a simple but effective model that we will use as baseline, UNIF SNN, an enhanced version of this previous model, Sentence-BERT and monoBERT, our BERT based proposed models, and we finish defining the training process, the metrics and the tests performed.

3.1.1 UNIF

When Deep Learning Met Code Search [4] is a very popular project due to its simplicity and its good results. The authors studied the state of the art of code search techniques to find a new approach with better results than existing ones. Previous projects Deep Code Search (DCS) [13] and Neural Code Search (NCS) [31], which were part of the state of the art of code search, were evaluated in this study. DCS is a model that uses DL to embed code method names, tokens and API sequences to a vector space and compared it to the embedding of its description. NCS embeds both code and description tokens in the same space using a combination of information retrieval techniques without any supervised technique.

The model proposed after this was UNIF, a supervised extension of NCS that used two embeddings matrices and an Attention mechanism [34]. Both code and description are embedded with their own matrix and cosine similarity of the vectors of both embeddings is calculated (see Figure 3.1.1). During the training process, the embedding and attention parameters are tuned to maximize the similarity between a piece of code and its description. The advantages of this proposed model is that it outperforms previous state-of-the-art approaches by using a simpler model than other DL models like DCS. The authors suggested that simpler models should be tested before starting adding complex mechanisms like Recurrent Neural Networks in code search projects.

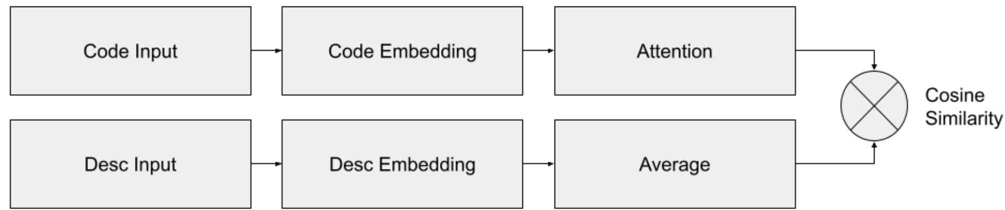


Figure 3.1.1: UNIF model

The UNIF model behaves as follows: first, each tokenized code snippet is transformed into a real-valued vector using an embedding layer. For example, a code snippet *"for(int i=0; i<10; i++) print(i)"* would be tokenized, creating a one hot vector (e.g. *[37, 24, ... 29]*), in which each number corresponds to the id of one of the words in the original code. The embedding layer will assign a dense vector for each id (e.g. *37* could be transformed to the vector *[0.73, 0.87, ... 0.23]*), generating an $N \times M$ matrix where N is the length of the tokenized code snippet and M is the embedding dimension or the length of the embedding vector of each word. The generation of these embeddings is random at first, depending on parameters that will be adjusted during training. The description of the code snippet will go through the same process but with an embedding layer and parameters independent from those of the code.

In the second step, the matrix $N \times M$ of the description is transformed into a 1-dimensional vector through an average operation. In the case of the code snippet, before this operation, the embedding is passed through a Self-attention layer to enrich it with information about its context.

At this point, the model would have generated vector of size M for both code snippet and description. Finally, the cosine similarity between both vectors is calculated, obtaining as a result the similarity between the snippet code and the description provided.

Section 3.3 shows how the training of this model is performed, in which the parameters of the embedding and attention layers will be tuned so that the cosine similarity between a code and its description is as high as possible, as well as to reduce the similarity between a snippet and a randomly chosen description.

3.1.2 UNIF SNN

Inspired on this last model, we propose a new architecture, where both description and code tokens are embedded in the same vector space, this means using the same parameters for generating the embedding vectors of the code and the description. The motivation behind this idea is that using the same vector space could allow the model to learn the semantics of words that are used in code snippets and in descriptions or queries. To study the usefulness of this architecture, we propose a modification of the UNIF model, described in the previous section, in which both code and description are fed to Siamese neural networks, consisting of an embedding layer and an attention layer (see Figure 3.1.2). In this way, we will be able to evaluate whether Siamese networks are an improvement for the code search models.

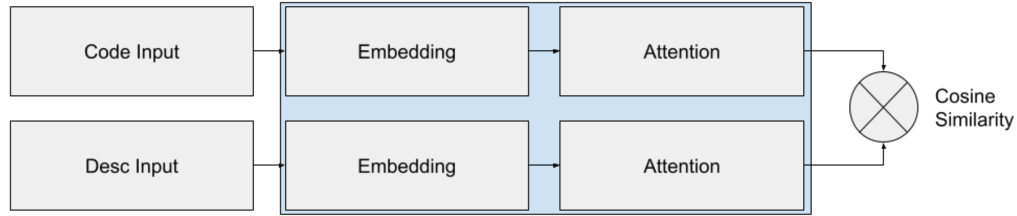


Figure 3.1.2: Unlike the original UNIF model, in UNIF SNN both the description and the code share the embedding and attention layer. For example, the word ”for” would have the same id and therefore the same embedding vector when it is included in a description and in a code snippet. This could allow the model to understand, for each word, its meaning within a description and within a code snippet.

3.1.3 Sentence-BERT and monoBERT

Sentence-Bert [29] is a modification of BERT models that uses Siamese neural networks to improve performance on semantic similarity tasks. As described in Section 2.1.4, the key is the way in which sentence pairs are fed to the model: each sentence is embedded individually using a BERT instance and a mean pooling layer. This architecture is flexible with the type of BERT model. For this project, we will try the general purpose models BERT and roBERTa.

RoBERTa [19] is an optimization of the original BERT model that outperforms it and other BERT variants in various semantic similarity tasks. The authors claim that BERT is undertrained, and their changes in hyperparameters, training with a larger dataset, longer sequences, and removal of the ”next sentence prediction” task improve BERT’s performance on all tests.

Opposed to Sentence-BERT approach we find the standard BERT architecture, monoBERT: using a single BERT entity to embed both sentences and fine-tune the model appending a final feedforward layer to classify results. The sentences are fed to the model with the following format:

[CLS] <SENTENCE 1> [SEP] <SENTENCE 2> [SEP] [PAD]

CLS is a token that BERT expects to receive at the beginning of each pair of sentences and its embedding can be considered as a representation of the entire sentence. *SEP* is a token placed at the end of each sentence used to separate them.

3.1.4 Multistage architecture

MonoBERT is not the best approach for information retrieval tasks such as code search because its huge latency compared to Sentence-BERT. In the case of the first one, every time a new query arrives, it is necessary to feed the model with this sentence and with all the documents (in our case, code snippets) in our corpus, but with the second one, the model is only fed with the new query, making it much faster. However, in this project monoBERT is going to be tested and compared with the rest of models and, in the case that monoBERT outperforms sentence-based approaches, it is necessary to find an architecture that reduces the number of candidate code snippets before feeding them to the model.

We propose a two-stage architecture (see Figure 3.1.3): first, a sentence-based model (UNIF, UNIF-SNN or Sentence-BERT) will obtain a number N of candidates, taking advantage of the efficiency of this architecture to search for candidates in large code repositories. As already explained, the embedded

code snippets can be stored in a database and, at the time of inference, it is only necessary to embed the new query and calculate the cosine similarity between it and the stored vectors. During the second stage, the monoBERT model will sort the N candidates to obtain the final ranking.

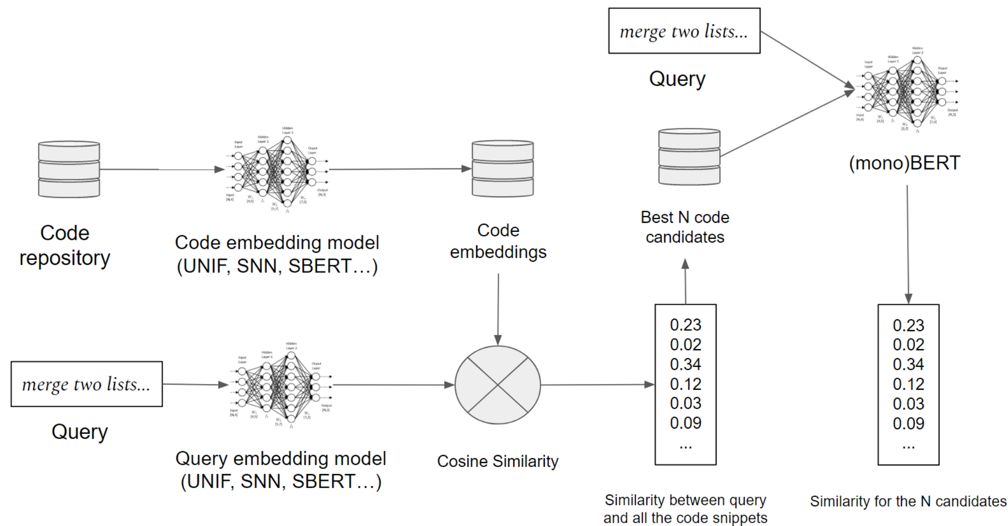


Figure 3.1.3: Two stages architecture.

We will not dwell too much on the selection of an optimal N, as long as the final results of this proposal are not much worse than the results of monoBERT when run outside this two-stage architecture.

3.2 Dataset

For this project, we looked for a dataset large enough to fine-tune the BERT models. We decided to use a dataset in which the code has been parsed and cleaned of programming characters (e.g., {, }, [and]) in order to test standard BERT models and compare them with simpler models such as UNIF. The chosen dataset comes from the DCS [13] project, a code search research that aims to match a large-scale corpus of java code methods with their descriptions. Due to the characteristics of the project, the corpus contains also the name of code method and the API sequence, that are going to be discarded in this work. The description of the dataset already tokenized is described in Table 3.2.1.

Table 3.2.1: Deep Code Search dataset summary.

Description-code tuples	18.223.872
Words in code corpus	187.708.864
Words in description corpus	180.242.654
Different words in vocabulary	13.645
Number of words longest sentence	155

This dataset has a different vocabulary for the code and the description. This means that a specific word (e.g. *array*) will have a different id associated in the code dataset (e.g., 132) than in the description set (e.g., 39). This is only acceptable for UNIF because both code and description have independent embeddings and they do not need to share the vocabulary. However, for SNN, as we use the same embedding for both models, we need each word to have the same id in the description and the code

set, that is why we retokenized this dataset for the SNN model. For the BERT models, we decoded the sentences (from token id to word string) and applied the associated BERT tokenizer.

3.3 Training

In this section we present the Triplet network framework, that is the methology followed to train UNIF, UNIF Siamese and Sentence-BERT, and also we describe how the monoBERT model was trained.

3.3.1 Triplet network framework

UNIF, UNIF Siamese and the Sentence-BERT models will be trained using triplet networks [14], an architecture that is popular in sentence similarity tasks. The idea is to feed a network with a sentence (anchor), feed the second with a similar sentence (positive) and feed the third with a randomly picked sentence or one with an opposite meaning (negative). Then, the similarities between the anchor and the positive sample (positive similarity) and between the anchor and the negative sample (negative similarity) are calculated. This model is trained using the following triplet loss function $\mathfrak{L}$:

$$\mathfrak{L} = \alpha - S_p + S_n$$

Where α is a value between 0 and 1 that defines a margin or difference between a positive and a negative value, S_p is the positive similarity and S_n is the negative similarity. During the training phase, the parameters of all the models will be tuned to maximize S_p and minimize S_n.

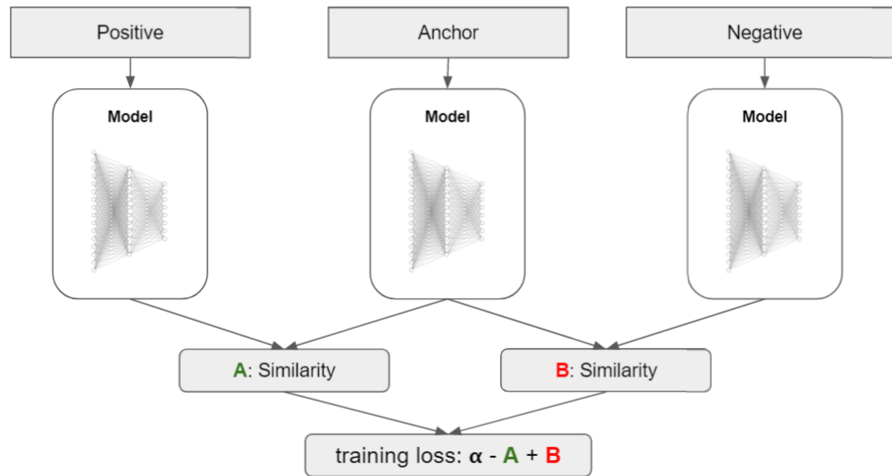


Figure 3.3.1: Triplet networks architecture.

For this project, the anchor of the triplet network will be a query or description, the positive sample will be the code snippet that matches this description, and the negative sample will be a randomly picked code snippet. The cosine similarity between the description and the code snippet will be used as the positive similarity and the one between the description and the negative sample will be the negative similarity.

Both the Positive and Negative models (see Figure 3.3.1) learn how to embed the code. To do so, they share all their parameters, making them siamese. The anchor model embeds the description, and only

in cases where the code and the description share an embedding model will they be siamese as well. For example, UNIF has independent embeddings for the code and the description, but in the case of its Siamese version (Section 3.1.2), as in Sentence-BERT, the description, the code and the negative code will be Siamese.

3.3.2 MonoBERT training

The monoBERT models requires a different training framework. A classification layer with a sigmoid function and one neuron as output will be added to the BERT module. Specifically, the output corresponding to the CLS token of the BERT model, which serves as a summary of the embedding of the entire model, will be the input to this classification layer (see Figure 3.3.2).

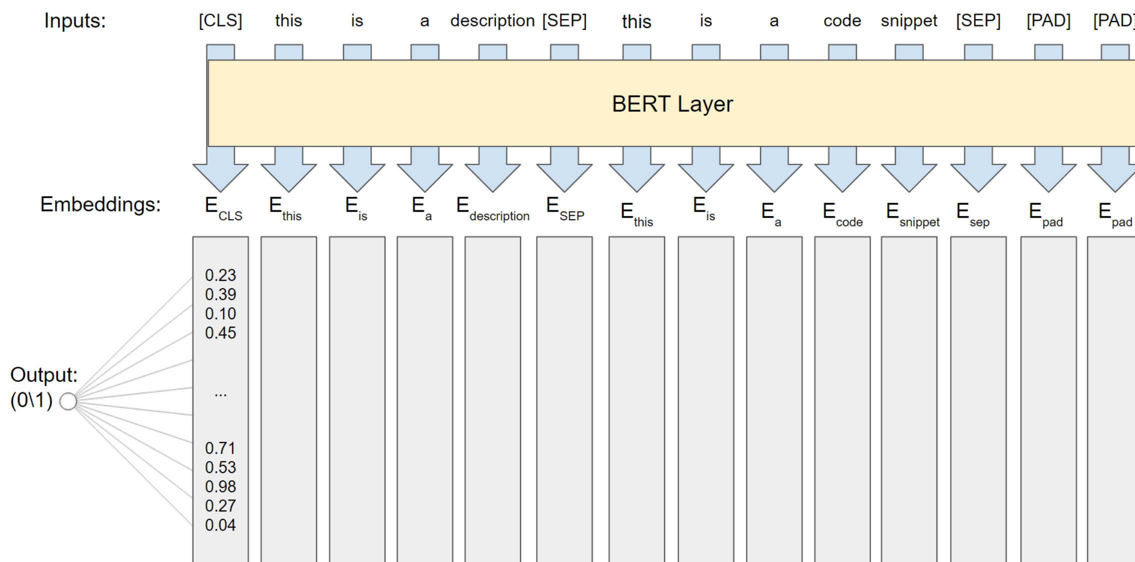


Figure 3.3.2: MonoBERT architecture. In this example, the description ”this is a description” and the snippet ”this is a code snippet” are fed to the BERT layer. Then, the embedding of the *CLS* token is used to generate the output (1 in case of matching, 0 in other case).

The dataset will be processed to match the input format of BERT and the output format of the classification layer. To do this, for each pair of code and description, a label with value 1 (the description matches the code) will be assigned, and a random code fragment will be searched for and associated with that description, with a label 0 (the description does not match the code). In this way the dataset will be duplicated, so that for each record (code-description pair) there will be a positive and a negative example (Table 3.3.1 shows the format of this dataset).

Table 3.3.1: For each Description-Code pair, a positive label (1) is assigned, and another entry is added to the dataset, with the same description, a randomly picked code snippet and a negative label (0).

Description	Code snippet	Label
Desc1	Code1	1
Desc1	Code random	0
Desc2	Code2	1
Desc2	Code random	0
Desc3	Code3	1
Desc3	Code random	0

3.4 Tests

After training each of the models described in section 3.1, they will be tested. In the next section we describe the metrics used to evaluate each model, as well as two additional tests to evaluate the robustness and usefulness of the obtained solution.

3.4.1 Metrics

There are many ways to compare the accuracy of code search systems. The most popular one is Top-N (or Recall@K), a metric used in recommendation algorithms that measures how often the true label is inside the Top N values of a prediction (Table 3.4.1 shows a toy example of this metric). This is an appropriated metric for code search because users often check the first results of a query in a search engine instead of just the first one.

Table 3.4.1: In this example, only the predicted values of rows 1, 2 and 5 are correct (60% accuracy). But, if we check the presence of the True value inside each top 3 predictions, also the row 4 would be correct (80% accuracy)

Id	True value	Predicted Value	Top 3 predicted values
1	A	A	A, B, D
2	B	B	B, C, D
3	C	D	D, A, B
4	A	B	B, A, C
5	B	B	B, C, D

For each query of the testset, the ranking of the code snippets of the testset is created and the position of the code fragment that corresponds to it is searched. To obtain the TopN metric, the proportion of queries whose code snippet was in position N or higher will be calculated. This process will be performed for each of the single-phase (not multistage) models.

In the case of the multi-stage model, the same test will be performed for monoBERT, but in this case, the snippets used for ranking will be a shorter list of candidates coming from the top of the ranking of a previous model (e.g., Sentence-BERT or UNIF). By testing high-accuracy, high-latency models such as monoBERT both individually and within a multi-stage architecture, we will be able to evaluate the extent to which accuracy is lost compared to the improvement in latency.

3.4.2 Rephrasing test

Once a model has been evaluated with the test set, we ask ourselves whether it is sufficient to test its robustness. As Gan el al. said [11], tests and training sets often have a common origin and follow the same distribution, which could call into question the generalizability of a NLP model. In their project, Gan et al. [11] modified (while maintaining their meaning) the test set questions of a Question Answering model and observed that the models worsened their results.

We propose to apply the same principle to Code Search. Given a query or description (D) and one of our models (M) that is able to find the code fragment (C) that matches that description, do I get the same element if I rephrase the query (D2), keeping the semantic meaning?

if M(D) = C, is M(D2) = C if D $\approx D2$?

We will propose a parallel dataset with descriptions selected from the test set (A). We will reformulate those descriptions into a second dataset (B) and run the models with both sets using the Top-N metrics. The variation between the results of the two sets will give us an idea of the extent to which the performance of the models is strongly dependent on the words used. This test will allow us to determine to what extent the models are able to relate the meaning of the query to the code, regardless of the words used.

3.4.3 Search examples

The use of metrics such as TopN makes it easy to compare different models, as well as to evaluate how much better or worse different design decisions, such as multistage architecture, are. However, these metrics do not always give an idea of the usefulness of these types of solutions for users. That is why at the end of the tests, with the best model obtained we will proceed to feed it with test queries, not obtained from the testset, and evaluate the relationship of the output snippets with the input query and the relevance of these results for a user.

Chapter 4

Experiments and Results

In this chapter we are going to describe the experiments conducted with each model and discuss their results.

4.1 Experiments

Each of the models were developed using the ML frameworks Tensorflow [23] and Keras [6] and packaged and deployed using Docker [24] in a server with 4 Nvidia GeForce RTX 2070 gpus. For the loading and configuration of BERT-based models we used the Hugging Face Transformers framework [35]. All the models were trained with learning rates between *4e-5* and *1e-6*. We realized that bigger learning rates, like *1e-4*, have negative impact in BERT models. We also evaluated different triplet margin values between 0.6 and 0.2 and saved the best results.

The training set contained 600K code-description pairs from the original Deep Code Search dataset (defined in Section 3.2) because the size of the original dataset (over 18 million pairs) made full training for all models unfeasible for the duration of this project. BERT-based models are resource-intensive. The amount of memory required during its training increases with the maximum length of sentences used. We, therefore, decided to truncate the descriptions and code fragments to 90 words, which is the length of the longest description in our dataset. This way no information will be lost from the descriptions during training.

The tests consist of iterate over a test set, generate the embedding for each description, calculate the similarity with K code snippets embeddings and extract the ranking of the correct snippet. Since it is unfeasible to test the non-sentence-models (mono-BERT and mono-roBERTa) with the entire test set, tests are performed with groups of $K{=}500$ description-code pairs from the test set. The results are shown in Table 4.1.1.

This experiments reveal that the simplest model, UNIF [4], gets better results with a Siamese architecture than with separate embeddings for the code and the description. Our experiments also reveal that this model, that uses an attention layer as embedding vector, get better results than the sentence-based models, that relies on the mean pooling of BERT output.

Table 4.1.1: Results of all the models executed with 500 descriptions and 500 code snippets from the test set.

	DCS				
Metric	Top1	Top3	Top5	Top15	Time
UNIF	0.47	0.66	0.74	0.87	36s
UNIF (SNN)	0.57	0.73	0.80	0.88	1m 7s
mono-BERT	**0.63**	0.83	0.86	0.93	17m 30s
mono-roBERTa	0.61	0.79	0.87	0.92	16m 5s
Sentence-BERT	0.49	0.71	0.77	0.87	2m 2s
Sentence-roBERTa	0.46	0.66	0.74	0.85	42s
UNIF+monoBERT	0.64	0.79	0.83	0.87	1m 9s
SNN+monoBERT	**0.64**	0.81	0.84	0.88	**1m 10s**
S-BERT+monoBERT	0.64	0.81	0.83	0.87	1m 14s
S-roBERTa+monoBERT	0.62	0.79	0.82	0.85	1m 13s

One possible reason for this behavior is the format of the input data. BERT is designed to generate embeddings from natural language sentence and the input to our model consists of tokens extracted from a code snippets instead of a traditional English sentence.

Our experiments confirm also that the *mono* architectures outperform sentence-based approaches in code search tasks as they do in other text retrieval tasks [17]. However, the high inference time in this models still making them unfeasible in production environments. For this reason we combine sentence-based models with *mono*models to reduce latency and minimize the impact on the quality of the output. As shown in Table 4.1.1, the Top1 metric for monoBERT is very similar to that of Sentence-roBERTa + monoBERT, but the execution time is considerably shorter in the latter. In all these cases, the first stage selects the top 15 candidates with the most similarity to this query. The choice of the number of candidates was arbitrary but is supported by the fact that the Top15 of all the first stage models is always higher than the Top1 and Top3 second stage models. The goal of this architecture is to have Top1 and Top3 scores close to those of the models used in the second stage, so when choosing the number of candidates K, we need a number in which the TopK score of the models used in the first stage is not lower than the Top1 and Top3 of the second stage models, otherwise, the first model would filter out the truth ground more times than the second model does, reducing the final performance of the architecture.

In order to validate the robustness of our models, we modified 41 descriptions from the test set and calculated the TopN metrics of these sentences and their original versions. As shown in Table 4.1.2, monoBERT outperforms UNIF and SNN, and all the multistage models outperform the single-stage models. The good results obtained in monoBERT are maintained, or even improved, in the multi-stage models that use it as the final ranker

In Table 4.1.3 we present some rephrasing examples for the model S-roBERTa+monoBERT, the one that had better results in this tests. The ranks column shows the number of code snippets that had better rank than the ground truth. For example, if rank is 0, it means that for that query the model returned the correct snippet at first position. If rank is 4, it means that four code snippets got better rank that the correct code, that got the fifth in the ranking.

Table 4.1.2: TopN metrics with 41 selected queries and their rephrased versions, with 500 code snippets

	Selected			Rephrased		
Metric	Top1	Top3	Top5	Top1	Top3	Top5
UNIF	0.53	0.85	0.90	0.29	0.53	0.63
UNIF SNN	0.70	0.82	0.90	0.43	0.63	0.70
monoBERT	0.70	0.97	0.97	0.51	0.68	0.78
Sentence-roBERTa	0.58	0.75	0.85	0.39	0.56	0.68
UNIF+monoBERT	0.78	0.92	0.95	0.53	0.70	0.73
SNN+monoBERT	0.78	0.95	0.95	0.53	0.73	0.75
S-roBERTa+monoBERT	0.75	0.95	0.95	0.53	0.73	0.75

For example, the examples 83 and 329 got better new rank (moves from second to first position) with the rephrased query, that contains the semantic meaning of the original sentence but with less words. The example 5 got a worse ranking even when both original and rephrased sentences are very similar. In the example 153, the ranking has worsened significantly with the new sentence, which is an example of a poorly detailed query. The word "java" probably made this search more difficult, as it may not be a common word in the training set method description. Example 15 is an example that this model has yet to learn about context, as it has not been able to identify "http request" with "post method call".

Table 4.1.3: Example of the ranking of descriptions and its paraphrased version with the S-roBERTa+monoBERT model.

Id	Original rank	New rank	Original	Rephrased
83	0	0	return the canonical path of this file	how to get canonical path of the given file
329	1	0	generate a random int as a token	how to get a random integer
73	1	0	turn array of bytes into string representing each byte as unsigned hex number	how to transform list bytes into string hexadecimal representation
5	0	17	pause processing at the socket	how to stop socket processing
153	2	223	get an xml representation of this object	from java object to xml
144	0	1	set the maximum allowed number of threads	how to define max threads
15	0	16	do a form post method call	how to perform a http post request

4.2 Code Search examples

In this section we are going to present some examples of queries feed to Sentence-roBERTa+monoBERT and the code snippet retrieved. For these tests, the model searched for the three code snippets with the highest similarity to the query provided among 10,000 candidates in the test set. It should be noted that the following queries have been made without knowing all the code snippets included in the repository, which can sometimes result in the required functionality not being found in the possible candidates.

In the first example (Table 4.2.1) the model retrieves two code fragments that solve the functionality

sought in the query, and even the second candidate fits better with the input query than with its own description. The third candidate, although it does not fit the requested functionality, is an example of how this model can be used to search for classes in private code repositories. In this case it is returning the *Sort* class from an Android library called *libgdx* [1].

Input query	how to sort arraylist	
Snippet rank	**Original description**	**Code retrieved**

First	sort the given array with a default annotation aware order comparator	`public static void sort(Object[] array) {`
		`  if (array.length > 1) {`
		`    Arrays.sort(array, INSTANCE);`
		`  }`
		`}`

Second	sort thumbs according to their width	`public void sort() {`
		`    Collections.sort(this);`
		`}`

Third	return a sort instance for convenience	`static public Sort instance() {`
		`  if (instance == null) instance = new Sort();`
		`  return instance;`
		`}`

Table 4.2.1: Search example 1

In the second example (Table 4.2.2) the two first code snippets retrieved are related with what the user would expect but does not really fit what the user would require for that input query. However, the third snippet matches the desired functionality.

In the third example (Table 4.2.3) only the second candidate does what the user would look for with the input query.

The fourth example (Table 4.2.4) none of the retrieved codes matches the query, but they are a good example of how this model could be used to find a *remove* functionality inside a framework from a private repository.

In the fifth example (Table 4.2.5), although it is the worst of the examples presented here since two of the snippets retrieved have little or nothing to do with the query, it contains a function to calculate the average of a list, including the sum of the elements of the list, which is the expected functionality.

For sixth example (Table 4.2.6), the model returns three ways to open a file, ignoring the required CSV format.

In the last example (Table 4.2.7) we use the query from the problem definition section (Section 1.2) and the model retrieves two code snippets that solves the problem.

In conclusion, these examples give a different perspective to the metrics provided in the tests presented

[1] https://github.com/libgdx/libgdx/blob/master/gdx/src/com/badlogic/gdx/utils/Sort.java

Input query	how to connect to a database	
Snippet rank	**Original description**	**Code retrieved**

First	construct an instance of explain result given a prepared statement object	`public PostgresExplainResult(PreparedStatement stmt ) throws SQLException {`
		`if (stmt == null) {`
		`throw (new NullPointerException(" PreparedStatement-argument-cannot-be-null") );`
		`}`
		`Connection database = stmt.getConnection();`
		`if (database == null) {`
		`throw (new NullPointerException("Failed-to- retrieve-Connection"`
		`+ "-from-PreparedStatement"));`
		`}`
		`stmt.execute();`
		`retrieveExplainString(stmt);`
		`stmt.close();`
		`}`

Second	get the primary column for a table	`public static String getPrimaryKeyColumn(Connection conn, String table) throws SQLException {`
		`logger.debug("getPrimaryKeyColumn(conn={},-table ={})---start", conn, table);`
		`DatabaseMetaData metadata = conn.getMetaData();`
		`ResultSet rs = metadata.getPrimaryKeys(null, null , table);`
		`rs.next();`
		`String pkColumn = rs.getString(4);`
		`return pkColumn;`
		`}`

Third	static method for initial connecting to db	`public static void init(String databaseName) throws SQLException {`
		`db = DriverManager.getConnection("jdbc:sqlite:" + databaseName + ".db");`
		`}`

Table 4.2.2: Search example 2

above, showing that, although the code snippets retrieved do not always perfectly match what the user is looking for, they are usually related and contain code that can be useful to the user.

Input query	how to upload file to server

Snippet rank	**Original description**
First	download a given file from a target url to a given destination file

Code retrieved:

```java
public static boolean DownloadFromUrl(String
    targetUrl, File file) {
try {
  URL url = new URL(targetUrl);
  URLConnection ucon = url.openConnection();
  InputStream is = ucon.getInputStream();
  BufferedInputStream bis = new BufferedInputStream
      (is);
  ByteArrayBuffer baf = new ByteArrayBuffer(50);
  int current = 0;
  while ((current = bis.read()) != -1) {
    baf.append((byte) current);
  }
  FileOutputStream fos = new FileOutputStream(file)
      ;
  fos.write(baf.toByteArray());
  fos.close();
 } catch (IOException e) {
  Log.d(LOG_TAG, "Failed-to-download-file:-" + e);
  return false;
 }
 return true;
}
```

Snippet rank	**Original description**
Second	binary streams the specified file to the http response in 1KB chunks

Code retrieved:

```java
public static void sendTempFile(File file,
    HttpServletResponse response) throws
    IOException {
  String mimeType = "image/png";
  String filename = file.getName();
  ServletUtilities.sendTempFile(file, response,
      mimeType);
}
```

Snippet rank	**Original description**
Third	lookup a resource based on the request UNK and send it using send file

Code retrieved:

```java
protected boolean handle(final String uri, final
    Request req, final Response res) throws
    Exception {
 boolean found = false;
 final File[] fileFolders = docRoots.getArray();
 File resource = null;
 for (int i = 0; i < fileFolders.length; i++) {
   final File webDir = fileFolders[i];
   resource = new File(webDir, uri);
   final boolean exists = resource.exists();
   final boolean isDirectory = resource.isDirectory
       ();
   if (exists && isDirectory) {
     final File f = new File(resource, "/index.html"
         );
     if (f.exists()) {
       resource = f;}
   }
 }
 pickupContentType(res, resource);
 sendFile(res, resource);
 return true;
}
```

Table 4.2.3: Search example 3

Input query	how to remove object from memory	
Snippet rank	**Original description**	**Code retrieved**

First	remove the specified object from this hash set	`@ Override public boolean remove(Object object) {`
		`  return backingMap.remove(object) != null;`
		`}`

Second	purge stale mappings from this map before read operations	`protected void purgeBeforeRead() {`
		`  purge();`
		`}`

Third	artificially match a data node used by remove	`final boolean tryMatchData() {`
		`  Object x = item;`
		`  if (x != null && x != this && casItem(x, null)) {`
		`    LockSupport.unpark(waiter);`
		`    return true;`
		`  }`
		`  return false;`
		`}`

Table 4.2.4: Search example 4

Input query	how to sum numbers in a list	
Snippet rank	**Original description**	**Code retrieved**

First	calculate the average length across all x value strings	`private void calcXValAverageLength() {`
		`  if (mXVals.size() <= 0) {`
		`    mXValAverageLength = 1;`
		`    return;`
		`  }`
		`  float sum = 1 f;`
		`  for (int i = 0; i < mXVals.size(); i++) {`
		`    sum += mXVals.get(i).length();`
		`  }`
		`  mXValAverageLength = sum / (float) mXVals.size();`
		`}`

Second	construct a disjunction	`public DisjunctionSumScorer(Weight weight, List < Scorer > subScorers, int minimumNrMatchers) throws IOException {`
		`  super(weight, subScorers.toArray(new Scorer[subScorers.size()]), subScorers.size());`
		`  if (minimumNrMatchers <= 0) {`
		`    throw new IllegalArgumentException("Minimum-nr-of-matchers-must-be-positive");`
		`  }`
		`  if (numScorers <= 1) {`
		`    throw new IllegalArgumentException("There-must-be-at-least-2-subScorers");`
		`  }`
		`  this.minimumNrMatchers = minimumNrMatchers;`
		`}`

Third	display a representation of this estimator	`public String toString() {`
		`  String result = "Discrete-Estimator.-Counts-=-";`
		`  if (m_SumOfCounts > 1) {`
		`    for (int i = 0; i < m_Counts.length; i++) {`
		`      result += "-" + Utils.doubleToString(m_Counts[i], 2);`
		`    }`
		`    result += "-(Total-=-" + Utils.doubleToString(m_SumOfCounts, 2) + ")-";`
		`  } else {`
		`    for (int i = 0; i < m_Counts.length; i++) {`
		`      result += "-" + m_Counts[i];`
		`    }`
		`    result += "-(Total-=-" + m_SumOfCounts + ")-";`
		`  }`
		`  return result;`
		`}`

Table 4.2.5: Search example 5

Input query	how to read a csv file	
Snippet rank	**Original description**	**Code retrieved**

First	open a file for UNK decompressing it if necessary	`public static BufferedReader openFileForBufferedUtf8Reading(final File file) {`
		`return new BufferedReader(new InputStreamReader( openFileForReading(file), Charset.forName(" UTF-8")));`
		`}`

Second	load file	`private String loadFile(File file) {`
		`StringBuffer sb = new StringBuffer();`
		`char[] buffer = new char[1024];`
		`BufferedReader br = null;`
		`try {`
		`br = new BufferedReader(new FileReader(file));`
		`int readChar = br.read(buffer);`
		`while (readChar != -1) {`
		`sb.append(buffer, 0, readChar);`
		`readChar = br.read(buffer);`
		`}`
		`} catch (Exception e) {`
		`Logger.getInstance().logError("Error-loading- source-file", e);`
		`} finally {`
		`if (br != null) {`
		`try {`
		`br.close();`
		`} catch (IOException e) {`
		`Logger.getInstance().logError("Closing- stream", e);`
		`}`
		`}`
		`}`
		`return sb.toString();`
		`}`

Third	open a file with a specified name for input	`public static void readFile(String fileName) {`
		`if (fileName == null) readStandardInput();`
		`else {`
		`BufferedReader newin;`
		`try {`
		`newin = new BufferedReader(new FileReader( fileName));`
		`} catch (Exception e) {`
		`throw new IllegalArgumentException("Can\'t- open-file-");`
		`}`
		`if (!readingStandardInput) {`
		`try {`
		`in .close();`
		`} catch (Exception e) {}`
		`} in = newin;`
		`readingStandardInput = false;`
		`inputErrorCount = 0;`
		`inputFileName = fileName;`
		`}`
		`}`

Table 4.2.6: Search example 6

<table>
<tr><th>Input query</th><td colspan="2">create a file and write to it</td></tr>
<tr><th>Snippet rank</th><th>Original description</th><th>Code retrieved</th></tr>
<tr><td>First</td><td>dump the array list of times to the file</td><td><pre>public static void printStatsToFile(Statistics s,
    String filename) {
  try {
    File file = new File(filename);
    if (!file.exists()) {
      file.createNewFile();
    }
    FileWriter fw = new FileWriter(file.
        getAbsoluteFile(), true);
    BufferedWriter bw = new BufferedWriter(fw);
    bw.write(s.toString());
    bw.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
}</pre></td></tr>
<tr><td>Second</td><td>save the string buffer into a file</td><td><pre>private static void saveFile(StringBuffer
    stringBuffer, String filename) {
  String path = (new File("")).getPath();
  File file = new File(path + "/" + filename);
  FileWriter output = null;
  try {
    output = new FileWriter(file);
    output.write(stringBuffer.toString());
  } catch (IOException e) {
    e.printStackTrace();
  } finally {
    try {
      output.close();
    } catch (IOException e) {}
  }
}</pre></td></tr>
<tr><td>Third</td><td>write the tags from this exif interface object into a jpeg UNK removing prior exif tags</td><td><pre>public void writeExif(byte[] jpeg, String
    exifOutFileName) throws FileNotFoundException,
    IOException {
  if (jpeg == null || exifOutFileName == null) {
    throw new IllegalArgumentException(
        NULL_ARGUMENT_STRING);
  }
  OutputStream s = null;
  try {
    s = getExifWriterStream(exifOutFileName);
    s.write(jpeg, 0, jpeg.length);
    s.flush();
  } catch (IOException e) {
    closeSilently(s);
    throw e;
  }
  s.close();
}</pre></td></tr>
</table>

Table 4.2.7: Search example 7

Chapter 5

Conclusions and Future Work

In this chapter we present the conclusions of this project and the proposed future work.

5.1 Conclusions

In this project we have presented the challenge of code search. We have reviewed some of the state-of-the-art techniques in NLP, including Attention mechanisms, Transformers and BERT models. We have also shown some of the most popular techniques and approaches in code search.

The proposed experiments have demonstrated the effectiveness of BERT models for code search, as well as multistage architectures are presented as a solution to the high latency of monoBERT models with minimal impact on output quality. The results of our experiments have shown that even more naive and unspecialized models, such as in our case UNIF and SNN, can be useful in the first stage of the multistage architecture to filter K candidates, provided that it shows a good result with the Top$K$ metric.

In this project we studied also the robustness of the models, testing them with sentences from outside their test set. Although all models show a drop in performance with the new sentences, which could be mitigated by training them with more data, it is certain that monoBERT maintains its performance with the multistage architecture.

Finally, the search examples shown in Section 4.2 reveal that in most cases the code snippets retrieved are related to the input query and partially solve the question posed.

5.1.1 Future Work

This project tests the superiority of generic BERT models in code search compared to simpler models, as well as an architecture that mitigates their inference times. For this purpose, the dataset used contains cleaned code snippets, with subword splitting and removal of non-alphanumeric characters, allowing comparison between general purpose models.

One way to improve the results would be to keep the code intact and use models specialized in source code, such as codeBERT[10] or cuBERT[15]. In addition, since it has been proven that simple models can be used for initial candidate filtering, an architecture could be tested in which a UNIF-SNN model selects

candidates using the cleaned code, so that the specialized monoBERT (e.g: codeBERT or cuBERT) model performs the re-ranking with the original version of the snippet.

One interesting approach would be to combine general purpose BERT based models and models specialized in source code models in a Sentence-based architecture. Our Sentence-BERT approach uses the same BERT model to embed the code and description (using a Siamese architecture). An architecture using BERT for description and codeBERT or cuBERT for code snippet could leverage the strengths of the former in NLP and the latter in source code processing.

Another direct way to improve the absolute results of this project is to try longer sentence lengths. To make the proposed experiments feasible in terms of training time and available resources, the input to the models was truncated to 90 words. Although we do not expect major changes in the difference between models (monoBERT will continue to outperform the other models), we do believe that all will see their TopN metrics greatly improved.

Bibliography

[1] Baldi, Pierre and Chauvin, Yves. “Neural networks for fingerprint recognition”. In: *neural computation* 5.3 (1993), pp. 402–418.

[2] Bromley, Jane, Guyon, Isabelle, LeCun, Yann, Säckinger, Eduard, and Shah, Roopak. “Signature Verification Using a ”Siamese” Time Delay Neural Network”. In: *Proceedings of the 6th International Conference on Neural Information Processing Systems*. NIPS’93. Denver, Colorado: Morgan Kaufmann Publishers Inc., 1993, pp. 737–744.

[3] Brown, Tom B, Mann, Benjamin, Ryder, Nick, Subbiah, Melanie, Kaplan, Jared, Dhariwal, Prafulla, Neelakantan, Arvind, Shyam, Pranav, Sastry, Girish, Askell, Amanda, et al. “Language models are few-shot learners”. In: *arXiv preprint arXiv:2005.14165* (2020).

[4] Cambronero, Jose, Li, Hongyu, Kim, Seohyun, Sen, Koushik, and Chandra, Satish. “When deep learning met code search”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 964–974.

[5] Chicco, Davide. “Siamese neural networks: An overview”. In: *Artificial Neural Networks* (2021), pp. 73–94.

[6] Chollet, François et al. *Keras*. `https://keras.io`. 2015.

[7] Das, Arpita, Yenala, Harish, Chinnakotla, Manoj, and Shrivastava, Manish. “Together we stand: Siamese networks for similar question retrieval”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 378–387.

[8] Denil, Misha, Bazzani, Loris, Larochelle, Hugo, and Freitas, Nando de. “Learning where to attend with deep architectures for image tracking”. In: *Neural computation* 24.8 (2012), pp. 2151–2184.

[9] Devlin, Jacob, Chang, Ming-Wei, Lee, Kenton, and Toutanova, Kristina. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[10] Feng, Zhangyin, Guo, Daya, Tang, Duyu, Duan, Nan, Feng, Xiaocheng, Gong, Ming, Shou, Linjun, Qin, Bing, Liu, Ting, Jiang, Daxin, et al. “Codebert: A pre-trained model for programming and natural languages”. In: *arXiv preprint arXiv:2002.08155* (2020).

[11] Gan, Wee Chung and Ng, Hwee Tou. “Improving the robustness of question answering systems to question paraphrasing”. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 6065–6075.

[12] Goodfellow, Ian, Bengio, Yoshua, Courville, Aaron, and Bengio, Yoshua. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[13] Gu, Xiaodong, Zhang, Hongyu, and Kim, Sunghun. “Deep code search”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 933–944.

[14] Hoffer, Elad and Ailon, Nir. “Deep metric learning using triplet network”. In: *International workshop on similarity-based pattern recognition*. Springer. 2015, pp. 84–92.

[15] Kanade, Aditya, Maniatis, Petros, Balakrishnan, Gogul, and Shi, Kensen. “Learning and Evaluating Contextual Embedding of Source Code”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 5110–5121.

[16] Li, Wei, Qin, Haozhe, Yan, Shuhan, Shen, Beijun, and Chen, Yuting. “Learning Code-Query Interaction for Enhancing Code Searches”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2020, pp. 115–126.

[17] Lin, Jimmy, Nogueira, Rodrigo, and Yates, Andrew. “Pretrained transformers for text ranking: Bert and beyond”. In: *arXiv preprint arXiv:2010.06467* (2020).

[18] Liu, Xinchen, Liu, Wu, Mei, Tao, and Ma, Huadong. “Provid: Progressive and multimodal vehicle reidentification for large-scale urban surveillance”. In: *IEEE Transactions on Multimedia* 20.3 (2017), pp. 645–658.

[19] Liu, Yinhan, Ott, Myle, Goyal, Naman, Du, Jingfei, Joshi, Mandar, Chen, Danqi, Levy, Omer, Lewis, Mike, Zettlemoyer, Luke, and Stoyanov, Veselin. “Roberta: A robustly optimized bert pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).

[20] LLC, MultiMedia. *Stack Overflow*. URL: `https://stackoverflow.com` (visited on 06/04/2021).

[21] Lv, Fei, Zhang, Hongyu, Lou, Jian-guang, Wang, Shaowei, Zhang, Dongmei, and Zhao, Jianjun. “Codehow: Effective code search based on api understanding and extended boolean model (e)”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 260–270.

[22] Martins, Marcelo de Rezende and Gerosa, Marco A. “CoNCRA: A Convolutional Neural Network Code Retrieval Approach”. In: *arXiv preprint arXiv:2009.01959* (2020).

[23] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Yangqing, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[24] Merkel, Dirk. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[25] Meyer, Andre N, Barton, Laura E, Murphy, Gail C, Zimmermann, Thomas, and Fritz, Thomas. “The work life of developers: Activities, switches and perceived productivity”. In: *IEEE Transactions on Software Engineering* 43.12 (2017), pp. 1178–1193.

[26] Mikolov, Tomas, Chen, Kai, Corrado, Greg, and Dean, Jeffrey. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[27] Mikolov, Tomáš, Yih, Wen-tau, and Zweig, Geoffrey. “Linguistic regularities in continuous space word representations”. In: *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*. 2013, pp. 746–751.

[28] Minelli, Roberto, Mocci, Andrea, Lanza, Michele, and Baracchi, Lorenzo. “Visualizing developer interactions”. In: *2014 Second IEEE Working Conference on Software Visualization*. IEEE. 2014, pp. 147–156.

[29] Reimers, Nils and Gurevych, Iryna. “Sentence-bert: Sentence embeddings using siamese bert-networks”. In: *arXiv preprint arXiv:1908.10084* (2019).

[30] Rodriguez, Ariel et al. “Empirical study on the relationship between developer’s working habits and efficiency”. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE. 2018, pp. 74–77.

[31] Sachdev, Saksham, Li, Hongyu, Luan, Sifei, Kim, Seohyun, Sen, Koushik, and Chandra, Satish. “Retrieval on source code: a neural code search”. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2018, pp. 31–41.

[32] Sadowski, Caitlin, Stolee, Kathryn T, and Elbaum, Sebastian. “How developers search for code: a case study”. In: *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 2015, pp. 191–201.

[33] Singer, Janice, Lethbridge, Timothy, Vinson, Norman, and Anquetil, Nicolas. “An Examination of Software Engineering Work Practices”. In: *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON ’97. Toronto, Ontario, Canada: IBM Press, 1997, p. 21.

[34] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N, Kaiser, Lukasz, and Polosukhin, Illia. “Attention is all you need”. In: *arXiv preprint arXiv:1706.03762* (2017).

[35] Wolf, Thomas, Debut, Lysandre, Sanh, Victor, Chaumond, Julien, Delangue, Clement, Moi, Anthony, Cistac, Pierric, Rault, Tim, Louf, Rémi, Funtowicz, Morgan, Davison, Joe, Shleifer, Sam, Platen, Patrick von, Ma, Clara, Jernite, Yacine, Plu, Julien, Xu, Canwen, Scao, Teven Le, Gugger, Sylvain, Drame, Mariama, Lhoest, Quentin, and Rush, Alexander M. “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. URL: `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.

[36] Yao, Ziyu, Peddamail, Jayavardhan Reddy, and Sun, Huan. “Coacor: Code annotation for code retrieval with reinforcement learning”. In: *The World Wide Web Conference*. 2019, pp. 2203–2214.