Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Algorithm Learning from Data

Bachelor's Thesis

Frederik Markus

fremarkus@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Peter Belcák, Florian Grötschla
Prof. Dr. Roger Wattenhofer

July 28, 2022

Acknowledgements

I would like to thank my supervisors for their invaluable contributions and patience without whom this thesis would not have been possible. Their expertise and background was paramount in exploring this topic. I would also like to thank Justin Studer, the creator of the SynCoBERT, for his help in setting up his model on current hardware. I would like to thank Professor Wattenhofer for enabling me to do my thesis at his department. And finally, I want to thank the team at TIK for providing me with computational resources which enabled me to run all of the experiments in this work.

Abstract

We manipulate code snippets to make each word less expressive and thus obfuscate the idea behind an entire snippet while ensuring uniqueness of all identifiers. The reduction in the performance metric when mangling keywords in the code is observed across all models. This suggests that these models are not as semantically robust as one would desire and that the actual writing itself matters in generating good code. We found that the best performing model – in terms of absolute performance – was SynCoBERT. In terms of smallest relative performance drop between baseline and manipulated datasets, the best performing model was GraphCodeBERT.

Contents

Chapter 1

Introduction

In the past few years, Machine Learning has become a buzzword synonymous with progress and state-of-the-art technologies with potentially unlimited possibilities. One of these technological applications is Natural Language Processing, where the goal is to provide machines with the ability to understand speech and text in a manner akin to humans. One particular niche for this application is in the understanding of code.

1.1 Code Search

Code search is the task of finding suitable code from a dataset of code snippets. The program is provided with a natural-language query, and the program should provide suitable code options in return. The principle issue with this is that the program needs to return code that is semantically related to the query. In order to achieve this, it therefore needs to "understand" how the query and the code are structured, what their fundamental semantic structures are, and what the relationship between the natural-language query and the code snippet is. The "understanding" can be achieved with several different approaches, which leads to multiple models.

1.2 Model Robustness

We currently have plenty of frameworks that can tackle these challenges with varying degrees of accuracy. However, often these networks still rely on keywords and phrases that are used in the code snippet and the corresponding description. Hence, a key question that arises is to what extent the code can be obfuscated and how this affects the performance of the varying models in returning the correct code to a given query.

Chapter 2

Related Work

2.1 Models

The principle goal behind all models and encoders used is to generate a working function for any given description. The techniques used to achieve this vary from model to model. The next sections provide an overview of the models used as well as an outline of their inner workings.

2.1.1 CodeSearchNet

CodeSearchNet describes a range of different models, all of which are based on the same architecture. The difference lies in the encoding methods that each model uses.

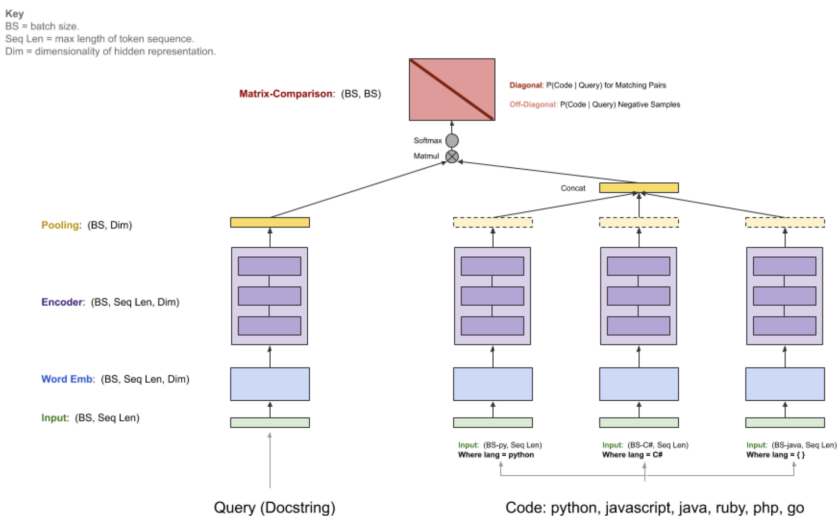

Figure 2.1: This figure presents the model architecture used in the CodeSearchNet. Taken from the CodeSearchNet GitHub repository [1].

Figure 2.1 illustrates the general architectural approach taken by CodeSearch-

Net. The model has two inputs: the query describing what the function is supposed to do and the code snippet. While the query has a single encoder, each programming language has its version. This approach is taken from earlier work [2, 3]. The idea is to use a joint embedding of code and query to implement a neural code search system. This is achieved by creating a map between each function snippet and the language it corresponds to and projecting this onto relatively close vectors. With this mapping achieved, a search method is implemented by creating an embedding of the query phrase and placing this in the same embedding space as the function embeddings. We can now return the code snippets close to the query in the embedding space (closeness in a hyper-dimensional space is defined through absolute distances between vectors). CodeSearchNet opted for this relatively simple implementation as it allows for quick, efficient indexing and searching as only a single vector has to be generated, even though more complex models, as presented in [3], have shown better results. The exact procedure is explained in detail in [4]: Each input sequence token is first preprocessed according to its semantics. This means the code tokens are split into subtokens, and natural language tokens are split using byte-pair encoding. These new tokens are processed to obtain token embeddings using one of the four model architectures:

- Neural Bag of Words: Here, each token (or subtoken) is embedded to a learnable vector representation
- Bidirectional RNN model: GRU cells, originally developed in [5], are employed to summarize the inputs. Note: This model was not used since there have been significant changes to the RNN layers in Tensorflow in the years since CodeSearchNet was first released and the original model no longer works.
- 1D Convolutional Neural Network: The model is applied to the input sequence of the tokens as per [6].
- Self-Attention: Multi-Head attention is used to compute representations of each token in the sequence, as used in [7]. A variant of this is the convolutional self-attention model, which is also employed in this study.

The token embeddings are combined into one sequence embedding using a pooling function (either mean or max pooling). The two sequences, one from the query and one from the code are then multiplied together to form a matrix over which a softmax is then applied to generate a comparison matrix with the correct function and query pair matching along the main diagonal. While training, the loss minimizing metric that is employed is defined as:

$$-\frac{1}{N}\sum_{i}\log\frac{\exp(E_c(\mathbf{c_i})^\intercal E_q(\mathbf{d_i}))}{\sum_j \exp(E_c(\mathbf{c_j})^\intercal E_q(\mathbf{d_i}))} \tag{2.1}$$

where N is the number of snippets, code and natural language description pairs are $(\mathbf{c_i}, \mathbf{d_i})$ and E_c, E_q are the code and query encoder respectively. The goal is therefore to maximize the inner product between each code snippet $\mathbf{c_i}$ and its respective description $\mathbf{d_i}$ while minimizing the distance to each distracting snippet $\mathbf{c_j}$ (where $i \neq j$).

2.1.2 CodeBERT

The CodeBERT architecture introduced in [8] significantly differs from CodeSearchNet in that it relies on a larger pretrained model, which is then fine-tuned for the task at hand. The idea behind this is that pretraining a model provides a "better general-purpose contextual representation" [8]. The advantage of the CodeBert model is that it can be used for different downstream tasks: Code search and code-to-text generation, which is something that CodeSearchNet cannot do. The CodeBERT architecture is based on the BERT model introduced in [9].

Pretraining

In contrast to the CodeSearchNet, CodeBERT can be trained with unimodal and bimodal data. Bimodal refers to data where both code and the associated description are available, while unimodal only provides the code. In the pretraining phase, the input is set as a concatenation of the natural language and the code segment, spliced by a special separator token *SEP*. The resulting input looks as follows: *[CLS]*,w_1,w_2,...,w_n,*[SEP]*,c_1,c_2,...,c_m,*[EOS]*, where w_i represents the i-th word piece and c_j the j-th code token. *[CLS]* represents a special token in front of the two segments that can be considered as an aggregated sequence representation for classification or ranking. *[EOS]* highlights the end of the string. Once CodeBERT has run, it will produce a contextual vector representation of each token for natural language and code as well as the representation for *[CLS]*. In addition, the framework also requires a pretraining step. This pretraining has two objectives:

- Masked Language Modeling (MLM): Random positions in the natural language and code tokens are chosen and replaced with a special *MASK* token. This is done for 15% of all tokens following [9]. The goal is then to predict the original tokens from a significantly sized vocabulary of bimodal data.
- Replaced Token Detection (RTD): Two data generators, one for natural language and one for code, are used to generate plausible alternatives for a group of randomly masked position. Based on [10], two efficient n-gram language models are created, both with bidirectional contexts. In contrast to the previous objective, both unimodal (for the code) and bimodal (for

the natural language) data can be used for training a discriminator. This discriminator is used to binarily classify whether a token has been corrupted or not.

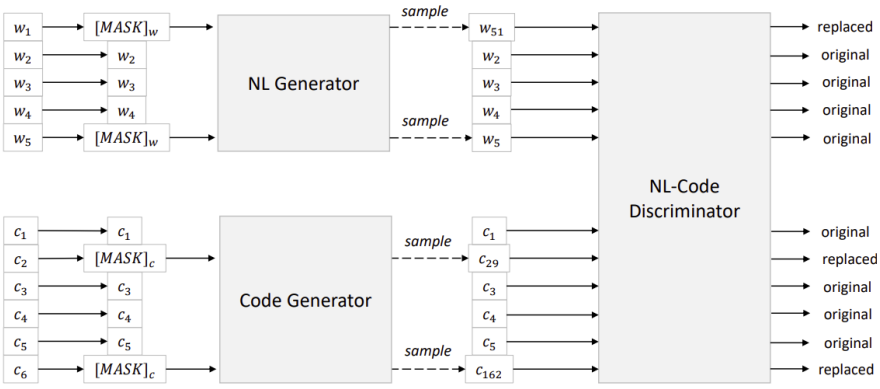

Figure 2.2: An illustration of the Replaced Token Detection approach. The two generators create a sensible replacement for the masked token. The aim is to train the discriminator, which is achieved through detecting plausible alternatives tokens, sampled from the two generators. Taken from [8].

Fine-Tuning and downstream tasks

Once pretraining has been completed, the model can be used to complete tasks further down the pipeline, including both code search and code-to-text generation. Both are relevant to us: code search is quintessential to the project, and code-to-text generation is used to help create the dataset for Typescript.
Code search on CodeBERT is similar to CodeSearchNet. We use the same evaluation metric and fine-tune a language-specific model for each programming language. Training occurs in a way not previously encountered. Each model is trained with a binary classification loss function. A softmax layer is linked to the CLS representation, which measures the semantic relevance between code and natural language query.
When using the code-to-text generation setting, we use an encoder-decoder framework and initialize the encoder of a generative model with CodeBERT. We use the smoothed BLEU score introduced in [11] as a metric.

2.1.3 GraphCodeBERT

GraphCodeBERT considers the inherent structure of code by leveraging its semantic-level information, known as data flow, during pretraining. As defined in [12], Data

flow is a graph in which nodes represent variables and edges represent the relation of "where-the-value-comes-from" between variables. Due to their less hierarchical nature, data flow graphs are usually less complex than syntactic representations, which include, for example, Abstract Syntax Trees. To generate the data flow and help the model learn the code representation from structure, two pretraining tasks are required: The first one is used to build the data flow graph for learning code structure representation, and the second one is used to align the representation between source code and code structure. The GraphCodeBERT model itself is based upon the Transformer neural architecture introduced in [7].

Data Flow Edge Prediction

Unlike an Abstract Syntax Tree, data flow diagrams are consistent across varying abstract grammars. This makes it easier to follow the semantics of code even when a variable is used in far-apart locations in the snippet. We first parse the code into an Abstract Syntax Tree to generate a data flow diagram. The leaves of the AST are used to identify the variable sequence. Each variable is chosen as a node, and directed edges are drawn between all related pairs of nodes (when the value of one variable is derived from another). The graph $G(C) = (V,E)$ is now the data flow graph consisting of nodes V and directed edges E used to represent all dependency relations between all variables in the source code C.
In the first pretraining task, we now randomly sample a fraction of the nodes in the data flow and mask all direct edges connecting the sampled nodes. This is achieved by adding an infinitely negative value to the mask matrix. The model then has to predict these masked edges. The idea behind this is to encourage the model to understand a structured representation of the code and where specific values are derived from.

Variable Alignment between Representations

The second pretraining task is used to encourage the model to align representations between the source code and the data flow. Here we predict edges between code tokens and nodes. This is achieved once again by first masking the edges between randomly selected nodes and code tokens and then predicting these masked edges.

Downstream Tasks

In a similar fashion to CodeBERT, GraphCodeBERT has a wide range of downstream applications. In [12], four downstream tasks are explored: code search, clone detection, code translation and code refinement. Relevant to our study is only code search.

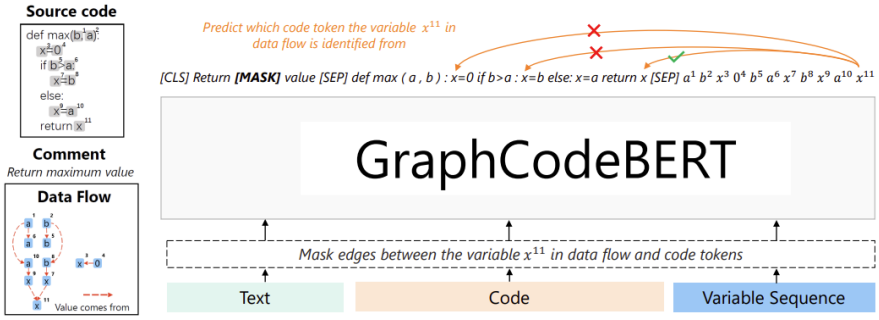

Figure 2.3: This figure illustrates an example of the node alignment. We first mask edges between variable x^{11} in the data flow and code tokens and subsequently predict which code token the variable in the data flow is identified from. The tick indicates that the variable is x^{11} is predicted from the variable x in "return x" based on the information in the data flow. Taken from [12]

. .

While the premise is the same, there are differences in execution between GraphCodeBERT and CodeSearchNet. While CodeSearchNet uses only 1000 candidate functions when testing, GraphCodeBERT extends its candidates to the entire function corpus, which is a sensible approach since it is closer to a real-life scenario. The evaluation metric is again chosen as MRR.

2.1.4 SynCoBERT

SynCoBERT, as developed in [13], is an amalgamation of multiple ideas not found in any other of the covered models. What primarily sets this model apart is the usage of Cross Momentum Contrastive Learning (xMoCo) [14], a framework that has been shown to function robustly with multi-modal data by employing multiple encoders. It evolved from the Momentum Contrastive Learning (MoCo) framework [15] and utilizes negatives samples more consistently by employing a dictionary of samples rather than just using in-batch samples. Building on this is the DyHardCode framework [16], which provides more meaningful negative samples that lead to more robust results. The conclusive step is using the Barlow Twins framework [17], a regularization step that does not rely on a contrastive learning objective. It aims at minimizing redundancy in the embedding features and benefits from larger embedding sizes than comparable contrastive learning models. The Barlow loss, defined as:

$$L_{Barlow} = \sum_i (1 - C_{ii})^2 + \lambda \sum_i \sum_{j \neq i} C_{ij}^2 \tag{2.2}$$

where λ is a positive constant trading off the importance of on-diagonal and off-diagonal terms and C is the cross-correlation matrix of the current embedding standardized and computed along the batch dimension, can be incorporated into the xMoCo framework as a regularization term multiplied with an appropriate weight hyperparameter. This leads to the following schematic for the overall framework, combining all discussed features:

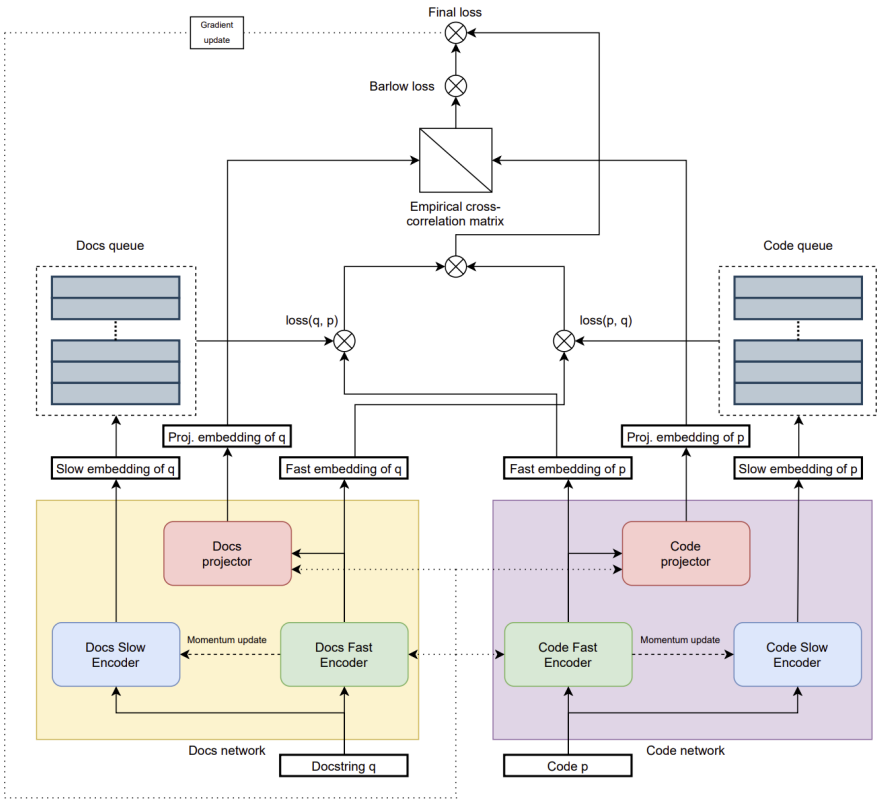

Figure 2.4: The complete framework combining xMoCo, DyHardCode and Barlow Loss. Taken from [13].

2.2 CodeSearch Dataset

All experiments are run using the cleaned version of the CodeSearch dataset, which was originally collected for the CodeSearchNet challenge in [4]. The original dataset is somewhat larger but contains certain elements that are not desired, which either leads to the function being changed or removed entirely from the dataset. This includes [18]:

- Remove functions that cannot be parsed into an Abstract Syntax Tree
- Remove functions with very few (less than 3) or very many tokens (more than 256)
- Remove functions that contain special tokens. These include but are not limited to <img ...> or https:...
- Remove functions where the description is not in English

The Typescript data has to be regarded in isolation since it was collected independently from the other languages. This yields:

PL	Train	Valid	Test
Python	251,820	13,914	14,918
Java	164,923	5,183	10,955
PHP	241,241	12,982	14,014
Go	167,288	7,325	8,122
Javascript	58,025	3,885	3,291
Typescript	4,408	1,407	N/A

Table 2.1: The dataset used for anonymization and running the models. Note that no Typescript test set is available.

Information concerning the original dataset can be found in the appendix. Applying the rules state above has decreased the usable data significantly, but ensures a higher quality of training and validation code.

2.3 Tree-Sitter

In order to be able to parse Typescript code correctly, we need to get its grammar. We use tree-sitter, a parser generator tool that builds syntax trees to do this. The tool is available for a wide range of languages [19], and while some are more mature than others, the languages we are considering are all reasonably complete. Furthermore, as the framework uses an incremental parser, it does not have to re-parse entire source files if something is edited. Since many GitHub repositories are constantly changing, this ability drastically reduces the time it takes to parse a file. It does this by marking the nodes that contain modified text as it walks the syntax tree. It then commences in an empty state and reuses the nodes of the previous tree that have not changed in the new tree. The tree-sitter library is mighty in that it allows one to utilize already existing parsers and facilitates the implementation of new parsers through extensive documentation, which is practical when trying to create new Codesearch datasets.

Chapter 3

Method and Procedure

A substantial amount of time was spent constructing parsers that allow us to anonymize varying identifiers of any given function. Moreover, an additional dataset was constructed from Typescript code snippets to expand the scope of this work.

3.1 Anonymization

In order to be able to assess the workings of the model, it is required to create a dataset that effectively hides the identifiers while still ensuring that the uniqueness of each identifier is maintained. To this end, it was decided to take each desired identifier and replace it with its hash (using SHA1, which is long enough to ensure that each identifier remains unique but not too long to unnecessarily inflate the file size) and a prefix. The prefix helps to keep an overview of the varying hashes while simultaneously providing very little information. The prefix is adjusted depending on the type of identifier: *fun* for functions, *var* for variables and *arg* for arguments. For example, while it is fairly clear what a function called "getdatetimesortedrows" aims to do, it is no longer obvious what a function called: "fun6e094284956ce52611ccd71d6ced854add7dd57c" will achieve. The chosen anonymization only mangles function and variable names, but the code has also been written to allow one to anonymize arguments. To generate these transformations, we largely rely on Abstract Syntax Trees (AST), where the tree is a way of representing the syntax of a programming language as a hierarchical tree-like structure [20]. This allows us to access individually nodes that are part of a certain construct of the tree. In principle, as long as an AST can be formed, any uniquely identifiable node can be "visited" and modified. If an AST cannot be formed due to technical limitations, other approaches must be chosen, as shown in the case of PHP.
With a cursory knowledge of ASTs, we can now lay out the basic pipeline for the anonymization process:

1. Bring the data into a readable form. For some programming languages this is already the case (Python) but for others some pre-processing is required chiefly due to the fact that original dataset is saved as .jsonl files (JSON lines). As such, for most programming languages the dataset was placed in .csv files, which allowed for easy accessibility later.

2. Next, isolate the "code" column from the dataset, containing the extracted functions and construct the AST for each function individually.

3. Define and implement a visitor class or method and use it to traverse the AST until the desired node is reached. The visitor class used is mostly an extension of a base class.

4. Transform the AST node names to their respective hashes, bearing in mind to also add a prefix to enable easier identification.

5. Recreate the function text from the AST and replace the original piece of code in the dataset with the newly anonymized one.

6. Return the dataset to the .csv format if required.

7. Open the dataset in Python, remove comments (if not already done so during the AST construction) and tokenize. To standardize the tokenization across all datasets, and not rely on the individual tokenizers from each language, which may parse single characters or brackets differently, all tokenizations were done in Python. Replace the new tokens in the dataset and save the anonymized and compressed file.

While an initial consideration was to write the anonymization scripts for all programming languages in Python, it turned out that these Python-based parsers often had severe deficiencies with respect to the amount of information contained within a single node and could therefore be usefully extracted. For example, a parser for Javascript that was written in Python identified nodes correctly as an "identifier". However, there was no additional information about what kind of "identifier" a node was, which is undoubtedly an essential piece of information. Thus, the decision was made to write each anonymizing script in the original language whenever possible.
One issue that occurs when building the AST is the fact that the parser cannot read all functions, either because they are written with outdated syntax or due to a parsing issue. This was most notable in Java, where several functions with missing closing brackets were present. Overall, however, these improper functions only made up a negligible part of each dataset, resulting in anonymization rates of between 98.5% and 100% for all datasets.

3.1.1 Python

As the GitHub repository for the CodeSearchNet challenge was written in Python, this provided a good starting point as the CodeSearchNet repository already had some supporting functions implemented. The anonymization procedure follows the basic pipeline described above. After reading the dataset from the .jsonl files, the comments are removed. This is done using regular expressions, which proved easier than individually identifying comment tokens. After building the AST and using the self-written visitor class to find and replace all desired node names, the function text is recreated, and the tokens are formed. These are then saved to the dataset, which is then saved to the desired data file type. This procedure is repeated for each dataset chunk.

Example Anonymization

This example anonymization illustrates the key aspects of the procedure:

```
def _check_series_localize_timestamps(s, timezone):
    """
    Convert timezone aware timestamps to timezone-naive in the specified
        timezone or local timezone.
    If the input series is not a timestamp series, then the same series is
        returned. If the input
    series is a timestamp series, then a converted series is returned.

    :param s: pandas.Series
    :param timezone: the timezone to convert. if None then use local timezone
    :return pandas.Series that have been converted to tz-naive
    """
    from pyspark.sql.utils import require_minimum_pandas_version
    require_minimum_pandas_version()
    from pandas.api.types import is_datetime64tz_dtype

    tz = timezone or _get_local_timezone()
    # TODO: handle nested timestamps, such as ArrayType(TimestampType
        ())?
    if is_datetime64tz_dtype(s.dtype):
        return s.dt.tz_convert(tz).dt.tz_localize(None)
    else:
        return s
```

Using the stated pipeline, we can turn this normal code snippet into its anonymized version:

```
def fun79a06b59(s, timezone):
    from pyspark.sql.utils import require_minimum_pandas_version
    require_minimum_pandas_version()
    from pandas.api.types import is_datetime64tz_dtype

    var1412349a = timezone or _get_local_timezone()
    if is_datetime64tz_dtype(s.dtype):
        return s.dt.tz_convert(var1412349a).dt.tz_localize(None)
    else:
        return s
```

The example above illustrates the effect of anonymization, using an example from the python dataset. It provides a good overview of the type of steps that are taken. The hashes that were formed have been shortened slightly for readability. We can see that both multi-line and single-line comments are removed from the snippet. The function name is anonymized, but simultaneously, function calls contained in the body, here in the form of *_get_local_timezone()*, are not changed (unless they are a recursive call). Import statements and standard function calls are not changed either. Throughout all datasets, we have decided only to change function names and variables. This importantly leaves out arguments, which are not touched on here. So the two arguments *s* and *timezone* are not changed in the argument of the function or the body.

3.1.2 Java

The Java anonymization script was written with the help of an IDE, which provides an excellent debugger that helps enormously with identifying the correct nodes to visit in the AST. In order to utilize the parser, each function had to be turned into a class which was accomplished by surrounding the existing function string with a temporary "foo class" that was removed at the end. After that, the remainder of the basic pipeline described above was followed precisely.

3.1.3 Javascript

The Javascript anonymization script was also written in an IDE. The pipeline above was followed with a few exceptions: Javascript has a special feature where functions can have no name, known as anonymous functions. This will throw an error while parsing into an AST as the parser requires a distinct function name to work correctly. To counter this, whenever an anonymous function was identified, a temporary name "foo" was inserted to allow the parser to function correctly. This "foo" function name was then removed at the end of the anonymization procedure to return an anonymous function as in the original snippet.

3.1.4 Go

For Go, the standard pipeline was followed with the small exception that comments could be removed using the parser directly. This somewhat improved run time, as the construction of the Syntax Tree took less time.

3.1.5 PHP

PHP is an outlier because the anonymization procedure deviated significantly from the basic pipeline described before. The chosen IDE (PhpStorm) struggled significantly with creating ASTs, mainly since the dataset contained code pertaining to varying versions of PHP, not all of which could be read and parsed by the standard PHP parser. Using different parsers did not yield any better results, so it was decided to modify the tokens directly instead. This was possible due to the fact that every variable in PHP requires a $ in front of it. This permitted us to step through the tokens and replace every token with its hash if a dollar sign is preceding it. Since the function name is always the token after the token "function", it was also quite straightforward to identify and anonymize the function name (as well as any recursive iterations of it). To identify and replace the argument tokens, it was necessary to find the first opening bracket after the function name, look for the revealing dollar sign and finally replace the subsequent token. Once each desired token was replaced, the tokens were reunited to form the function text. To do this, heuristics were applied since, most of the time, tokens should be placed next to each other with a space in the middle. Sometimes, there are cases where two tokens should be placed next to each other, for example, when a method of a class is called via the dot or arrow operator. However, since it is impossible to foresee every possible combination that someone may use in their code, best practices were used that work for the vast majority of cases:

- If a token has length one, do not place any spaces around it so that the previous and next token form one contiguous word with the one character token.
- If a token is of the desired identifying type (identified through a flag set whenever that particular token is encountered), do not place a space behind it. If the next character is one character long or of special nature (described below), no space is required behind the current token, and it does not destroy the functionality if the next token is not of the described nature.
- If a token of special nature is identified and the previous token is not of the type variable, strip the last character currently written in the code string (which will be a space) and do not place a space after the next one. The tokens of special nature are "::" and "->". These tokens are special because

they should form one contiguous token string with their surrounding tokens but are also longer than one character. If the previous token is a variable, do not strip the last character; otherwise, proceed as usual.

Removing comments using the tokens requires very little information. Since all comments in the dataset are written as a single token, all that is required is to identify tokens that start with "//" and do not end with "//" and delete them. The requirement at the end of the token is necessary because specific identifiers in PHP have this syntax but do not act as comments. Once the heuristics are applied to generate the function text, the dataset can be saved as described in the basic pipeline.

Below is an example to more clearly visualize the special case of PHP anonymization. The sample function is taken from the original Codesearch dataset:

```
final public function handle(RequestInterface $request){
    $processed = $this->processing($request);
    if ($processed === null) {
        // the request has not been processed by this handler => see the
            next
        if ($this->successor !== null) {
            $processed = $this->successor->handle($request);
        }
    }
    return $processed;
}
```

The corresponding tokenization for the sample function is:

["final", "public", "function", "handle", "(", "RequestInterface", "$", "request", ")", "{", "$", "processed", "=", "$", "this", "->", "processing", "(", "$", "request", ")", ";", "if", "(", "$", "processed", "===", "null", ")", "{", "// the request has not been processed by this handler => see the next", "if", "(", "$", "this", "->", "successor", "!==", "null", ")", "{", "$", "processed", "=", "$", "this", "->", "successor", "->", "handle", "(", "$", "request", ")", ";", "}", "}", "return", "$", "processed", ";", "}"]

We can see that tokenization handles all tokens except comments as expected. Instead of being split into their individual words, comments are taken as a single token, which simplifies their removal. Next, the tokens themselves are considered and modified. We can compare the anonymized tokenization.

The corresponding anonymized tokenization is:

["final", "public", "function", "funa2dd7ec6", "(", "RequestInterface", "$", "request", ")", "{", "$", "var46c7abc9", "=", "$", "varc2543fff", "->", "processing", "(", "$", "request", ")", ";", "if", "(", "$", "var46c7abc9", "===", "null", ")", "{", "if", "(", "$", "varc2543fff", "->", "successor", "!==", "null", ")", "{", "$", "var46c7abc9", "=", "$", "varc2543fff", "->", "successor", "->", "funa2dd7ec6", "(", "$", "request", ")", ";", "}", "}", "return", "$", "var46c7abc9", ";", "}"]

The comment has been removed, and all variables and function names have been replaced while simultaneously neglecting the arguments throughout the code. The names have once again been shortened for readability. From the tokenizations, we can reconstruct the new, anonymized sample function that applies the stated heuristics:

```
final public function funa2dd7ec6(RequestInterface $request ){
    $var46c7abc9=$varc2543fff->processing($request);
    if ($var46c7abc9=== null ){
        if ($varc2543fff->successor !== null ){
            $var46c7abc9=$varc2543fff->successor->funa2dd7ec6 ($request);
        }
    }
    return $var46c7abc9;
}
```

We can see the effect of the applied heuristics: all arrows, -> are placed in one contiguous string, and other single character tokens, such as $, are placed with the next token. This is vitally important in the case of $ as it uniquely identifies variables and arguments in a PHP function. In the sample function above, we can also see a recursive function call being anonymized when handle($request) is called. The general rule here is that recursive function calls are anonymized while calls to other functions are not.

3.1.6 Typescript

Typescript first gets compiled to Javascript and then executed. The approach was to manipulate the compiled Javascript code since a quasi-AST representation could be generated by loading the code from a file. While the quasi-AST still maintains all the features of an AST, we call this a quasi-AST because the structure is generated in Javascript using Javascript syntax to describe the Typescript function and not, as in other cases, in the native programming language. This yields a node object that could be modified using Javascript. Since no AST representation exists, there is no visitor class that could be expanded. So instead, the approach was to identify the desired nodes and modify the function text to

the desired effect. This does not work with certainty as the quasi-AST representation can sometimes be thrown due to the complexity of the code provided. Thus, it was again decided to use heuristics to ascertain where the desired nodes should be located in the quasi-AST and modify them accordingly. Once this anonymization step is completed, the Typescript code can be processed similarly to the Javascript code described above.

3.2 Dataset Construction

In addition to the dataset anonymization, the idea to expand the default dataset was quickly brought forward. The libraries to do this were already provided in the CodeSearchNet repository. The function parser library provided is already quite an extensive framework, with the central issue being the code being limited to six languages. Expanding this library is facilitated by the fact that we can quite easily import the grammar for the desired language from the tree-sitter repository [19] in order to be able to parse that language. Building on this, we are then required to add visitor functionality that can retrieve the code snippet, docstring, and all other relevant metadata from the input. Doing this can be quite troublesome for languages with convoluted and complex syntax, such as C++.

3.2.1 Typescript

Typescript offered itself as a natural candidate since it is syntactically very similar to Javascript. As such, many of the tools used for Javascript can be applied to Typescript. The initial step was to scrape repositories on GitHub that contained Typescript code. A list containing Typescript code was first compiled using a GitHub scraper tool [21] and then passed to the function parser library. This library takes all Typescript functions found on the individual repositories and combines them to form a dataset similar to the existing CodeSearchNet dataset. One major issue that was encountered when collecting this data was the fact that about 90% of functions that were scraped did not have an associated descriptive docstring. It was attempted to mend this by using CodeBERT's code2nl library. Naturally, this requires training data where both docstring and function are available. Since not enough Typestring data was available to train the model, it was decided to fine-tune it using the much larger Javascript dataset since the two languages are syntactically very similar. It was also attempted to fine-tune using the few available Typestring functions, but this did not improve the outcome. The final step was then to predict the missing docstrings. The whole dataset was split into training and validation sets which were then used to train the varying models.

3.3 Dataset Formatting

In order to be able to run the models properly, some data formatting was required. To ensure the same standards across all models, all datasets provided to the varying models are derived from the standard CodeSearchNet datasets, which are saved as chunked, gzipped, .jsonl files. Using chunked files can reduce the memory requirements when loading the dataset. It is highly advisable to do this, especially on lower-end machines, as it could otherwise be possible that there is not enough memory to load the entire dataset at once. This is especially noticeable for the larger datasets. With the cleaned datasets, this was no longer a problem. The list below indicates what kind of data format is required for each model:

- CodeBERT: The CodeBERT dataset requires input IDs for each function necessary to generate the CLS representation. This is significant because no mention is made in [8] on how exactly this input ID is generated for the dataset. Due to this reason, the CodeBERT dataset was downloaded independently from the other datasets. Furthermore, it also seems that the dataset did not have its comments removed, so some cleaning of the dataset was done using the rules above. The data format for this model is a single .txt file.
- GraphCodeBERT: The GraphCodeBERT dataset requires one single .jsonl file per partition and .txt files containing the URLs linking to the functions. Furthermore, the dataset contains a "codebase" file which contains all validation and testing functions (so the actual testing and validation files have empty spaces in their respective "code" columns).
- SynCoBERT: The SynCoBERT dataset requires a single .jsonl file per partition for train, valid and test data.

Chapter 4

Experiment Details

4.1 Server and Shell Scripting

All experiments were conducted on the TIK Arton cluster at ETH Zurich. This cluster houses several high-end graphics cards ranging from Tesla K40C's on the lower end of the performance spectrum to Nivida Geforce RTX 3090's on the cutting edge. While interactive sessions on each graphics card can be conducted, the recommended way of running an experiment is through job scripts. These are shell scripts that can be used to specify numerous parameters pertaining to the type and number of graphics cards to use, the location of the error and output log files, the size of the memory being allocated, and several others.
The most significant element of each job script is, of course, running the particular model that is to be trained and evaluated. The exact commands naturally vary from model to model, so it was decided to create a separate folder and associated job script for each model. The exact specifications for each model can be found in Appendix tables A.1 to A.6. This facilitated debugging the individual setups and allowed for a simplified overview of all individual runs.

4.2 Environments

All models are written in Python and primarily based on the PyTorch or the Tensorflow Machine Learning frameworks. In addition, all models are run in their own Conda Environments, which are tailored specifically to their dependencies. This took time to set up since the instructions were not always completely up-to-date and sometimes left critical dependencies unspecified (when no specific version was provided) or unmentioned (the package was not mentioned in the notes but then threw an error when trying to run the model). As such, significant care was required to ensure that all packages were set up as needed.

4.3 GPU specification

All models except SynCoBERT were run on whatever GPU was available at the time since the computation time was within the two-day limit specified by the cluster regardless of GPU. For SynCoBERT, four Nvidia Geforce RTX 3090 GPUs were used for training. If this setup is not available, it is advisable to set down the batch size; otherwise, the GPU memory may not be large enough to handle the data. Furthermore, if there are time constraints concerning the runtime on the server, it is advisable to reduce the number of training epochs. The number of GPUs used for training varied and was based on the documentation of each model.

Chapter 5

Results

5.1 Comparison to stated scores

The results stated in the CodeSearchNet challenge paper [4] could be confirmed and were even improved upon in some cases. The paper does not precisely state what hyperparameters were selected when conducting the runs on the model, so all models were trained using the default parameters suggested. For GraphCodeBERT, the stated baselines scores [12] were met quite precisely. The SynCoBERT scores were marginally lower than those stated in [13] across the board, suggesting that tweaking the hyperparameters may be beneficial. For CodeBERT, the stated baselines [8] scores could also be matched closely.

Model	Python		Java		Javascript		PHP		Go	
	Base	Anon	Base	Anon	Base	Anon	Base	Anon	Base	Anon
NBoW	0.634	0.507	0.660	0.013	0.399	0.008	0.684	0.612	0.802	0.030
SelfAtt	0.634	0.491	0.643	0.172	0.445	0.054	0.693	0.521	0.871	0.293
ConvSelfAtt	0.622	0.456	0.639	0.265	0.342	0.096	0.697	0.553	0.873	0.530
1D-CNN	0.505	0.256	0.537	0.147	0.158	0.021	0.609	0.360	0.801	0.250
GraphCodeBERT	0.694	**0.694**	0.690	0.539	0.643	0.431	0.647	**0.646**	0.895	**0.807**
CodeBERT	0.677	N/A	0.675	0.523	0.619	0.384	0.629	0.596	0.882	0.656
SynCoBERT	**0.718**	0.685	**0.761**	**0.630**	**0.681**	**0.439**	**0.701**	**0.646**	**0.929**	0.786

Table 5.1: The table above presents the MRR scores of all models across all examined languages. The first column in each language represents the baseline score, while the second column represents the anonymized version where the function name and all variables have been anonymized.

5.2 Effect of Anonymization

The effects of anonymizing the dataset are reflected in the models' resulting Mean Reciprocal Rank (MRR) scores. Across all languages and all models, the MRR score of the anonymized dataset is smaller than its baseline counterpart. The differences vary from model to model and language to language, but a clear trend can be observed across the board.

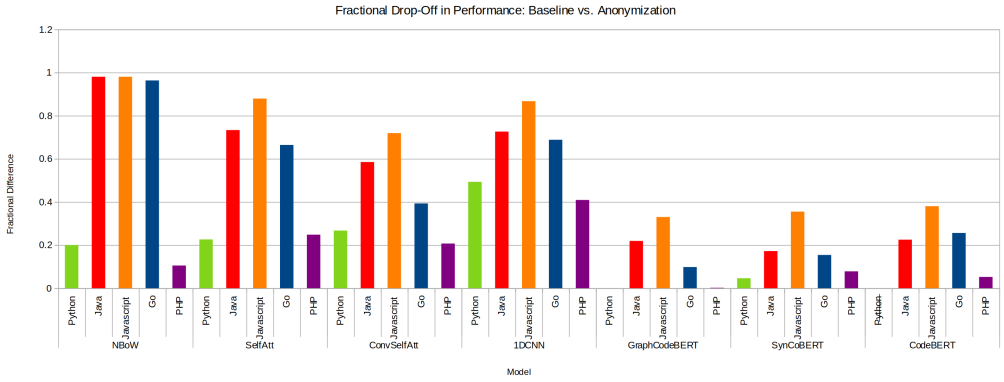

Figure 5.1: This plot illustrates the normalized relative loss between the baseline and the anonymized scores.

The values in Figure 5.1 are calculated from Table 5.1 as:

$$\frac{MRR_{base} - MRR_{anon}}{MRR_{base}} \tag{5.1}$$

A lower score in Figure 5.1 thus equals better robustness to anonymization.

5.3 Typescript dataset

The Typescript dataset has to be considered separately from the other models because it does not meet the standards of the dataset in terms of size and quality of descriptions. This is reflected in the validation scores in Table 5.2: the MRR scores are significantly lower than for the other languages across all models.

Dataset / Model	Baseline	Anonymized
NboW	0.025	0.010
SelfAtt	0.077	0.057
ConvSelfAtt	0.067	0.052
1D-CNN	0.025	0.016
GraphCodeBERT	0.044	0.031
SynCodeBERT	0.020	0.013

Table 5.2: Typescript dataset validation scores

We must note that the dataset was not run on the CodeBERT model due to a lack of available input IDs. For all other languages, these input IDs are available when downloading the dataset. However, no mention is made in [8] on how to appropriately generate these IDs; thus, the model was avoided.

Chapter 6

Conclusion and Outlook

6.1 Conclusion

Based on the results, it is clear that SynCoBERT shows the best performance across almost all languages for both the baseline score and the anonymized dataset. This suggests that using bimodal encoders is beneficial for code search and also shows that pre-training, as done in CodeBERT and GraphCodeBERT, is not required to achieve state-of-the-art performance.
However, one must differentiate between state-of-the-art performance and the relative drop between baseline and anonymized scores. Figure 5.1 illustrates that while SynCoBERT performs best across nearly all datasets, except for the anonymized versions of Python and Go, GraphCodeBERT's relative performance loss between baseline and anonymized datasets is better in all but one case. This suggests that GraphCodeBERT is more robust to anonymization, which implies that a graph-based approach can be, semantically speaking, sensible.
We can also see from Figure 5.1 that PHP and Python are the two most robust languages. They have the smallest performance drop across all models, and, in the case of GraphCodeBERT, Python even maintains its MRR score. This is frankly an extraordinary result since it suggests that the "understanding" of the model is not based on the phrasing or wording of any particular identifier but rather solely on its semantic role in the function snippet.
We can also say that sufficient training data is vital in order to achieve sensible scores. In the case of Typescript, there were not enough collected data pairs to create a baseline score that resembles the other languages. Therefore, it does not warrant a comparison to them. The collected samples were also not very good at predicting docstrings using the code2nl library. There were a lot of repetitive descriptions and incomplete sentences. Nevertheless, the difference between its baseline and the anonymized scores persists, suggesting that even with little data, the effects of anonymization can still be seen.
The unavailable Python anonymization for CodeBERT also needs to be highlighted. The CodeBERT dataset uses a tokenized version of the code. This is troublesome because it means that functions are not adequately recognized. In order to mend this, it was attempted to recombine the single character tokens

in a manner akin to the method presented for PHP anonymization, but this did not fix the parsing error. Furthermore, since there is no way of finding particular identifiers, as was the case for PHP, this effectively makes the dataset "unanonymizable".

6.2 Outlook

6.2.1 Model Selection

One could employ other models to see how they fare against the already presented models. One notable example is GraphSearchNet [22], which was briefly considered another model to be run but was then discarded due to several issues with the code and a lack of technical support on the side of the author. GraphSearchNet was also limited in its linguistic abilities since the code only works for Java and Python. Perhaps, future work could look to implement this graph-based approach for other programming languages as well.
Another model that could be regarded is the UnixCoder framework [23], which evolved recently from the CodeBERT model. This framework has been shown to perform favorably compared to CodeBERT, GraphCodeBERT, and SynCoBERT. Furthermore, it has also tested well against other datasets, namely AdvTest [24] and CosQA [25], which indicates good generalization performance.

6.2.2 Programming Languages

For this project, the six languages in the original CodeSearchNet dataset were enhanced by one additional language in the form of Typescript. Even though Typescript is its own language, it is semantically very similar to Javascript. Perhaps the dataset could be further enhanced by including other languages that deviate more significantly in a grammatical sense. While it is theoretically possible to add any language to the dataset, in practice, it is advisable to utilize and expand the already existing libraries. This means it is sensible to choose a programming language for which the tree-sitter grammar [19] already exists and can be downloaded from the respective repository. The sole remaining difficulty then lies in implementing a node search to get the desired functions, function metadata, and the associated description from the chosen GitHub repository.
One issue that should also be kept in mind is the availability of GitHub code for training. There is no point in assorting a dataset if there are not enough functions and docstrings on GitHub. Even with enough functions, it is still necessary to also have enough associated docstrings. Using the code2nl library can help generate descriptions for some functions; however, the quality of these cannot be guaranteed and can often be arbitrary and bear no resemblance to the function. While it is possible to source code from other, private, projects, this option is, of

course, not available to everyone and can also not be reproduced by third parties, so this step is not recommended.
Another consideration that must be taken into account is that an anonymization tool must be constructed for the language. For this to be possible, an AST must be created, which is not possible in all programming languages (in which case another workaround must be found, as demonstrated for Typescript). Before making the effort of building a dataset, it should therefore be ensured that there are tools to build an AST for the chosen language.

6.2.3 Expansion of Existing Languages

One considerable issue facing particularly the Typescript dataset is the small size (just over 5800 functions), even though a significant number of the most popular Typescript repositories (515) on GitHub were scraped. This is because a vast number of potential functions are what would be called "anonymous" functions in Javascript, meaning that they do not have a function name but possess a function body. The issue with this is that the grammar does not recognize these anonymous functions as functions, and as such, they were not picked up by the visitor class. An attempt to rectify this by visiting the respective nodes was problematic because it led to other undesired variable declarations being picked up. It is somewhat surprising that so few “actual” Typescript functions are being scraped from GitHub since the same procedure was applied for the other languages, and the number of functions that were scraped for those languages was much more significant.
Future work could perhaps look at rectifying this issue more comprehensively to generate more useful datasets for Typescript and other languages. Only then could a fair comparison between the already established and the new languages be done.

Bibliography

[1] "Codesearchnet model architecture Link." [Online]. Available: https://github.com/github/CodeSearchNet/blob/master/images/architecture.png

[2] X. Gu, H. Zhang, and S. Kim, "Deep code search," *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 933–944, 2018.

[3] B. Miutra and N. Craswell, "An introduction to neural information retrieval," *Found. Trends Inf. Retr.*, vol. 13, no. 1, p. 1–126, dec 2018. [Online]. Available: https://doi.org/10.1561/1500000061

[4] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019. [Online]. Available: http://arxiv.org/abs/1909.09436

[5] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *CoRR*, vol. abs/1409.1259, 2014. [Online]. Available: http://arxiv.org/abs/1409.1259

[6] Y. Kim, "Convolutional neural networks for sentence classification," *CoRR*, vol. abs/1408.5882, 2014. [Online]. Available: http://arxiv.org/abs/1408.5882

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[8] 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," *CoRR*, vol. abs/2002.08155, 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[9] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[10] D. Jurafsky, "Speech and language processing," 2000.

[11] C.-Y. Lin and F. J. Och, "ORANGE: a method for evaluating automatic evaluation metrics for machine translation," in *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*. Geneva, Switzerland: COLING, aug 23–aug 27 2004, pp. 501–507. [Online]. Available: https://aclanthology.org/C04-1072

[12] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," *CoRR*, vol. abs/2009.08366, 2020. [Online]. Available: https://arxiv.org/abs/2009.08366

[13] J. Studer, "Contrastive learning for programming languages," 2021. [Online]. Available: https://pub.tik.ee.ethz.ch/students/2021-HS/BA-2021-25.pdf

[14] N. Yang, F. Wei, B. Jiao, D. Jiang, and L. Yang, "xmoco: Cross momentum contrastive learning for open-domain question answering," 2021. [Online]. Available: https://aclanthology.org/2021.acl-long.477.pdf

[15] K. He, H. Fan, Y. Wu, S. Xie, and R. B. Girshick, "Momentum contrast for unsupervised visual representation learning," *CoRR*, vol. abs/1911.05722, 2019. [Online]. Available: http://arxiv.org/abs/1911.05722

[16] Anonymous, "Contrastive learning of natural language and code representations for semantic code search," 2021. [Online]. Available: https://openreview.net/forum?id=eiAkrltBTh4

[17] J. Zbontar, L. Jing, I. Misra, Y. LeCun, and S. Deny, "Barlow twins: Self-supervised learning via redundancy reduction," *CoRR*, vol. abs/2103.03230, 2021. [Online]. Available: https://arxiv.org/abs/2103.03230

[18] "Graphcodebert github repository Link." [Online]. Available: https://github.com/microsoft/CodeBERT/tree/master/GraphCodeBERT/codesearch

[19] "Tree-sitter documentation Link." [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[20] "Abstract syntax tree Link." [Online]. Available: https://www.techopedia.com/definition/22431/abstract-syntax-tree-ast

[21] "Github scraper Link." [Online]. Available: https://github.com/khuyentran1401/top-github-scraper

[22] S. Liu, X. Xie, L. Ma, J. Siow, and Y. Liu, "Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search," *arXiv preprint arXiv:2111.02671*, 2021.

[23] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," 2022. [Online]. Available: https://arxiv.org/abs/2203.03850

[24] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[25] J. Huang, D. Tang, L. Shou, M. Gong, K. Xu, D. Jiang, M. Zhou, and N. Duan, "Cosqa: 20, 000+ web queries for code search and question answering," *CoRR*, vol. abs/2105.13239, 2021. [Online]. Available: https://arxiv.org/abs/2105.13239

[26] "Codesearch full dataset, Link." [Online]. Available: https://github.com/github/CodeSearchNet/blob/master/notebooks/ExploreData.ipynb

Appendix A

Hyperparameters for models

The used hyperparameters for all models are listed below:

A.1 CodeSearchNet

max num epochs	300
test batch size	1000
distance metric	cosine

Table A.1: CodeSearchNet hyperparameters

Specify *do_test* and *do_eval* or *do_train* for testing or training respectively. The remaining hyperparameters are identical between the two.

A.2 GraphCodeBERT

num training epochs	10
code length	256
data flow length	64
nl length	128
train batch size	32
eval batch size	64
learning rate	$2 * 10^{-5}$

Table A.2: GraphCodeBERT hyperparameters

Specify *do_test* and *do_eval* or *do_train* for testing or training respectively. The remaining hyperparameters are identical between the two.

A.3 CodeBERT

model type	roberta
task name	codesearch
max seq length	200
per gpu train batch size	32
per gpu eval batch size	32
num train epochs	10
learning rate	10^{-5}
gradient accumulation steps	1

Table A.3: CodeBERT hyperparameters for Training

model type	roberta
task name	codesearch
max seq length	200
per gpu train batch size	32
per gpu eval batch size	32
num train epochs	8
learning rate	10^{-5}

Table A.4: CodeBERT hyperparameters for Testing

A.4 SynCoBERT

After training is completed it can occur that the model continues to run if the testing flag has been raised. While this will lead to an error, it will not crash the model. As such it is necessary to stop the script and re-run it with the following specifications:

effective queue size	4096
effective batch size	32
learning rate	10^{-5}
num hard negatives	2
debug data skip interval	1
num workers	2
always use full val	*set as parameter*
num epochs	6
language	*specify desired language*
checkpoint base path	*specify base path*
checkpoint name	*specify checkpoint name*
generate checkpoints	*set as parameter*
shuffle	*set as parameter*

Table A.5: SynCoBERT hyperparameters for training. Note: for the large datasets, Python and PHP, the training gets quite close to the 48-hour limit on the TIK Arton cluster, especially when the cluster is busy. As such it may be advisable to increase the batch size slightly to accommodate these datasets.

effective queue size	4096
effective batch size	32
learning rate	10^{-5}
num hard negatives	2
debug data skip interval	1
num workers	2
always use full val	*set as parameter*
num epochs	6
language	*specify desired language*
checkpoint base path	*specify base path*
checkpoint name	*specify checkpoint name*
do test	*set as parameter*
skip training	*set as parameter*
shuffle	*set as parameter*

Table A.6: SynCoBERT hyperparameters for testing. Note that setting $--skip\_training$ will make most of the above set training parameters redundant.

Appendix B

CodeSearch Dataset

The original CodeSearch data corpus. [26]

PL	Train	Valid	Test
Python	412,178	23,107	22,176
Java	454,451	15,328	26,909
PHP	523,712	26,015	28,391
Go	317,832	14,242	14,291
Javascript	123,889	8,253	6,483

Table B.1: The original dataset before cleaning and processing