Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Challenging the Lexical Focus of Code Search

Semester 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

February 27, 2023

Acknowledgements

I would like to thank my two supervisors, Peter and Florian, who provided invaluable feedback while undertaking the thesis. Furthermore, I would like to thank Vandit who provided regular feedback and reports on the functionality of the code. Finally, I would like to thank Professor Wattenhofer for enabling me to write this semester thesis in his group.

Abstract

The principle goal of this work was to challenge the lexical focus of existing code search models by showing that removing lexical clues from snippets will reduce performance. We also aimed to construct a code search/code summarization data engineering structure that would allow the user to specify certain parameters without requiring any further instructions. Its modular design approach means that users can introduce new models, datasets, and modifications without the need for consideration of the whole structure. A secondary goal was to create a larger number of possible attacks, expanding on the work done previously and using these to create a benchmarking database for various models, datasets, and attacks. We were able to show performance drops in the assessed metrics, with the severity of the drops varying depending on the attack, across all languages and models.

Contents

Chapter 1

Introduction

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 principal 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 gain an understanding of 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. A good example of a use case to better understand the goal of code search is the following: We have a corpus of documentation strings and paired code snippets and a neural network architecture that has been trained to conduct code search. A user now poses a search query in natural language to the network, and the network provides the user with a piece of code, a snippet, from the corpus that does exactly, or as closely as possible, what the user has queried.

1.2 Model Robustness

There currently exists a plethora of frameworks that can tackle these challenges with varying degrees of accuracy. As these models' inner workings are black boxes (relying primarily on neural networks), there is some uncertainty as to what the model specifically relies on. The leading assumption is that keywords and phrases that are used in the code snippet and the corresponding description play the most significant role. 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.

1.3 Attacks

As our principal interest was to study the effect of the removal of lexical clues from a code snippet on the performance of the assessed metrics, we had to build datasets that, in some way or form, reduced or removed lexical information. We call these modifications "attacks" as they assault the original datasets to enforce some form of restriction on the code snippets. As the modifications could vary in the implemented restrictiveness, we designed multiple attacks with varying approaches to how the lexical clues are changed, resulting in wide-ranging degrees of severity.

1.4 Pipeline

This thesis will explore the implementation of a data engineering structure, henceforth referred to as "the pipeline". It combines various shell scripts that are used to assemble an automatized approach to conducting code search and code summarization training and testing jobs for various datasets and store the gathered results in a server-based database. This approach accelerates the individual steps of creating an attacked dataset, training a model on this dataset, testing the model, and its modular construction allows for liberal expansions in all adjustable parameters. The pipeline, along with the scripts for the attacks, is located on the TIK Arton cluster at ETH Zurich.

1.5 Inference Type

As part of the pipeline we have implemented two different types of inference that can be selected. The first, called Confused, uses the attacked dataset to both train and then test a model. The second, called Not Confused, uses the baseline dataset to train the model and then the modified dataset to test it.

1.6 Code Summarization

In addition to code search, the pipeline structure was also constructed to enable a user to train models for Code Summarization, which is the task of generating concise natural language descriptions of source code in order to improve program comprehension and maintenance. [1]. We can observe that the two tasks of code search and code summarization are related. In this thesis, we will focus solely on the task of code search. Nonetheless, the pipeline structure has been written to support code summarization jobs as well as their differing metrics.

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. The description of the models has been taken from work done previously in [2].

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.

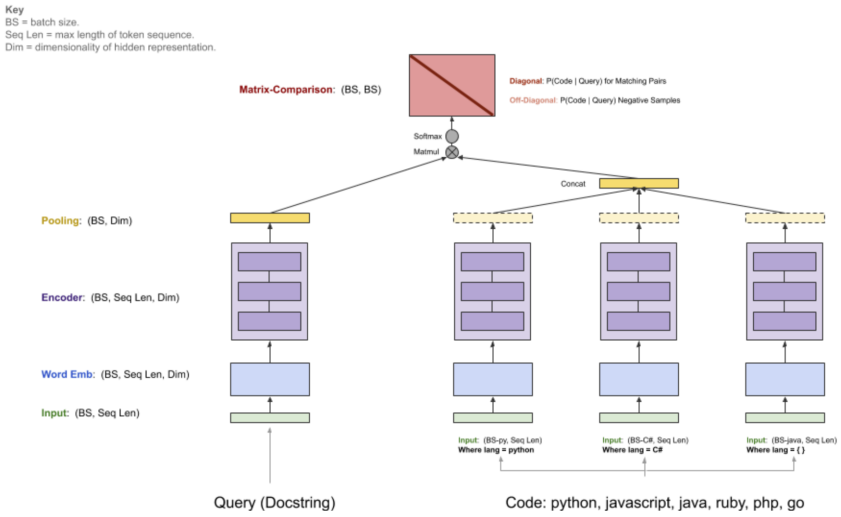

Figure 2.1: This figure presents the model architecture used in the CodeSearchNet. Taken from the CodeSearchNet GitHub repository [3].

Figure 2.1 illustrates the general architectural approach taken by CodeSearchNet. 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 [4, 5]. 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 [5], have shown better results. The exact procedure is explained in detail in [6]: 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 [7], 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 [8].
- Self-Attention: Multi-Head attention is used to compute representations of each token in the sequence, as used in [9]. 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 GraphCodeBERT

GraphCodeBERT considers the inherent structure of code by leveraging its semantic-level information, known as data flow, during pretraining. As defined in [10], 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 [9].

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.

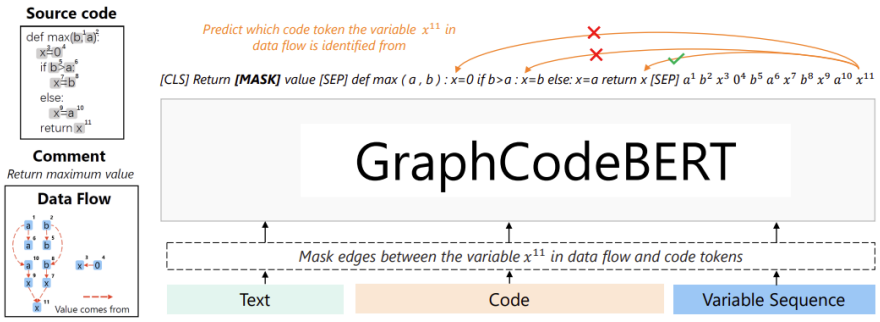

Figure 2.2: This figure illustrates an example of the node alignment. We first mask edges between variable x^{11} in the data flow and code tokens and subsequently predict which code token the variable in the data flow is identified from. The tick indicates that the variable is x^{11} is predicted from the variable x in "return x" based on the information in the data flow. Taken from [10]

. .

Downstream Tasks

GraphCodeBERT has a wide range of downstream applications. In [10], four downstream tasks are explored: code search, clone detection, code translation and code refinement. Relevant to our study is only code search.
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.

2.1.3 SynCoBERT

SynCoBERT, as developed in [11], 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) [12], 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 [13] 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 [14], which provides more meaningful negative

samples that lead to more robust results. The conclusive step is using the Barlow Twins framework [15], 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:

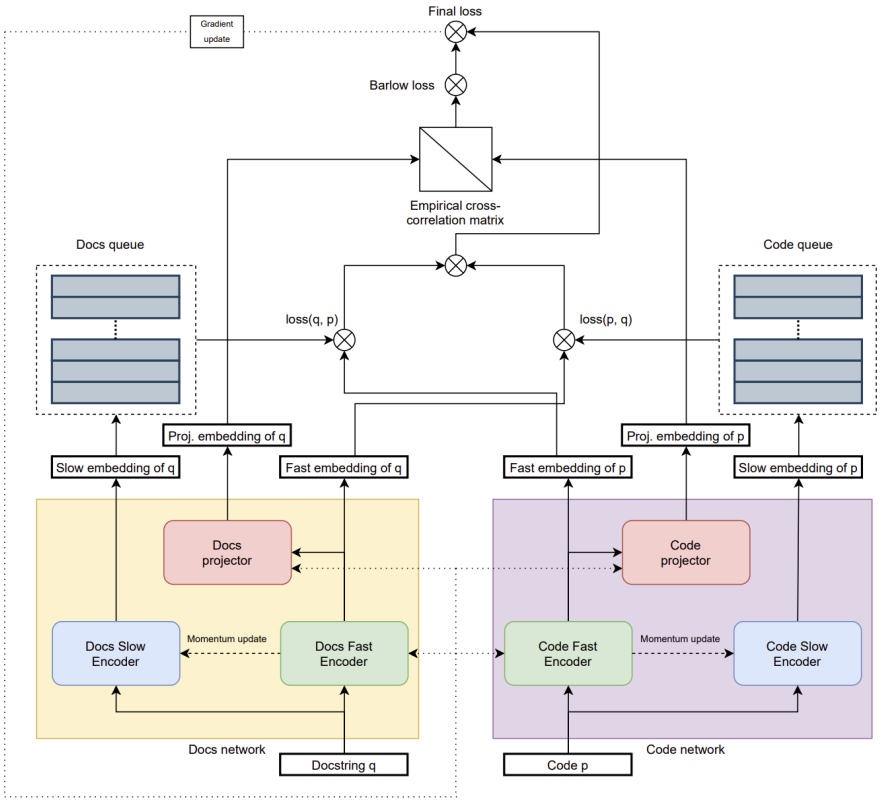

Figure 2.3: The complete framework combining xMoCo, DyHardCode and Barlow Loss. Taken from [11].

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 [6]. 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 [16]:

- 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

Note that the Funcom and TL-CodeSum datasets were not collected as part of this original group and thus must be viewed in isolation.

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
TL-CodeSum	69,708	8,714	8,714
Funcom	1,949,120	99,000	100,000

Table 2.1: The dataset used for the attacks and running the models.

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, validation and testing code. Note that Funcom and TL-CodeSum were added for the sake of completeness. More information about their construction can be found in section 3.2.

Chapter 3

Method and Procedure

3.1 Pipeline

The principle goal of the pipeline was to facilitate the procedural steps that previously had to be conducted independently to modify a particular dataset and run it on a particular model. This includes the creation of a modified dataset, the training of this dataset on a particular model, the testing of this dataset on a particular model, and the recording of the received scores. The idea was to design it in a modular fashion so that it could be both used and extended as desired. Figure 3.1 illustrates the overall layout of the pipeline architecture. Using this figure as a guide, we can describe the individual parts that comprise the data pipeline.

3.1.1 Launching a job

In order to start a job, it is necessary to provide seven distinct parameters. These are, in the order in which they must be passed as arguments:

1. Model: This parameter specifies the model to be used.
2. Dataset: This parameter specifies the base dataset that is to be used for this job.
3. Attack: This parameter specifies the attack that is to be applied to the dataset prior to the training phase.
4. Training: We can pass this parameter in order to specify whether training is meant to take place. This can be useful, for example, when a model has already been trained in a previous job, but we now want to test using a different inference scheme. To skip training, we can simply pass "skip_training".
5. Testing: This parameter specifies whether testing should be carried out or not. To skip testing, we can simply pass "skip_testing".

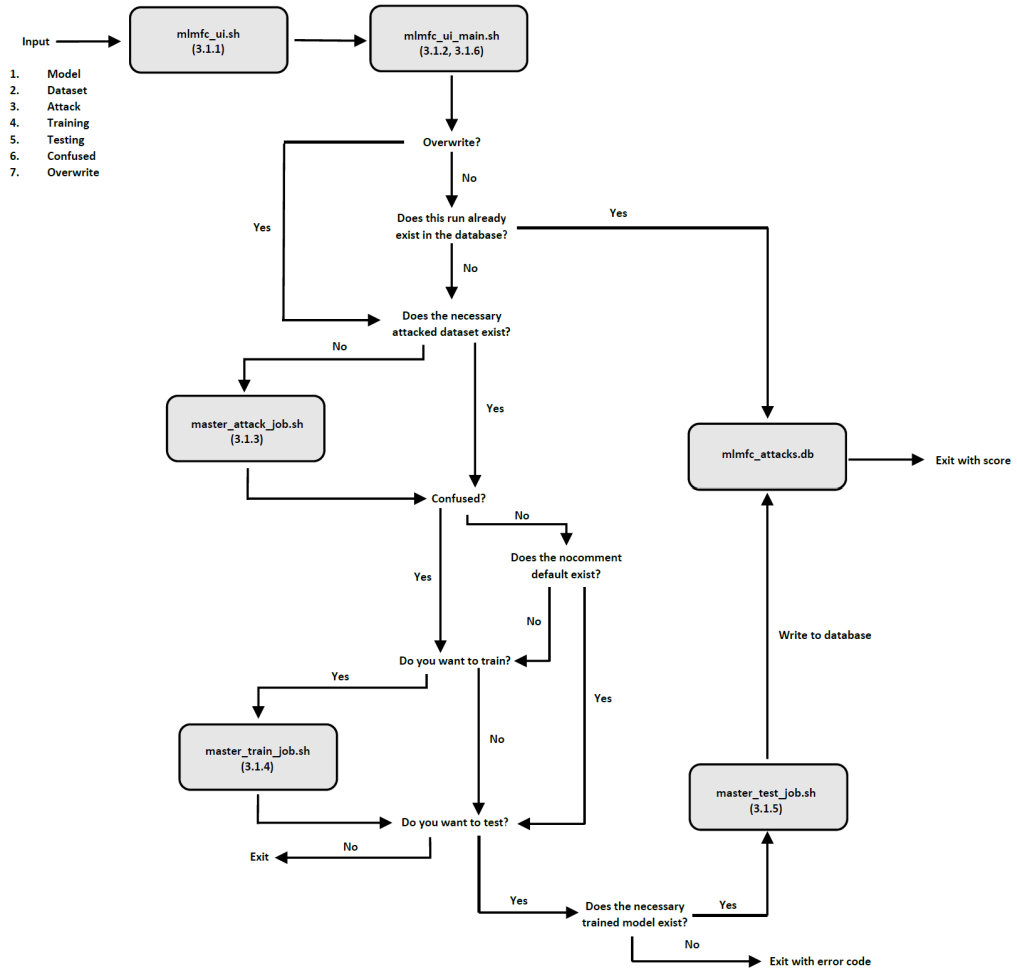

Figure 3.1: An illustration of the entire pipeline architecture. The corresponding sections are labelled in each box.

6. Confused: This parameter specifies the type of inference that is to be done when evaluating. We have defined two different possibilities for evaluation. The first one, called Confused, uses the modified dataset for training as well as for testing. The second, called Not Confused, uses the No Comment dataset for training (No Comment is our default baseline) and, subsequently, the modified dataset for testing.

7. Overview: This parameter forces a run of the set of parameters even if this exact set already exists in the dataset. This can be useful when a faulty modification is discovered, or the model did not complete training as expected but still carried out testing.

With the parameters specified, we can now start to explore the pipeline. The chosen parameters are passed to the `mlmfc_ui.sh` script which acts as a wrapper for the `mlmfc_ui_main.sh` script. In `mlmfc_ui.sh`, we set certain environmental parameters that are used by the SLURM (Simple Linux Utility for Resource Management) job scheduler. Based on the model and dataset passed as parameters, the job will require a different set of environmental specifications. For jobs with more intensive resources requirements due to parameter-heavy models or larger dataset size, we can specify more powerful GPUs, such as the Nvidia Geforce RTX 3090 or the Nvidia Quattro A6000, as well as request more memory on a particular node. Furthermore, we can use parallelism to use multiple GPUs on one node simultaneously. We can also restrict jobs to only run on certain compute capability classes. An overview of available SLURM commands can be found at [17].

3.1.2 Moving to the main script

Having set the environmental variables in place, `mlmfc_ui.sh` calls `mlmfc_ui_main.sh` which envelopes most of 3.1. We first check whether the Overwrite parameter has been set or not. If not, we can check whether a run with this particular set of parameters already exists in the database. On top of the specified parameters, we must also compare the existing git hash. This is the hash that is created whenever a new commit is pushed on the Gitlab site of the project [18]. It provides a chronological order to updates, and, as such, we can check whether any particular run in the database also uses the most recent git hash. If this is the case, and all parameters match, and we have not specified the Overwrite parameter, the job will exit and return the respective scores from the database.

3.1.3 Moving to the attack script

If we cannot find a matching score in the database, the next step is to check for the existence of the attacked dataset. If this is not the case, we call up the `master_attack_job.sh` script, which contains all possible attacks for all available languages. By using appropriate case statements, we can select the correct attack and dataset and use this to generate the desired, modified dataset. These attacks are written either in Python or the respective language of the dataset. Source code for all attacks has been made available on [18].

Once the modified dataset has been generated and stored, we can assess the type of inference that is to be conducted. If we are using evaluation on the Confused dataset, we can proceed to training as no further checks are required. However, if we are using inference on the Not Confused dataset, we need to enquire whether the default No Comment model executable exists. If this is the case, we can skip the training step and proceed to the testing phase.

3.1.4 Moving to the training script

If training has not been skipped (either because the appropriate argument has not been passed or because the No Comment executable does not exist), we enter the `master_train_job.sh` script, which contains training calls to the available models. These are listed in case statements and link to the respective model's source code. The training phase is normally the most time-consuming stage of the pipeline taking anywhere from ten minutes to 35 hours, depending on the size and complexity of the chosen model. Progress for the training stage can be tracked through the log files of the job number.

3.1.5 Moving to the testing script

Once the training phase has been completed, we move on to the testing phase. We first need to check whether the testing argument has been passed. If this is not the case, testing will be skipped, and then we will exit the pipeline without writing anything into the database. There may be instances where a user has chosen to skip the pipeline's training stage but wants to test a model executable, so a test call for a non-existent model may be made. In this case, the pipeline exits with an error message stating that the model executable for the desired parameter combination does not exist. If testing has been selected and the necessary executable exists, we move to the `master_test_job.sh` which, in similar fashion to `master_train_job.sh`, lists testing calls for all available models.

3.1.6 Returning to the main script

Once the testing stage has been completed, we parse the desired scores from the job log files. Each job submitted to the cluster has a unique, running job ID, meaning that the respective scores' extraction is always deterministic. Based on whether we are conducting a code search or a code summarization task, the scores that will be extracted from the log files are divergent. If we are conducting code search, the scores that will be extracted are: MRR, Top-1, Top-5, and Top-10, where MRR is the Mean Reciprocal Rank [19], and the Top scores represent the fractions of snippets that are ranked in the first, first five and first ten positions respectively. If we conduct a code summarization task, the scores that will be extracted from the log file are: BLEU-composite, BLEU1, BLEU2, BLEU3, and BLEU4. BLEU-1 to BLEU-4 refer to the corpus-level modified N-gram precision. The BLEU scores are the count of all (corpus-level) n-grams of the predicted sentences that match with n-grams in their corresponding reference sentence(s) divided by the number of n-grams in the predicted sentences. The BLEU composite is just the geometric mean of these four scores. [20]

Once the desired scores have been extracted from the log files, the last step is to write them into the database using an SQLite query. This has allowed us to

create a benchmarking database, containing the various combinations of models, datasets, and attacks.

In order to be able to draw comparisons between the scores received by the user and the scores stated in the papers where the models were presented, the hyperparameters chosen for training and testing are the default values provided in the implementation.

3.1.7 Benefits of modular pipeline architecture

The primary benefit of this pipeline architecture is that it is entirely modular. This means anyone can add their own model, dataset, or attack without requiring in-depth knowledge of the entire process, as they can simply append a case statement to the end of the necessary file. To facilitate this process, we have also written a user manual that provides an overview of how to add one's own work to the pipeline and a quick-start guide. The manual can be found at: [21]. The modular architecture also facilitates debugging as we can run different sections in isolation from one another, so the process of finding and fixing errors can be accelerated.

3.2 Dataset Construction

In addition to the already established CodeSearchNet datasets, we also wanted to use datasets for which no baseline exists. To this end, we added two further datasets, both based on the Java programming language. These are TL CodeSum [22] and Funcom, [23], which is the largest dataset at hand at roughly 2.1 million snippets and paired descriptions. In order to fit the datasets neatly into the pipeline, modifications were done to add missing columns, change the data format to jsonl and change column names to make them accessible to the attack scripts.

3.3 Attacks

Building on the work done previously [2], the idea was to expand upon the number of possible attacks (modifications that could be taken to affect the lexical structure of the code snippet without affecting its meaning). To this end, a range of attacks was developed. In general, we can classify the modifications that can be applied into three categories: semantic attacks, where we change the meaning and functionality of a snippet, syntactic attacks, where we change the functionality but not the meaning of a snippet and lexical attacks, where neither the meaning nor the functionality is changed but a snippet changes on a symbolic level. "Functionality" can be thought of as the algorithmic implementation of an idea, while "meaning" is the idea itself that is to be implemented in code. In

this thesis, we focus on lexical attacks, specifically those presented below, but research into the other categories has also been done. It is important to note that while lexically attacked snippets may look visually different from the original, their functionality is fundamentally preserved. Furthermore, the substitution of any individual identifier is deterministic and unique, meaning that based on a known scheme, we can say exactly what the modified identifier will look like. Specific algorithmic implementations of every attack can be found at [18] and are provided for every dataset currently usable by the pipeline.

We will use the example code snippet below, taken from the Python dataset, to demonstrate the effect that the individual attacks have:

```
def _check_series_localize_t(s, timezone):
    from pandas.api.types import is_datetime64tz_dtype
    tz = timezone or _get_local_timezone()
    #handle nested time case
    if is_datetime64tz_dtype(s.dtype):
        return s.dt.tz_convert(tz).dt.tz_localize(None)
    else:
        return s
```

3.3.1 No Comment

The No Comment attack provide the baseline dataset against which all other datasets are compared and scored. As some, but not all, of the comments present in the dataset, may be used as tokens for training, it was considered more appropriate to remove all of them throughout all datasets and do this consistently throughout all attacks. Besides the removal of comments, no other modification was undertaken. The idea behind this attack was the belief that the models leverage the lexical information contained in the comments quite heavily, so removing them would result in a significant weakening in the scores.

The effect of the attack on the example code snippet:

```
def _check_series_localize_t(s, timezone):
    from pandas.api.types import is_datetime64tz_dtype
    tz = timezone or _get_local_timezone()
    if is_datetime64tz_dtype(s.dtype):
        return s.dt.tz_convert(tz).dt.tz_localize(None)
    else:
        return s
```

3.3.2 Full Hash

The fullhash is an attack that was taken from previous work [2]. For example, it is clear to a reader that a variable called "counter" will be keeping count of some iterative loop or similar. However, if we now hash this string and replace it in the snippet with var458796e4e963a163322319ba62d683315a930a09, it is no longer clear what this variable is supposed to achieve. While previously [2] we had the option to specifically select what type of identifiers we wanted to modify, here we chose the most aggressive strategy, which means hashing all function names, arguments, and variables using a SHA1 hash. The code can be adapted to test other hashing combinations as well. The idea behind this attack is to provide a benchmark score for the case when all lexical resemblance between the original snippet and the newly modified snippet has been destroyed.

The effect of the attack on the example code snippet (hashes have been shortened for readability):

```
def fun79a06b59(argdd4ba62f, arg5a901fe9):
    from pandas.api.types import is_datetime64tz_dtype
    var1412349a = arg5a901fe9 or _get_local_timezone()
    if is_datetime64tz_dtype(argdd4ba62f.dtype):
        return argdd4ba62f.dt.tz_convert(var1412349a) \
        .dt.tz_localize (None)
    else:
        return argdd4ba62f
```

3.3.3 K-Shift Dataset

This attack results in a cyclical shift of identifiers between code snippets by an amount k. To create this shift, we create a dictionary of key-value pairs where the keys are the identifiers from the current snippet, and the values are the identifiers from the k-th previous snippet. We store the two sets as a list (providing an order to them) and assign the values to the keys in the order in which they are encountered in the abstract syntax tree. This means that the identifiers present in a snippet no longer match with the snippet that they are part of. In our database, we have created two separate instances of the attack: the first one shifts identifiers by three snippets (3-shift-dataset), and the second shifts identifiers by 64 snippets (64-shift-dataset). The idea was to see whether a more significant shift would result in a lower score. This is a valid consideration since the datasets group snippets coming from the same repository in sequence (a consequence of the GitHub scraping method used to compile the datasets). Programmers writing code for a single repository will reuse identifiers in order to maintain an overview of the code structure. This reuse between different sections of code and, by extension, different functions means that consecutive

code snippets in the datasets will likely use similar, if not identical, identifiers. As such, shifting by three snippets would mean that identifiers are more likely to stay within the same repository that they were originally scraped from than when shifting by 64 snippets and thus more meaningfully contribute to the overall understanding of a piece of code.

The effect on the example code snippet relies on another snippet that is k snippet instances away from the example snippet. For demonstration purposes, we have extracted the following identifiers from the other snippet: ['user', 'reach', 'flow', 'difference', 'time_in_zone']. The effect will be as follows:

```
def user(reach, flow):
    from pandas.api.types import is_datetime64tz_dtype
    difference = flow or _get_local_timezone()
    if is_datetime64tz_dtype(reach.dtype):
        return reach.dt.tz_convert(difference).dt.tz_localize(None)
    else:
        return reach
```

Note the fact that we have extracted more identifiers from the other snippet than can be used in the example snippet, so the last one, 'time_in_zone', will not be used.

3.3.4 K-Shift Snippet

This attack cyclically rotates the individual identifiers of a snippet. We can create a dictionary by listing any individual identifier we find inside the abstract syntax tree as a key and then shift them according to the specified instance of "k" to generate the values for the dictionary. We again created two instances of this attack, the first shifting by three identifiers (3-shift-snippet) and the second by 64 identifiers (64-shift-snippet). Since very few code snippets have 64 unique identifiers, shifting by this amount may result in a scenario where an identifier will find itself once again in its original position or where the same effect could be achieved using a much smaller amount of shifting.

The effect on the example code snippet will be demonstrated using the 3-shift instance of the attack. We have four unique identifiers: check_series_localize_t, s, timezone, tz. Using 3-shifting means that the following key-value pairs (where the original identifier is the key and the newly shifted identifier is the value) are created: {_check_series_localize_t: s, s: timezone, timezone: tz, tz: _check_series_localize_t}. Substituting the values into the snippet leads to the following:

```
def s(timezone, tz):
    from pandas.api.types import is_datetime64tz_dtype
```

```
_check_series_localize_t = tz or _get_local_timezone()
if is_datetime64tz_dtype(timezone.dtype):
    return timezone.dt.tz_convert(_check_series_localize_t) \
    .dt.tz_localize(None)
else:
    return timezone
```

3.3.5 Most Popular

In a first iteration over the entire dataset, this attack creates a ranking of the number of occurrences of any individual identifier over the whole dataset as well as an identical ranking within each snippet. We then iterate over the dataset again and now link up each unique identifier in a snippet with its matching partner in the overall ranking of all identifiers based on its rank. The idea of this attack is to drastically reduce the total number of identifiers that are used in the entire dataset. Since most snippets in any dataset have few unique identifiers, as the snippets are not hundreds of lines long, most identifiers that exist only in a couple of snippets will never be matched to any identifier in one snippet. This means that the number of identifiers that the models encounter when training is reduced massively, and the belief is that this will hamper performance as the model has not been "exposed" to as many varied tokens. Looking at the available Python dataset, here are the five most common identifiers in the dataset, along with the number of occurrences of each one:

1. data: 61,956
2. i: 61,614
3. os: 53,136
4. name: 47,070
5. x: 45,864

Using this information, we can create a mapping from the most common identifiers in our example snippet to the most common identifiers in the whole dataset. For our example snippet, we have the following number of occurrences, ordered by frequency: {s: 4, timezone: 3, tz: 3, _check_series_localize_t: 1}. We can now map these to the overall most common identifiers. If there are two snippet identifiers with the same number of occurrences, then the one that is encountered first in the abstract (in our case, this means timezone is mapped before tz since arguments are visited first). Our mapping now looks as follows: {s: data, timezone: i, tz: os, _check_series_localize_t: name}. Substituting this into the snippet yields:

```
def name(data, i):
    from pandas.api.types import is_datetime64tz_dtype
    os = i or _get_local_timezone()
    if is_datetime64tz_dtype(data.dtype):
        return data.dt.tz_convert(os).dt.tz_localize(None)
    else:
        return data
```

3.3.6 Ordered ID

This attack considers all identifiers that are present in an individual snippet and then renames them with a label. So the first identifier that is encountered is labeled id1, the second id2, and so on. The idea is to drastically reduce the number of unique identifiers the model encounters when training similar to the Most Popular attack.

The effect on the example snippet can be seen below. The identifiers are labeled in the order they are encountered in the abstract syntax tree.

```
def id1(id2, id3):
    from pandas.api.types import is_datetime64tz_dtype
    id4 = id3 or _get_local_timezone()
    if is_datetime64tz_dtype(id2.dtype):
        return id2.dt.tz_convert(id4).dt.tz_localize(None)
    else:
        return id2
```

3.3.7 Random Permutation

This attack is derived from the k-shift snippet attack and as such serves as a comparative attack. Rather than shifting all identifiers by a set amount k, each snippet shifts its identifiers by a random number between one and twenty.

For the example snippet, we can use a randomly select shift of one. This means that the following key-value pairs are created: {_check_series_localize _t: tz, s: _check_series_localize_t timezone: s, tz: timezone}.

```
def tz(_check_series_localize_t, s):
    from pandas.api.types import is_datetime64tz_dtype
    timezone = s or _get_local_timezone()
    if is_datetime64tz_dtype(_check_series_localize_t \.dtype):
        return _check_series_localize_t.dt.tz_convert(timezone) \
        .dt.tz_localize(None)
```

```
    else:
        return _check_series_localize_t
```

3.3.8 Translation

The translation attack translates all comments in the original dataset into a language of choice. To do this, we used the Google Cloud Translation API to translate the extracted comments using Google's own Neural Translation software. While in principle, the comments can be translated into any available language, we chose to only create one instance and translate the comments into Spanish. The idea behind this attack is that the language model may leverage language structures present in a foreign language but not in English, which could provide an edge when aiming to find appropriate snippets in code search.

Using the Spanish instance of the attack, we can demonstrate the effect on our example snippet:

```
def _check_series_localize_t(s, timezone):
    from pandas.api.types import is_datetime64tz_dtype
    #manejar el caso de tiempo anidado
    tz = timezone or _get_local_timezone()
    if is_datetime64tz_dtype(s.dtype):
        return s.dt.tz_convert(tz).dt.tz_localize(None)
    else:
        return s
```

3.4 Database Utilization

All scores, both for code search and code summarization are stored locally in a .db file to facilitate benchmarking comparisons. To write and read this file, we use SQLite queries. The write queries are integrated as part of the pipeline described previously, and we have also provided a script called `sqliteinteract.py` that can be used to craft queries tailored to your specific wishes. Furthermore, we can also write queries explicitly tailored to specific requirements (for example: fetch all scores with an MRR over 0.5), which was previously impossible. Alternatively, we can also use the pipeline structure to read out single values for certain combinations of parameters. We were also briefly considering using an external Oracle Server in combination with MySQL; however, this was quickly discarded due to fact that local, on-server storage cost less money.

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 different high-end graphics cards ranging from Tesla K40C's on the lower end of the performance spectrum to Nvidia Quattro RTX A6000 on the high. While interactive sessions on each graphics card can be conducted, the recommended way of running an experiment is through shell scripts. As such, the pipeline was constructed around the usage of several linking scripts that can be used in combination to achieve the desired results.

4.2 Environments

All models are written in Python and primarily based on the PyTorch or Tensorflow Machine Learning frameworks. In addition, all models are run in their own Conda Environments, 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). Furthermore, we have also set up environments that work with all available attacks. In contrast to [2], we now run all attacks directly on the server. Previously, the modified datasets were generated locally and then uploaded to the server, but this has now been automatized so that only a baseline version of the dataset is required to be present on the server, and everything else is created using shell scripts.

Chapter 5
Results

In this section we will highlight the most significant results that have been gathered. An overview of all collected scores can be found at [24].

5.1 Default versus No Comment

In the first step, we can compare our baseline set, which is the No Comment dataset, with the scores stated in the CodeSearchNet challenge paper [6], the GraphCodeBERT paper [10] and the SynCoBERT paper [11]. As mentioned in section 2.2, the CodeSearchNet paper uses the original default dataset (which the researchers) collected themselves), GraphCodeBERT and SynCoBERT use a slightly modified version of this original.

The most crucial modification that was applied was the comment removal. Since some, but importantly not all, comments, will be used in the tokenization. As we cannot reliably say which comments will be used, the removal of all comments provided a fair method of evaluation. However, one will notice nevertheless that our stated No Comment baseline scores do not quite line up with No Comment scores reported by GraphCodeBERT and SynCoBERT. The reason for this is that we used a different tokenization to create our datasets, which could be addressed in future experiments (covered in more detail in section 6.2). Furthermore, we also collected scores in the form of Top-1, Top-5 and Top-10, for which no baseline comparison score exists. In addition, Funcom and TL-CodeSum were two datasets that were added on top of the existing languages. As such, no baseline score exists for them. Funcom posed additional challenges as its size was multiple times larger than the next largest dataset. This meant we were unable to run it on the more complex models due to the time constraints imposed by the cluster.

This table illustrates that even just the comment removal will lead to a, in some cases, significant performance drop across all metrics. This suggests that a lot of lexical clues that the model leverages are contained solely in the comments of a code snippet.

Model	Python		Java		Javascript		PHP		Go		TL-CodeSum		Funcom	
	Base	NoCom	Base	NoCom	Base	NoCom	Base	NoCom	Base	NoCom	Base	NoCom	Base	NoCom
NBoW	0.634	0.537	0.660	0.011	0.399	0.010	0.684	0.679	0.802	0.062	N/A	0.042	N/A	0.009
SelfAtt	0.634	0.454	0.643	0.273	0.445	0.143	0.693	0.693	0.871	0.804	N/A	0.367	N/A	**0.254**
ConvSelfAtt	0.622	0.455	0.326	0.326	0.342	0.193	0.697	0.669	0.873	0.804	N/A	0.421	N/A	0.241
1D-CNN	0.505	0.310	0.537	0.253	0.158	0.059	0.609	0.615	0.801	0.757	N/A	0.327	N/A	0.206
GraphCodeBERT	0.694	**0.763**	0.690	0.637	0.643	**0.564**	0.647	**0.756**	0.895	0.850	N/A	**0.644**	N/A	N/A
SynCoBERT	**0.718**	0.711	**0.761**	**0.673**	**0.681**	0.480	**0.701**	0.701	**0.929**	**0.888**	N/A	0.418	N/A	N/A

Table 5.1: The table above presents the MRR scores of all models across all examined languages for two versions of the datsets. The first column in each language represents the baseline score, while the second column represents the No Comment version that was used as the baseline for our experiments.

5.2 Confused versus Not Confused

The tables below illustrate the overall trend of Confused inference versus Not Confused inference using the random permutation attack. Tables 5.2 and 5.2 illustrate the overall trend that the Not Confused inference cannot keep up with the Confused Inference scores, performing worse across all models and all datasets.

Model	Python				Java				Javascript				PHP			
	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
NBoW	0.643	0.545	0.762	0.822	0.011	0.002	0.009	0.015	0.012	0.002	0.012	0.020	0.713	0.620	0.830	0.878
SelfAtt	0.516	0.417	0.626	0.696	0.189	0.114	0.258	0.332	0.100	0.054	0.138	0.186	0.616	0.517	0.733	0.801
ConvSelfAtt	0.497	0.404	0.601	0.669	0.294	0.207	0.387	0.461	0.139	0.080	0.184	0.244	0.669	0.569	0.788	0.847
1D-CNN	0.339	0.303	0.496	0,581	0.193	0.118	0.259	0.333	0.036	0.010	0.044	0.077	0.621	0.521	0.737	0.805
GraphCodeBERT	**0.725**	**0.623**	**0.853**	**0.900**	0.615	0.507	0.749	0.812	**0.526**	**0.414**	**0.659**	**0.735**	**0.736**	**0.635**	**0.863**	**0.909**
SynCoBERT	0.683	0.585	0.801	0.857	**0.660**	**0.563**	**0.778**	**0.832**	0.447	0.345	0.566	0.641	0.683	0.585	0.801	0.859

Model	Go				TL-CodeSum				Funcom			
	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
NBoW	0.140	0.093	0.179	0.224	0.036	0.014	0.049	0.069	0.009	0.001	0.006	0.013
SelfAtt	0.649	0.561	0.754	0.809	0.378	0.301	0.460	0.515	0.230	**0.138**	0.315	0.426
ConvSelfAtt	0.734	0.672	0.807	0.839	0.435	0.346	0.533	0.594	**0.231**	0.137	**0.320**	**0.434**
1D-CNN	0.714	0.647	0.793	0.828	0.291	0.208	0.377	0.440	0.178	0.101	0.245	0.331
GraphCodeBERT	0.839	0.775	0.919	0.943	**0.623**	**0.516**	**0.757**	**0.827**	N/A	N/A	N/A	N/A
SynCoBERT	**0.884**	**0.839**	**0.938**	**0.955**	0.517	0.408	0.655	0.735	N/A	N/A	N/A	N/A

Table 5.2: Results of all datasets and models modified with the Random Permutation attack and Confused Inference.

Model	Python				Java				Javascript				PHP			
	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
NBoW	0.007	0.001	0.005	0.011	0.013	0.003	0.011	0.021	0.011	0.003	0.010	0.017	**0.696**	**0.600**	**0.814**	**0.869**
SelfAtt	0.491	0.383	0.617	0.701	0.149	0.088	0.199	0.261	0.093	0.052	0.121	0.168	0.543	0.442	0.662	0.735
ConvSelfAtt	0.399	0.294	0.513	0.602	0.175	0.110	0.229	0.298	0.123	0.071	0.165	0.225	0.485	0.384	0.699	0.676
1D-CNN	0.273	0.187	0.353	0.441	0.100	0.057	0.129	0.179	0.030	0.012	0.040	0.055	0.398	0.301	0.499	0.588
GraphCodeBERT	**0.522**	**0.413**	**0.651**	**0.732**	0.456	0.347	0.581	0.664	**0.462**	**0.352**	**0.589**	**0.676**	0.572	0.461	0.702	0.780
SynCoBERT	0.472	0.372	0.586	0.663	**0.482**	**0.380**	**0.599**	**0.673**	0.370	0.276	0.467	0.555	0.518	0.416	0.632	0.713

Model	Go				TL-CodeSum				Funcom			
	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
NBoW	0.115	0.070	0.151	0.194	0.028	0.008	0.037	0.058	0.009	0.001	0.006	0.014
SelfAtt	0.270	0.222	0.312	0.349	0.179	0.111	0.239	0.305	**0.197**	**0.116**	**0.270**	**0.365**
ConvSelfAtt	0.250	0.212	0.281	0.313	0.224	0.144	0.301	0.382	0.169	0.095	0.227	0.318
1D-CNN	0.210	0.179	0.236	0.257	0.081	0.040	0.111	0.157	0.126	0.064	0.173	0.251
GraphCodeBERT	0.377	0.298	0.457	**0.537**	**0.464**	**0.359**	**0.584**	**0.669**	N/A	N/A	N/A	N/A
SynCoBERT	**0.398**	**0.328**	**0.468**	0.535	0.383	0.281	0.501	0.595	N/A	N/A	N/A	N/A

Table 5.3: Results of all datasets and models modified with the Random Permutation attack and Not Confused Inference.

5.3 Best and Worst Performance by dataset

One set of interesting results can be derived from looking at the data from absolute and relative (to the baseline No Comment dataset) standpoints and looking for the best and worst overall scores for each dataset. In this instance, we will just consider the Confused Inference. We have discovered some negative relative scores. These correlate to an improvement in performance. All other attacks' scores fall between the two extremes shown below. Thus any conclusions that can be drawn for the best and worst scores will also hold for the scores in between.

Considering the worst scores first:

Most Popular	**MRR**	**Top-1**	**Top-5**	**Top-10**
NBoW	0.007	0.001	0.005	0.010
SelfAtt	0.007	0.001	0.004	0.009
ConvSelfAtt	0.008	0.001	0.006	0.012
1D-CNN	0.007	0.001	0.005	0.009
GraphCodeBERT	0.001	0.000	0.000	0.001
SynCoBERT	0.000	0.000	0.000	0.000

Table 5.4: The worst absolute scores for the Most Popular attack for the Python dataset.

Most Popular	**MRR [%]**	**Top-1 [%]**	**Top-5 [%]**	**Top-10 [%]**
NBoW	98.70	99.77	99.24	98.62
SelfAtt	98.46	99.71	99.30	98.63
ConvSelfAtt	98.24	99.70	98.95	98.23
1D-CNN	97.74	99.52	98.80	98.26
GraphCodeBERT	99.86	100	100	99.89
SynCoBERT	100	100	100	100

Table 5.5: The worst relative scores for the Most Popular attack for the Python dataset expressed as a percentage change relative to the baseline No Comment scores.

Fullhash	MRR	Top-1	Top-5	Top-10
NBoW	0.010	0.002	0.010	0.016
SelfAtt	0.049	0.024	0.063	0.092
ConvSelfAtt	0.082	0.045	0.106	0.149
1D-CNN	0.036	0.015	0.045	0.065
GraphCodeBERT	0.291	0.198	0.386	0.474
SynCoBERT	0.225	0.143	0.305	0.384

Table 5.6: The worst absolute scores for the Fullhash attack for the Javascript dataset.

Fullhash	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	0.00	-100.00	-11.11	0.00
SelfAtt	65.73	73.33	66.31	61.34
ConvSelfAtt	57.51	64.00	58.27	52.85
1D-CNN	38.98	44.44	43.04	39.25
GraphCodeBERT	48.40	56.48	44.30	38.28
SynCoBERT	53.13	62.37	48.91	42.86

Table 5.7: The worst relative scores for the Fullhash attack for the Javascript dataset expressed as a percentage change relative to the baseline No Comment scores.

Fullhash	MRR	Top-1	Top-5	Top-10
NBoW	0.525	0.419	0.649	0.720
SelfAtt	0.417	0.311	0.536	0.621
ConvSelfAtt	0.421	0.298	0.562	0.664
1D-CNN	0.284	0.186	0.383	0.480
GraphCodeBERT	0.486	0.356	0.597	0.682
SynCoBERT	0.461	0.355	0.578	0.662

Table 5.8: The worst absolute scores for the Fullhash attack for the PHP dataset.

Fullhash	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	22.68	27.38	19.28	16.67
SelfAtt	39.83	48.34	33.00	27.37
ConvSelfAtt	37.07	47.99	28.32	20.95
1D-CNN	53.82	63.95	47.46	39.92
GraphCodeBERT	38.10	46.06	31.85	25.95
SynCoBERT	34.24	41.23	29.51	24.08

Table 5.9: The worst relative scores for the Fullhash attack for the PHP dataset expressed as a percentage change relative to the baseline No Comment scores.

We can now look at the best performing attacks, those that have created the smallest performance drop from the baseline No Comment, for a selection of datasets.

Random Permutation	**MRR**	**Top-1**	**Top-5**	**Top-10**
NBoW	0.643	0.545	0.762	0.822
SelfAtt	0.516	0.417	0.626	0.696
ConvSelfAtt	0.497	0.404	0.601	0.669
1D-CNN	0.339	0.303	0.496	0.581
GraphCodeBERT	0.725	0.623	0.853	0.900
SynCoBERT	0.683	0.585	0.801	0.857

Table 5.10: The best absolute scores for the Random Permutation attack for the Python dataset.

Random Permutation	**MRR [%]**	**Top-1 [%]**	**Top-5 [%]**	**Top-10 [%]**
NBoW	-19.74	-25.87	-15.81	-12.76
SelfAtt	-13.66	-20.52	-9.25	-5.78
ConvSelfAtt	-9.23	-19.53	-2.04	1.33
1D-CNN	-9.35	-45.67	-19.52	-12.60
GraphCodeBERT	4.98	7.01	3.07	2.28
SynCoBERT	3.94	4.88	3.26	2.39

Table 5.11: The best relative scores for the Random Permutation attack for the Python dataset expressed as a percentage change relative to the baseline No Comment scores.

3-Shift-Snippet	MRR	Top-1	Top-5	Top-10
NBoW	0.009	0.001	0.007	0.014
SelfAtt	0.094	0.047	0.132	0.179
ConvSelfAtt	0.136	0.077	0.187	0.248
1D-CNN	0.056	0.023	0.075	0.108
GraphCodeBERT	0.538	0.432	0.665	0.741
SynCoBERT	0.450	0.348	0.567	0.643

Table 5.12: The best absolute scores for the 3-Shift-Snippet attack for the Javascript dataset.

3-Shift-Snippet	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	10.00	0.00	22.22	12.50
SelfAtt	34.27	47.78	29.41	24.79
ConvSelfAtt	29.53	38.40	26.38	21.52
1D-CNN	5.08	14.81	5.06	-0.93
GraphCodeBERT	4.61	5.05	4.04	3.52
SynCoBERT	6.25	8.42	5.03	4.32

Table 5.13: The best relative scores for the 3-Shift-Snippet attack for the Javascript dataset expressed as a percentage change relative to the baseline No Comment scores.

64-Shift-Snippet	MRR	Top-1	Top-5	Top-10
NBoW	0.713	0.622	0.825	0.874
SelfAtt	0.630	0.533	0.749	0.810
ConvSelfAtt	0.682	0.581	0.805	0.860
1D-CNN	0.627	0.531	0.740	0.805
GraphCodeBERT	0.737	0.636	0.861	0.908
SynCoBERT	0.685	0.586	0.805	0.860

Table 5.14: The best absolute scores for the 64-Shift-Snippet attack for the PHP dataset.

64-Shift-Snippet	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	-5.01	-7.80	-2.61	-1.16
SelfAtt	9.09	11.46	6.38	5.26
ConvSelfAtt	-1.94	-1.40	-2.68	-2.38
1D-CNN	-1.95	-2.91	-1.51	-0.75
GraphCodeBERT	2.51	3.64	1.71	1.41
SynCoBERT	2.28	2.98	1.83	1.38

Table 5.15: The best relative scores for the 64-Shift-Snippet attack for the PHP dataset expressed as a percentage change relative to the baseline No Comment scores.

The tables above illustrate that the Shift Snippet attacks (both in the 64 and 3 instance variation) and the derived Random Permutation attack perform best, while Fullhash and Most Popular perform worst across all metrics. The tables provided above show the general trend in performance decrease that can be observed across all datasets. The explanation lies in the severity of an individual attack and is explored in more detail in section 5.5. Regardless of the attack, we can say that the performance (with few exceptions) decreases across all observed metrics. We must also mention the instances where a performance increase has been observed. We believe this is not due to the model itself but to the tokenization used to tokenize the individual snippets in the attacked dataset (both No Comment and otherwise). This in itself is a significant drawback of the tested models (observed in all models except GraphCodeBERT) and should be addressed in the design of future models.

5.4 Weighted scores per attack per model

We can utilize the received data to derive observations. In table 5.16, we have calculated a weighted score for every model and attack combination across all four metrics for the confused inference. This means that for every attack and model, we have taken each dataset's scores and calculated an average for each metric, accounting for the size of the individual dataset. Since not all code snippets can be anonymized (due to faulty code, which prevents the construction of an abstract syntax tree), the dataset sizes are not equal to those of the default dataset. An overview can be found in Appendix table E.1. In our eyes, this provides an appropriate way to estimate the performance of a particular model on a particular attack for a dataset in a generic programming language.

5.5 Severity of attack against Performance

Using the weighted scores presented in table 5.16, we can use the boldly stated scores, which highlight the best-performing model for each metric and each attack, to illustrate the performance of an attack against the severity of an attack. By severity, we mean the degree to which the lexical identifiers in an attack have been obfuscated. While there is some degree of subjectivity to this, we can generally sort the attacks by three guiding questions, asked by increasing severity:

1. Do we move around identifiers between individual snippets?
2. If we are moving identifiers between snippets, how significant is the movement?
3. Do we repeat individual identifiers across the dataset?

Using these questions, we can rank the severity of attacks and the associated scores of the four metrics presented in figure 5.1.

The figure suggests that there is a correlation between the severity of the attack and the observed performance loss across all metrics. The more severe the attack, the higher the performance drop across the assessed metrics.

Model	No Comment				Fullhash				3-shift-dataset				64-shift-dataset				3-shift-snippet			
	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
NBoW	0.116	0.090	0.137	0.157	0.092	0.065	0.110	0.132	0.062	0.040	0.077	0.096	0.034	0.019	0.039	0.054	0.135	0.110	0.156	0.172
SelfAtt	0.343	0.247	0.427	0.515	0.240	0.155	0.321	0.412	0.175	0.103	0.237	0.320	0.173	0.101	0.228	0.318	0.312	0.226	0.398	0.484
ConvSelfAtt	0.338	0.248	0.429	0.518	0.251	0.163	0.336	0.429	0.212	0.129	0.288	0.384	0.202	0.122	0.273	0.369	0.347	0.255	0.441	0.528
1D-CNN	0.285	0.199	0.371	0.460	0.170	0.099	0.229	0.313	0.129	0.065	0.177	0.258	0.119	0.059	0.162	0.239	0.274	0.191	0.357	0.444
GraphCodeBERT	**0.734**	**0.641**	**0.849**	**0.895**	0.479	0.371	0.605	0.685	**0.520**	**0.404**	**0.657**	**0.740**	**0.472**	**0.369**	**0.595**	**0.667**	0.654	0.559	**0.769**	**0.820**
SynCoBERT	0.418	0.330	0.552	0.636	**0.500**	**0.399**	**0.618**	**0.692**	0.491	0.387	0.611	0.686	0.460	0.363	0.575	0.646	**0.660**	**0.576**	0.763	0.812

Model	64-shift-snippet				Most Popular				Ordered ID				Random Permutation			
	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10	MRR	Top-1	Top-5	Top-10
NBoW	0.134	0.108	0.155	0.171	0.049	0.033	0.057	0.072	0.083	0.058	0.100	0.121	0.133	0.107	0.154	0.171
SelfAtt	0.320	0.231	0.411	0.498	0.174	0.107	0.231	0.307	0.201	0.125	0.268	0.352	0.311	0.220	0.400	0.496
ConvSelfAtt	0.346	0.252	0.442	0.533	0.197	0.123	0.265	0.346	0.222	0.140	0.295	0.382	0.328	0.236	0.419	0.515
1D-CNN	0.277	0.195	0.357	0.443	0.132	0.072	0.180	0.253	0.145	0.078	0.198	0.278	0.262	0.188	0.341	0.420
GraphCodeBERT	**0.680**	**0.586**	**0.796**	**0.844**	**0.374**	**0.290**	**0.473**	**0.535**	**0.510**	**0.397**	**0.643**	**0.724**	**0.709**	**0.612**	**0.830**	**0.879**
SynCoBERT	0.644	0.557	0.751	0.805	0.362	0.288	0.450	0.506	0.478	0.378	0.596	0.672	0.688	0.598	0.796	0.848

Table 5.16: Weighted scores for confused inference per attack per model.

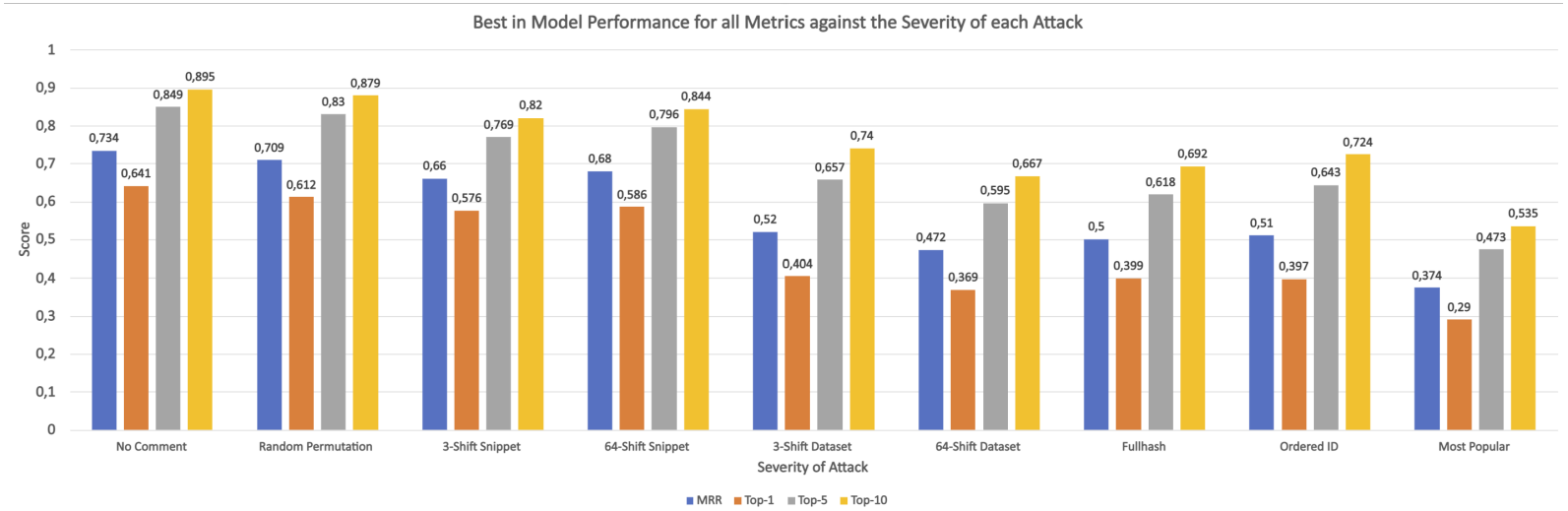

Figure 5.1: Metrics' performance against the different attacks sorted by severity. Based on our subjective opinion, the severity of the attacks is increasing from left to right.

Chapter 6

Conclusion and Outlook

6.1 Conclusion

The primary conclusion that can be drawn from the data is that removing or obfuscating lexical clues will reduce the performance across all models regardless of the severity of the obfuscation. This illustrates that the model does not understand the meaning of a code snippet but instead solely relies on lexical information to create connections between the natural language query and the code snippet.

Our conclusion is further supported by the fact that just the removal of comments will lead to a significant drop in performance, as observed in table 5.1. Since all other attacks also have their respective comments removed, we can attribute the majority of the performance loss across all metrics to the comment removal in itself. This highlights the models' heavy reliance on natural language clues rather than the information found in the code's structure, as changing lexical identifiers does not have as large of an effect as simply removing the comments from the snippet.

We can also see that the more complex models, GraphCodeBERT and SynCoBERT, which use pretrained architectures with more parameters, perform better than those mentioned in the original CodeSearchNet paper [6]. Regardless of the chosen attack, the performance drop seen for GraphCodeBERT and SynCoBERT is nearly always lower than for the other group of models.

We would normally presume that providing a model with more data points should improve its robustness, leading to more stable and improved results. Having added the Funcom dataset, which is nearly ten times the size of the next largest dataset, our assumption was initially that this would lead to a substantial improvement in the scores fetched for Java-based datasets. However, this hypothesis could not be confirmed as the dataset performed on par and often even below the much smaller CodeSearchNet Java and TL-CodeSum datasets. This indicates that the performance of a dataset on the varying models is not necessarily linked to its size but rather to the nature of the programming language

that the dataset is based on. Considering performance metrics by language, we can split the programming languages into three groups:

1. Languages that are (nearly) averse to any modification
2. Languages that are not impacted by mild modifications but are affected by severe modifications
3. Languages that are impacted even by small modifications.

Into the first group belong PHP and Go, into the second Python and Java (with its three datasets), and into the third group just Javascript.

We can also state that more severe lexical attacks, as defined in section 5.5, will lead to larger drops in performance across all metrics. This means that we can not only state that there is a performance drop but that the magnitude of this drop is also directly related to the restrictiveness that the attack imposes on the dataset.

The confused inference method scores higher across all metrics for all models and all attacks. The conclusion that can be deduced from this is that the models perform better when the training data is more similar to the test data from a lexical point of view, even if this means that the lexical information contained in a single snippet is reduced.

6.2 Outlook

As discussed, a central tenet of the design philosophy behind the pipeline is its modular flexibility. Any aspect can be appended to, so naturally, it is sensible to add more attacks, models, and programming language datasets. Regarding the attacks, we have also designed translation attacks that were not pursued due to financial considerations. Finding a cost-efficient alternative for this in the future may be beneficial. Furthermore, another interesting angle could be to utilize other hashes than SHA1, which, for example, are shorter and could be tokenized into fewer symbols. We have also been considering adding the CodeBERT model to the pipeline [25]; however, as of the time of writing, we have yet to be successful in our attempts to integrate this model into the pipeline. Other models, including those that are tasked with conducting code summarization, such as CodeXGlue [26], have also been considered to be added in the future. Adding new programming languages will rely on generating suitable datasets as well as writing or adapting the existing attacks to work for this new programming language. Furthermore, based on the existing datasets and the performance metrics that have been observed grouped by programming language, it would also be an interesting research question to investigate what exactly makes some languages more robust to modifications than others.

One feature in particular that has been discussed is the possibility of compounding multiple different attacks on top of one another. This is currently not supported by the pipeline and would require a major rewrite of a large section of the code base. On top of the technical difficulties, it would also enormously increase the number of possible combinations of attacks that have to be investigated. Even just compounding two attacks would, with the current set of available attacks, lead to more than 400 possible datasets that must be tested. Furthermore, the additional benefit to be gained from doing this is marginal, as even just our set of current attacks leads to (significant) performance decreases.

As seen from the results and stated in the results, a different tokenization using the Python tokenization library rather than the Natural Language Toolkit library (NLTK) may result in different results for the simpler CodeSearchNet family of models. Investigating these discrepancies between different tokenizers could in itself be an interesting research topic.

Another consideration that should be made in the future is to challenge the notion that MRR should be considered as the central metric. The CodeSearchNet authors do not explain why MRR should be used as a measure of performance for this ranking task. We have added the Top-1, Top-5, and Top-10 metrics to gain additional performance insights. Interestingly it appears that MRR most often scores somewhere between the score of the Top-1 and the Top-5 metric. Nevertheless, we should question why MRR is the go-to metric for code search.

Bibliography

[1] E. Shi, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "CAST: enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees," *CoRR*, vol. abs/2108.12987, 2021. [Online]. Available: https://arxiv.org/abs/2108.12987

[2] F. Markus, "Algorithm learning from data," 2022. [Online]. Available: https://pub.tik.ee.ethz.ch/students/2022-FS/BA-2022-36.pdf

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

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

[5] 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

[6] 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

[7] 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

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

[9] 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

[10] 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

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

[12] 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

[13] 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

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

[15] 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

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

[17] "Slurm overview Link." [Online]. Available: https://computing.ee.ethz.ch/Services/SLURM

[18] "Gitlab repository Link." [Online]. Available: https://gitlab.ethz.ch/disco-students/mlmfc

[19] "Mrr definition Link." [Online]. Available: https://en.wikipedia.org/wiki/Mean_reciprocal_rank

[20] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[21] "User manual for pipeline Link." [Online]. Available: https://www.overleaf.com/read/jshzxwrfpfsf

[22] "Tl codesum dataset Link." [Online]. Available: https://github.com/xing-hu/TL-CodeSum

[23] "Funcom dataset Link." [Online]. Available: https://s3.us-east-2.amazonaws.com/icse2018/index.html

[24] "Mlmfc full score set, Link." [Online]. Available: https://docs.google.com/spreadsheets/d/1u0HSNgnu87EG3GNypja5ypwW8pCFqGgDLIqBgtSiabI/edit?usp=sharing

[25] 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

[26] "Codexglue code summarization model Link." [Online]. Available: https://github.com/microsoft/CodeXGLUE/tree/main/Code-Text/code-to-text/code

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

Appendix A

Complete list of tables for worst performances

The results section only covered some of the worst performances and outliers observed, so below we have assembled a list of all worst scores for all available datasets.

Most Popular	**MRR**	**Top-1**	**Top-5**	**Top-10**
NBoW	0.007	0.001	0.005	0.010
SelfAtt	0.007	0.001	0.004	0.009
ConvSelfAtt	0.008	0.001	0.006	0.012
1D-CNN	0.007	0.001	0.005	0.009
GraphCodeBERT	0.001	0.000	0.000	0.001
SynCoBERT	0.000	0.000	0.000	0.000

Table A.1: The worst absolute scores for the Most Popular attack for the Python dataset.

Most Popular	**MRR [%]**	**Top-1 [%]**	**Top-5 [%]**	**Top-10 [%]**
NBoW	98.70	99.77	99.24	98.62
SelfAtt	98.46	99.71	99.30	98.63
ConvSelfAtt	98.24	99.70	98.95	98.23
1D-CNN	97.74	99.52	98.80	98.26
GraphCodeBERT	99.86	100	100	99.89
SynCoBERT	100	100	100	100

Table A.2: The worst relative scores for the Most Popular attack for the Python dataset expressed as a percentage change relative to the baseline No Comment scores.

Most Popular	MRR	Top-1	Top-5	Top-10
NBoW	0.009	0.001	0.006	0.013
SelfAtt	0.092	0.045	0.119	0.173
ConvSelfAtt	0.172	0.099	0.237	0.312
1D-CNN	0.078	0.036	0.101	0.157
GraphCodeBERT	0.455	0.344	0.584	0.668
SynCoBERT	0.465	0.361	0.585	0.665

Table A.3: The worst absolute scores for the Most Popular attack for the Java dataset.

Most Popular	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	18.18	50.00	45.45	35.00
SelfAtt	66.30	75.81	67.40	60.77
ConvSelfAtt	47.24	58.23	43.57	37.60
1D-CNN	69.17	78.70	70.03	61.99
GraphCodeBERT	28.57	35.10	24.25	19.42
SynCoBERT	30.91	37.44	25.95	20.83

Table A.4: The worst relative scores for the Most Popular attack for the Java dataset expressed as a percentage change relative to the baseline No Comment scores.

Fullhash	MRR	Top-1	Top-5	Top-10
NBoW	0.010	0.002	0.010	0.016
SelfAtt	0.049	0.024	0.063	0.092
ConvSelfAtt	0.082	0.045	0.106	0.149
1D-CNN	0.036	0.015	0.045	0.065
GraphCodeBERT	0.291	0.198	0.386	0.474
SynCoBERT	0.225	0.143	0.305	0.384

Table A.5: The worst absolute scores for the Fullhash attack for the Javascript dataset.

Fullhash	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	0.00	-100.00	-11.11	0.00
SelfAtt	65.73	73.33	66.31	61.34
ConvSelfAtt	57.51	64.00	58.27	52.85
1D-CNN	38.98	44.44	43.04	39.25
GraphCodeBERT	48.40	56.48	44.30	38.28
SynCoBERT	53.13	62.37	48.91	42.86

Table A.6: The worst relative scores for the Fullhash attack for the Javascript dataset expressed as a percentage change relative to the baseline No Comment scores.

Fullhash	MRR	Top-1	Top-5	Top-10
NBoW	0.525	0.419	0.649	0.720
SelfAtt	0.417	0.311	0.536	0.621
ConvSelfAtt	0.421	0.298	0.562	0.664
1D-CNN	0.284	0.186	0.383	0.480
GraphCodeBERT	0.486	0.356	0.597	0.682
SynCoBERT	0.461	0.355	0.578	0.662

Table A.7: The worst absolute scores for the Fullhash attack for the PHP dataset.

Fullhash	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	22.68	27.38	19.28	16.67
SelfAtt	39.83	48.34	33.00	27.37
ConvSelfAtt	37.07	47.99	28.32	20.95
1D-CNN	53.82	63.95	47.46	39.92
GraphCodeBERT	38.10	46.06	31.85	25.95
SynCoBERT	34.24	41.23	29.51	24.08

Table A.8: The worst relative scores for the Fullhash attack for the PHP dataset expressed as a percentage change relative to the baseline No Comment scores.

64-shift-dataset	MRR	Top-1	Top-5	Top-10
NBoW	0.013	0.002	0.014	0.023
SelfAtt	0.224	0.140	0.204	0.384
ConvSelfAtt	0.354	0.255	0.455	0.545
1D-CNN	0.220	0.149	0.285	0.359
GraphCodeBERT	0.590	0.470	0.736	0.812
SynCoBERT	0.670	0.565	0.797	0.854

Table A.9: The worst absolute scores for the 64-shift-dataset attack for the Go dataset.

64-shift-dataset	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	79.03	93.55	82.72	79.82
SelfAtt	72.14	81.18	76.77	57.71
ConvSelfAtt	56.24	66.05	48.18	39.98
1D-CNN	70.94	78.47	65.87	58.55
GraphCodeBERT	30.59	40.36	20.69	14.62
SynCoBERT	24.55	33.14	15.21	10.86

Table A.10: The worst relative scores for the 64-shift-dataset attack for the Go dataset expressed as a percentage change relative to the baseline No Comment scores.

64-shift-dataset	MRR	Top-1	Top-5	Top-10
NBoW	0.016	0.004	0.016	0.030
SelfAtt	0.218	0.149	0.283	0.340
ConvSelfAtt	0.275	0.201	0.350	0.416
1D-CNN	0.145	0.088	0.194	0.248
GraphCodeBERT	0.464	0.350	0.601	0.689
SynCoBERT	0.379	0.274	0.501	0.586

Table A.11: The worst absolute scores for the 64-shift-dataset attack for the TL-CodeSum dataset.

64-shift-dataset	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	61.90	76.47	72.88	64.29
SelfAtt	40.60	50.17	35.24	29.75
ConvSelfAtt	34.68	42.07	29.29	24.50
1D-CNN	55.66	65.63	51.98	44.77
GraphCodeBERT	27.95	34.82	22.45	18.27
SynCoBERT	9.33	16.97	9.24	7.86

Table A.12: The worst relative scores for the 64-shift-dataset attack for the TL-CodeSum dataset expressed as a percentage change relative to the baseline No Comment scores.

64-shift-dataset	MRR	Top-1	Top-5	Top-10
NBoW	0.009	0.001	0.006	0.014
SelfAtt	0.155	0.085	0.211	0.299
ConvSelfAtt	0.170	0.095	0.232	0.332
1D-CNN	0.105	0.046	0.144	0.225
GraphCodeBERT	N/A	N/A	N/A	N/A
SynCoBERT	N/A	N/A	N/A	N/A

Table A.13: The worst absolute scores for the 64-shift-dataset attack for the Funcom dataset.

64-shift-dataset	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	0.00	0.00	0.00	0.00
SelfAtt	38.98	44.81	36.64	31.26
ConvSelfAtt	29.46	37.09	29.05	22.43
1D-CNN	49.03	61.67	50.17	42.16
GraphCodeBERT	N/A	N/A	N/A	N/A
SynCoBERT	N/A	N/A	N/A	N/A

Table A.14: The worst relative scores for the 64-shift-dataset attack for the Funcom dataset expressed as a percentage change relative to the baseline No Comment scores.

Appendix B

Complete list of tables for best performances

The results section only covered some of the best performances and outliers observed, so below we have assembled a list of all best scores for all available datasets.

Random Permutation	MRR	Top-1	Top-5	Top-10
NBoW	0.643	0.545	0.762	0.822
SelfAtt	0.516	0.417	0.626	0.696
ConvSelfAtt	0.497	0.404	0.601	0.669
1D-CNN	0.339	0.303	0.496	0.581
GraphCodeBERT	0.725	0.623	0.853	0.900
SynCoBERT	0.683	0.585	0.801	0.857

Table B.1: The best absolute scores for the Random Permutation attack for the Python dataset.

Random Permutation	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	-19.74	-25.87	-15.81	-12.76
SelfAtt	-13.66	-20.52	-9.25	-5.78
ConvSelfAtt	-9.23	-19.53	-2.04	1.33
1D-CNN	-9.35	-45.67	-19.52	-12.60
GraphCodeBERT	4.98	7.01	3.07	2.28
SynCoBERT	3.94	4.88	3.26	2.39

Table B.2: The best relative scores for the Random Permutation attack for the Python dataset expressed as a percentage change relative to the baseline No Comment scores.

3-Shift-Snippet	MRR	Top-1	Top-5	Top-10
NBoW	0.010	0.001	0.009	0.016
SelfAtt	0.188	0.113	0.256	0.336
ConvSelfAtt	0.297	0.209	0.388	0.459
1D-CNN	0.222	0.144	0.298	0.373
GraphCodeBERT	0.623	0.513	0.757	0.817
SynCoBERT	0.659	0.560	0.778	0.829

Table B.3: The best absolute scores for the 3-Shift-Snippet attack for the Java dataset.

3-Shift-Snippet	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	9.09	50.00	18.18	20.00
SelfAtt	31.14	39.25	29.86	23.81
ConvSelfAtt	8.90	11.81	7.62	8.20
1D-CNN	12.25	14.79	11.57	9.69
GraphCodeBERT	2.20	3.21	1.82	1.45
SynCoBERT	2.08	2.95	1.52	1.31

Table B.4: The best relative scores for the 3-Shift-Snippet attack for the Java dataset expressed as a percentage change relative to the baseline No Comment scores.

3-Shift-Snippet	MRR	Top-1	Top-5	Top-10
NBoW	0.009	0.001	0.007	0.014
SelfAtt	0.094	0.047	0.132	0.179
ConvSelfAtt	0.136	0.077	0.187	0.248
1D-CNN	0.056	0.023	0.075	0.108
GraphCodeBERT	0.538	0.432	0.665	0.741
SynCoBERT	0.450	0.348	0.567	0.643

Table B.5: The best absolute scores for the 3-Shift-Snippet attack for the Javascript dataset.

3-Shift-Snippet	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	10.00	0.00	22.22	12.50
SelfAtt	34.27	47.78	29.41	24.79
ConvSelfAtt	29.53	38.40	26.38	21.52
1D-CNN	5.08	14.81	5.06	-0.93
GraphCodeBERT	4.61	5.05	4.04	3.52
SynCoBERT	6.25	8.42	5.03	4.32

Table B.6: The best relative scores for the 3-Shift-Snippet attack for the Javascript dataset expressed as a percentage change relative to the baseline No Comment scores.

64-Shift-Snippet	MRR	Top-1	Top-5	Top-10
NBoW	0.713	0.622	0.825	0.874
SelfAtt	0.630	0.533	0.749	0.810
ConvSelfAtt	0.682	0.581	0.805	0.860
1D-CNN	0.627	0.531	0.740	0.805
GraphCodeBERT	0.737	0.636	0.861	0.908
SynCoBERT	0.685	0.586	0.805	0.860

Table B.7: The best absolute scores for the 64-Shift-Snippet attack for the PHP dataset.

64-Shift-Snippet	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	-5.01	-7.80	-2.61	-1.16
SelfAtt	9.09	11.46	6.38	5.26
ConvSelfAtt	-1.94	-1.40	-2.68	-2.38
1D-CNN	-1.95	-2.91	-1.51	-0.75
GraphCodeBERT	2.51	3.64	1.71	1.41
SynCoBERT	2.28	2.98	1.83	1.38

Table B.8: The best relative scores for the 64-Shift-Snippet attack for the PHP dataset expressed as a percentage change relative to the baseline No Comment scores.

3-Shift-Snippet	MRR	Top-1	Top-5	Top-10
NBoW	0.163	0.111	0.208	0.256
SelfAtt	0.727	0.652	0.817	0.857
ConvSelfAtt	0.790	0.732	0.863	0.889
1D-CNN	0.742	0.680	0.820	0.846
GraphCodeBERT	0.843	0.778	0.924	0.949
SynCoBERT	0.888	0.844	0.942	0.953

Table B.9: The best absolute scores for the 3-Shift-Snippet attack for the Go dataset.

3-Shift-Snippet	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	-162.90	-258.06	-156.79	-124.56
SelfAtt	9.58	12.37	6.95	5.62
ConvSelfAtt	2.35	2.53	1.71	2.09
1D-CNN	1.98	1.73	1.80	2.31
GraphCodeBERT	0.82	1.27	0.43	0.21
SynCoBERT	0.00	0.12	-0.21	-0.10

Table B.10: The best relative scores for the 3-Shift-Snippet attack for the Go dataset expressed as a percentage change relative to the baseline No Comment scores.

3-Shift-Snippet	MRR	Top-1	Top-5	Top-10
NBoW	0.044	0.022	0.059	0.077
SelfAtt	0.400	0.319	0.487	0.543
ConvSelfAtt	0.452	0.361	0.554	0.614
1D-CNN	0.350	0.269	0.439	0.495
GraphCodeBERT	0.621	0.512	0.755	0.826
SynCoBERT	0.501	0.408	0.642	0.725

Table B.11: The best absolute scores for the 3-Shift-Snippet attack for the TL-CodeSum dataset.

3-Shift-Snippet	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	-4.76	-29.41	0.00	8.33
SelfAtt	-8.99	-6.69	-11.44	-12.19
ConvSelfAtt	-7.36	-4.03	-11.92	-11.43
1D-CNN	-7.03	-5.08	-8.66	-10.24
GraphCodeBERT	3.57	4.66	2.58	2.02
SynCoBERT	-19.86	-23.64	-16.30	-13.99

Table B.12: The best relative scores for the 3-Shift-Snippet attack for the TL-CodeSum dataset expressed as a percentage change relative to the baseline No Comment scores.

64-Shift-Snippet	MRR	Top-1	Top-5	Top-10
NBoW	0.009	0.001	0.005	0.010
SelfAtt	0.237	0.148	0.325	0.423
ConvSelfAtt	0.251	0.156	0.344	0.450
1D-CNN	0.186	0.106	0.260	0.354
GraphCodeBERT	N/A	N/A	N/A	N/A
SynCoBERT	N/A	N/A	N/A	N/A

Table B.13: The best absolute scores for the 64-Shift-Snippet attack for the Funcom dataset.

64-Shift-Snippet	MRR [%]	Top-1 [%]	Top-5 [%]	Top-10 [%]
NBoW	0.00	0.00	16.67	28.57
SelfAtt	6.69	3.90	2.40	2.76
ConvSelfAtt	-4.15	-3.31	-5.20	5.14
1D-CNN	9.71	11.67	10.03	9.00
GraphCodeBERT	N/A	N/A	N/A	N/A
SynCoBERT	N/A	N/A	N/A	N/A

Table B.14: The best relative scores for the 64-Shift-Snippet attack for the Funcom dataset expressed as a percentage change relative to the baseline No Comment scores.

Appendix C

Hyperparameters for models

The used hyperparameters for all models are listed below. If hyperparameters are left unspecified, the default value is assumed.

C.1 CodeSearchNet

max num epochs	300
test batch size	1000
distance metric	cosine

Table C.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.

C.2 GraphCodeBERT

Specify *do_test* and *do_eval* or *do_train* for testing or training respectively. The remaining hyperparameters are identical between the two. For the training batch size, 64 is used only in the case of the Funcom dataset all other dataset are run with batch size 32.

C.3 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:

num training epochs	10
code length	256
data flow length	64
nl length	128
train batch size	32/64
eval batch size	64
learning rate	$2 * 10^{-5}$
seed	123456

Table C.2: GraphCodeBERT hyperparameters

effective queue size	4096
effective batch size	64/128
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 C.3: SynCoBERT hyperparameters for training. The effective batch size is set at 128 for Python datasets only. All other dataset are run at batchsize 64.

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 C.4: SynCoBERT hyperparameters for testing. Note that setting $--$*skip_training* will make most of the above set training parameters redundant.

Appendix D

CodeSearch Dataset

The original CodeSearch data corpus. [27]

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 D.1: The original dataset before cleaning and processing

Appendix E

Weighted scores calculation dataset

Below are the size of the datasets that are used to calculate the weighted scores. Consider that we must use the test sizes since the calculated scores are based on this.

PL	**Size**
Python	14,723
Java	10,953
PHP	14,014
Go	8,122
Javascript	3,235
TL-CodeSum	8,689
Funcom	99,032

Table E.1: The dataset used for calculating the weighted scores. Note that there may be small variations, depending on which dataset is used for the calculation. Above are the values for the No Comment dataset.