Master Thesis

August 20, 2020

Transfer Learning for Code Search

How Pre-training Improves Deep Learning on Source Code

Christoph Schwizer

of Niederuzwil SG, Switzerland (09-918-210)

supervised by

Prof. Dr. Harald C. Gall

Dr. Pasquale Salza

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Master Thesis

Transfer Learning for Code Search

How Pre-training Improves Deep Learning on Source Code

Christoph Schwizer

University of Zurich UZH

s.e.a.l. software evolution & architecture lab

Master Thesis

Author:	Christoph Schwizer, christoph.schwizer@uzh.ch
URL:	chrisly-bear.github.io/tlcs
Project period:	25.02.2020 – 25.08.2020

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

First of all, I would like to thank Prof. Dr. Harald C. Gall and Dr. Pasquale Salza for allowing me to write this thesis at the software evolution & architecture lab (s.e.a.l.) and providing me access to the university's cloud infrastructure, which proved invaluable for experimenting with such large machine learning models. I am especially grateful to Dr. Pasquale Salza for his support throughout this process. His critical thinking made sure that I pursued this research with the necessary scientific rigor.

Furthermore, I would like to express my gratitude towards the staff of the University of Zurich Library of Natural Sciences for making their premises available to students despite the difficult circumstances caused by the Coronavirus pandemic. Their protective measures ensured a safe environment and offered me a welcome alternative to working from home.

Finally, my thanks go to my family, friends, and girlfriend, who supported me in all their best ways.

Abstract

The Transformer architecture and transfer learning have marked a quantum leap in natural language processing (NLP), improving on the state of the art across a range of NLP tasks. This thesis examines how these advancements can be applied to and improve code search. To this end, we pre-train a BERT-based model on combinations of natural language and source code data and evaluate it on pairs of StackOverflow question titles and code answers. Our results show, that the pre-trained models consistently outperform the models that were not pre-trained. In cases where the model was pre-trained on natural language *and* source code data, it also outperforms our Elasticsearch baseline. Furthermore, transfer learning is particularly effective in cases where a lot of pre-training data is available and fine-tuning data is limited.

We demonstrate that NLP models based on the Transformer architecture can be directly applied to source code analysis tasks, such as code search. With the development of Transformer models that are designed more specifically for dealing with source code data, we believe the results on source code analysis tasks can be further improved.

Zusammenfassung

Die Entwicklung des Transformers und die Einführung von Transfer Learning markierten einen Quantensprung in der Computerlinguistik und führten zu massgeblichen Verbesserungen in einer Vielzahl von Problemen in der Analyse natürlicher Sprache. Diese Masterarbeit untersucht, wie diese Fortschritte genutzt werden können, um Codesuche zu verbessern. Dazu trainieren wir ein Modell auf natürlicher Sprache und Programmiersprachen vor und evaluieren es danach auf einem Datensatz bestehend aus StackOverflow Fragen und Antworten. Wir verwenden dafür ein Modell, dass auf BERT basiert. Unsere Resultate zeigen, dass die vortrainierten Modelle die Modelle, welche kein Vortraining erhielten, übertreffen. Wenn das Modell auf einer Kombination von natürlicher Sprache *und* Programmiersprachen vortrainiert wurde, erzielt es ebenfalls bessere Ergebnisse als unser Elasticsearch Modell. Es stellt sich heraus, dass Transfer Learning insbesondere dann effektiv ist, wenn grosse Datensätze für das Vortrainieren vorhanden, die Datensätze für das Zielproblem aber limitiert sind.

Wir zeigen, dass Transformer Modelle direkt auf Probleme der Source Code Anlayse, wie zum Beispiel Codesuche, angewendet werden können. Wir glauben, dass mit der Entwicklung von Transformer Modellen, welche spezifisch für die Codesuche entwickelt wurden, die Ergebnisse von Codesuche weiter verbessert werden können.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AST	abstract syntax tree	32
BPE	byte-pair encoding	17

Chapter 1

Introduction

The naturalness hypothesis of source code states that source code shows similar statistical properties as natural language [1]. It is thus possible to apply processing techniques from natural language to source code and improve predictive performance on various code analysis tasks. Indeed, over the last few years, there have been several applications of source code analysis, many of which successfully applied statistical models that were originally designed for NLP, such as n-gram models or recurrent neural networks (RNNs).

Such models need to be capable of dealing with the input data and extracting useful features from it. For natural language, for instance, the model needs to understand the relationships between the words in a text, such as finding the noun to which a pronoun refers or the subject to which verb belongs. For quite a while, RNNs have been the state-of-the-art model in many NLP sequence processing tasks. However, RNNs are limited in their ability to model long-distance relationships between sequence tokens. Recently, the Transformer architecture has overcome this limitation and outperforms RNNs in many NLP tasks [2–4]. However, when it comes to source code analysis, research on Transformer-based models is still scarce. Thus, it is not clear if Transformers are as capable at modeling source code as they are at modeling natural language. We believe that Transformers can indeed be effective in modeling source code as the problem is related to natural language modeling. Just as in natural language, detecting relationships between tokens in source code, such as the relationship between variable declaration and variable access or between opening and closing parentheses, is crucial to its understanding.

In addition to creating better language models, Transformers have become popular for their application of *transfer learning*. The idea behind transfer learning is to leverage a large corpus of data to *pre-train* a model and then *fine-tune* the model on a smaller dataset. Commonly, the pre-training dataset is extensive, easy to acquire, but unlabeled and not closely related to the problem we want to solve. On the other hand, the fine-tuning dataset is generally characterized by being small, difficult to acquire, but often labeled and closely related to our problem task. The intuition behind transfer learning is that, during pre-training, the model learns useful abstractions of the data which are effective for solving the problem or *"downstream"* task. In NLP, pre-training usually consists of learning a language model on large corpora of natural language text. Then, this pre-trained model can be employed in any particular downstream task, such as machine translation, sentiment analysis, part-of-speech tagging, summarization, or question answering. For source code, the same method can be applied: Train a language model on source code (i.e., a *"source code model"*), then apply it to downstream tasks, such as bug detection, authorship attribution, summarization, or code search. The promise of transfer learning is that unlabeled data, which often exists in abundance, can be used to train better models than would be possible with just training it on labeled data, which is usually scarce.

The goal of this work is to further research on Transformers and their effectiveness in modeling source code. In particular, we want to leverage the transfer learning capabilities of BERT, a state-of-the-art Transformer-based NLP architecture, by pre-training it on a large source code corpus. Because transfer learning has proven to be a valuable technique in NLP, we want to examine its potential in source code modeling. To assess the model's ability of extracting meaningful features from source code, we evaluate it on a code search task. We choose code search because, as we show in Section 3.1, it is crucial for a model to extract meaningful, semantic features in order for it to perform well on code search. Thus, if we can show the effectiveness of Transformers and transfer learning on code search, we can expect them to be an effective solution for other source code analysis problems as well.

We focus on the following research questions:

- **RQ1: Does a pre-trained English natural language model improve code search performance?** For a code search model to perform well, it needs to have a good understanding of the user's information need, which is expressed in the form of a natural language query. To this end, we use a pre-trained English language model to examine whether pre-training allows the code search model to learn better query representations and leads to better code search results.
- **RQ2: Does a pre-trained single-language source code model improve code search performance?** For the code search model to retrieve a relevant code snippet from the search corpus for a given query, it has to build good representations of the source code snippets in the search corpus. To answer this question, we pre-train a source code model on a specific programming language (e.g., Java), fine-tune it, and evaluate it on data of the same programming language (Java).
- **RQ3: Does a pre-trained English natural language model in combination with a pre-trained single-language source code model improve code search performance?** This research question is the combination of research questions RQ1 and RQ2. The hypothesis is that, if a pre-trained natural language model and a pre-trained source code model both lead to better code search performance, the combination of the two might lead to even better performance.
- **RQ4: Does a pre-trained multi-language source code model improve code search performance?** For this purpose, we will pre-train a source code model on several programming languages, fine-tune it, and evaluate it on a single programming language as well as on a multi-language search corpus.

The main contributions of this thesis are (1) the application of Transformers and transfer learning to code search and evaluation thereof, (2) the design of custom pre-training tasks for source code modeling and making the pre-trained source code models available to the research community, and (3) the mining and publishing of a code search evaluation dataset from StackOverflow. The pre-trained models and the evaluation dataset are available at chrisly-bear.github.io/tlcs.

This thesis is structured as follows: In Chapter 2, the state of the art in code search is presented and research gaps, which this work tries to fill, are identified. Chapter 3 provides the necessary theoretical background on code search and the BERT model. Chapter 4 specifies the exact methodology applied to answer each research question, while Chapter 5 explains the procedure that was used to gather the necessary data. The results of this research are presented in Chapter 6 and then discussed in Chapter 7. Finally, this thesis concludes in Chapter 8 with a summary of the findings and the contributions of this work, as well as an outlook on future research in this area.

Chapter 2

State of the Art in Code Search

This chapter provides an overview of the current research on code search and is divided into three sections. Section 2.1 reviews works that make use of Transformer-based models, while Section 2.2 highlights works that use different models. Finally, the difficulties of evaluating code search and how different evaluation datasets try to solve these are described in Section 2.3.

2.1 Code Search using Transformers

Little research has gone into the application of Transformers to code search. Husain et al. [5] build a range of neural network models and compare their performance on the code search task. One of their models is the BERT-based *Self-Attention* model. They train and evaluate their models on pairs of docstring-code pairs mined from open-source repositories on GitHub. Even without pre-training, the Self-Attention model shows good results when trained and evaluated on the same dataset[1]. Our work closely resembles that of Husain et al. as we use the same multimodal embedding architecture (see Section 3.2) and BERT for encoding the natural language query and the source code snippets. The main difference to their approach is that we *pre-train* the encoders before applying them to code search. Furthermore, while we also use GitHub data for pre-training, we fine-tune and evaluate our model on StackOverflow data, which we believe better approximates code search than docstring-code pairs.

Feng et al. [6] build on the work of Husain et al., but instead of using cosine similarity between the outputs of two separate encoders, they concatenate the query and code sequence, feed it to a *single* encoder, and measure the similarity between query and code snippet by using a summarizing token in the output sequence. They then use this approach to pre-train their model "bimodally", i.e., instead of pre-training the query and code encoder separately as we do in this work, they pre-train their single encoder on sequence pairs of natural language and source code. An important difference to their method is that our pre-training data (mined from GitHub) is different from our fine-tuning and evaluation data (mined from StackOverflow), which we argue reflects a more common transfer learning scenario.

2.2 Code Search using Non-Transformer Models

Allamanis et al. [7] learn bimodal representations of source code and natural language and apply them to code search. They train a statistical model that estimates the probability distribution

[1]They also evaluate their models on a manually annotated dataset. In that evaluation, the Self-Attention model performs poorly. The authors explain this by the discrepancy between docstring and search queries.

$P(\mathcal{C}|\mathcal{L})$, where $\mathcal{C}$ is a code snippet and $\mathcal{L}$ is a natural language sequence. They make use of a parse tree representation of source code and evaluate their model on two retrieval tasks: (1) retrieving source code given a natural language query and (2) retrieving a source code description given a code snippet.

Iyer et al. [8] extend the work of Allamanis et al. by training a long short-term memory (LSTM) neural network with attention and applying it to code summarization, i.e., generating natural language descriptions from code snippets. While code summarization takes code as input and generates natural language as output, they can still apply their model to code search: Given a natural language query as input and a code snippet from the search corpus, they calculate the conditional probability of the query being generated from the code snippet. By calculating the conditional probability for all code snippets in the search corpus they can return the ones with the highest probabilities. Furthermore, because their model uses attention, they can visualize on which parts of the source code the model focuses while generating the summary tokens.

Ye et al. [9] learn token embeddings based on the Skip-gram model by Mikolov et al. [10] from (1) texts which mix natural language and code tokens, such as programming tutorials and API references, (2) pairs of source code and natural language, and (3) the English Wikipedia (which also contains entries where natural language and code tokens are mixed). They evaluate their embeddings on two code search tasks: bug localization (given a bug report, find affected source code files) and API recommendation for StackOverflow questions. They found that using word embeddings as additional features leads to performance improvements on the bug localization task.

Sachdev et al. [11] derive a purely token-based approach to code search. They use the fastText algorithm [12] to learn embeddings for source code tokens. Then, they use these embeddings to encode both the source code and the search query. The advantage of their approach is that they do not require any labeled training data. However, this approach relies on the assumption that source code and search queries use the same vocabulary and thus it ignores the lexical gap that might exist between them.

Wan et al. [13] combine multiple source code representations: source code tokens, abstract syntax trees (ASTs), and control flow graphs. By using attention, they hypothesize that the neural network will automatically select the most useful features from the different representations. Furthermore, the attention mechanism allows them to visualize the features on which the neural network concentrates during retrieval. To train and evaluate their approach, they created a dataset of more than 28 000 C code snippets and their description. They compare their results with two other code search models and improve on both of them.

2.3 Code Search Datasets

A peculiarity of the code search problem is that it is not immediately clear how to evaluate it. For one, there are slight differences in the specific application of code search. One can imagine that doing code search on a company internal project or repository comes with different search intents than searching for code on the internet. The intent for the former might be to do a refactoring of the code. Thus, it will be more concerned with recall, i.e., finding all relevant classes in the project which need refactoring. The latter might be aimed at finding an implementation to a programming problem, say, how to generate random numbers in Python. In this scenario, we are probably more interested in good precision, i.e., finding a few solutions that are relevant, rather than recall, i.e., finding all possible solutions.

Furthermore, even if the intent is clear, it might not be clear how relevant a search result actually is. A search result might be relevant to one person, but less relevant to another. The relevance might depend on the searcher's programming style preference, e.g., if they prefer a compact code snippet compared to a more verbose but more legible code snippet. It might also depend on the searcher's knowledge, i.e., a more experienced developer might be satisfied with a crude code snippet which points them in the right direction whereas a novice developer might only find results relevant that solve exactly the problem that they had. Relevance might additionally depend on the context in which the search is performed. For instance, a company developer might not find a code snippet relevant because it uses a third party library which is not allowed in their company, whereas a freelance developer might consider such a library and find the search result perfectly relevant. Typically, the agreement among different annotators, the so-called *interannotator agreement*, is measured with Cohen's kappa coefficient κ [14]. κ is one if the annotators assign the same relevance to all query-code pairs, and it is zero if the relevance assignments match a random allocation.

Finally, the most difficult challenge when it comes to evaluating code search is to find annotated data. The annotated data consists of triples (q, d, r), where q is the search query, d a document in the search corpus, in our case a code snippet, and r the relevance annotation, i.e., how relevant document d is with regards to query q. The annotation can be binary (relevant, not relevant) or on a scale (e.g., 1 to 5 where 1 is not relevant at all and 5 is very relevant). Ideally, for a given set of queries $Q = \{q_1 \dots q_n\}$ and a search corpus $D = \{d_1 \dots d_m\}$, the evaluation data contains annotations for every combination of query and document. To account for the above-mentioned differences in assessing the relevance of a code snippet, it is preferred to have several annotations from different annotators for each query-document pair. Of course, creating such an evaluation dataset of reasonable size is infeasible. Assuming we have $n = 100$ queries and a search corpus of size $m = 5\,000$, we would have to annotate $100 * 5\,000 = 500\,000$ samples.

Fortunately, we can reasonably evaluate a code search system even with small evaluation datasets, e.g., by using measures such as mean reciprocal rank (MRR) or discounted cumulative gain (DCG) (see Section 3.1.1). This approach is used in several previous works [5, 7, 8].

The same observation — namely that they are difficult to produce — can be made for training data. However, the scarcity problem is more profound when it comes to training because deep neural networks require a lot of data to reasonably approximate the underlying target function. Optimally, there would exist a large annotated dataset of the described form (query, document, annotation), which can be used for both training and evaluation of a code search system.

A common way to overcome the training data scarcity problem is to use documentation data instead of query data. Documentation is abundantly available in source code, especially on the level of methods, which are often well documented. By using these *docstrings* we can build a large enough training corpus consisting of docstring-method pairs. This is what Husain et al. do [5]. One of their contributions is that they provide the CodeSearchNet dataset, a benchmark for training and evaluating code search models. It consists of over two million text-code pairs, namely methods and their documentation, and an additional four million methods without documentation. Moreover, Husain et al. created a human-annotated evaluation corpus consisting of 4 026 query-code pairs. Each pair states how relevant a method is to the given query. At this time, the evaluation corpus is not made available by the authors, which is why we mined our own training and evaluation corpus. Contrary to the CodeSarchNet dataset, which is mined from public GitHub repositories and consists of method-docstring pairs, our dataset was mined from StackOverflow data and consists of pairs of question titles and code answers (see Chapter 5).

Chapter 3

Theoretical Background

The theoretical framework on which this thesis builds is explained in the following chapter. First, the code search problem is defined in Section 3.1. Then, Section 3.2 illustrates the multimodal embedding model (MEM). Finally, a brief overview is given of the Transfomer and BERT models and on transfer learning in Section 3.3.

3.1 Code Search

Code search or *code retrieval* is the task of retrieving source code from a large code corpus given a natural language user query (e.g., "how to convert string to int in java"). The goal of the retrieval system is to return source code documents that are most relevant to the user query. In other words, the semantics of the source code should correspond to the semantics of the natural language query. The corpus may be a large public repository, such as GitHub, or it can be an organization's private code base.

Code search can be an effective tool for software developers. It helps them quickly find examples of how to implement a particular feature, find software libraries which provide a certain functionality, navigate through their own code base, or even find pieces of source code which need to be changed in order to accommodate user concerns such as feature requests or bug fixes [9, 15].

Traditional retrieval systems are based on token matching. They compare the tokens in the search query with the tokens in the search corpus' documents and return those documents with the biggest overlap between query tokens and document tokens (often, individual tokens are weighted by their frequency and inverse document frequency or *tf-idf* [16]). This approach has proven very useful for matching natural language queries with natural language documents, such as books or webpages. However, when matching natural language queries with source code documents, it is less effective. One reason for this is that the tokens in the query do not necessarily match the tokens in the source code. For example, the query "read json data" would not find a method called `deserializeObjectFromString` (tokenized into `deserialize object from string`) even though it might be relevant to the query. This discrepancy between the query language and the language in the documents of the search corpus is referred to as *lexical gap* or *heterogeneity gap*.

3.1.1 Evaluation Metrics

Typical evaluation metrics for information retrieval (IR) are recall, precision, F-measure, discounted cumulative gain (DCG), and mean reciprocal rank (MRR).

Recall determines how many of all the relevant documents in the search corpus were retrieved. For example, if a search returns 10 documents, 3 of which are relevant, and our corpus contains 8 relevant documents in total, then the recall is $3/8 = 37.5\%$. An IR system can easily reach 100% recall by returning the entire search corpus. For this reason, recall is sometimes limited to the first k results (e.g., $k = 10$).

Precision measures how many of the retrieved documents are relevant. For example, if a search returns 10 documents and 3 of these are relevant, then the precision is $3/10 = 30\%$. A variation of precision, in which only the first k results are considered is called "precision at k". "Average precision" additionally accounts for the order of the search results and assigns higher values to relevant documents that appear higher up in the list of search results, while "mean average precision" summarizes the average precision values over several queries by their mean.

F-measure is the harmonic mean of recall and precision. IR systems commonly face a recall-precision tradeoff. The F-measure is thus a useful metric as it considers both recall and precision.

If, instead of just being relevant or not relevant, the documents in our search results can be judged by *how* relevant they are, e.g., on a scale from 1 to 5, we can evaluate our search results using DCG. DCG takes into account both the relevance judgments of the retrieved documents and their rank. To reach high DCG values, an IR system has to return highly relevant documents before less relevant ones.

The metrics above only make sense if *several* documents in the search corpus are relevant. If instead, there is exactly *one* relevant document in the corpus, the reciprocal rank is a more suitable metric. The reciprocal rank is the inverse rank of the relevant document. For instance, if the relevant document is returned at position 4, the reciprocal rank is $1/4 = 0.25$, if it is returned at position 1, the reciprocal rank is $1/1 = 1$. The intuition behind the reciprocal rank is that, if the relevant document appears at position k, the user has to go through k documents to find the relevant one. At this point, the precision is $1/k$, which is also the reciprocal rank. Finally, the mean reciprocal rank (MRR) is the average of multiple reciprocal ranks, e.g., from multiple queries.

3.1.2 Related Problems

There are two problems that are closely related to the code search problem, namely *code generation* and *code summarization*. The methods devised in this thesis are thus relevant to these problems as well.

Code Generation. Similarly to code search, the input to the code generation problem is also a natural language query. However, the output is not retrieved from a corpus of existing code snippets, but instead generated by the model itself. Thus, models which solve the code generation problem are *generative* models, whereas models in code search are purely analytical models. The code generation problem can be seen as a (neural) machine translation problem, i.e., the goal is to translate from natural language to source code.

Code Summarization. Code summarization is the process of generating natural language from source code. From an input-output perspective, it is the opposing problem to code search and code generation, i.e., the input is a source code snippet and the output a natural language snippet. Like with code generation, the models used in code summarization are generative, i.e., their goal is to create unseen snippets of natural language. The goal of code summarization is to use the generated natural language snippet as documentation of the source code or to improve commit messages.

3.2 The Multimodal Embedding Model

A multimodal embedding model (MEM) builds vector representations ("embeddings") for each mode, e.g., natural language and source code, such that similar concepts are located in the same region of a shared vector space[1]. Recent work has relied on multimodal embeddings to overcome the lexical gap [17, 18].

Multimodal embeddings are especially useful for code search as they allow for retrieval using a simple distance-based similarity metric, such as cosine similarity. At query time, the natural language query is encoded into its vector representation and compared to all source code vectors in the search corpus. Finally, the source code documents are returned as a list sorted by their distance to the query vector in increasing order. Under the premise that the multimodal embeddings encode semantic similarity between natural language and source code effectively, the search results at the beginning of the list should be highly relevant to the user as they capture the meaning of their query.

To transform a natural language query into its vector representation, the MEM runs the query through an encoder E_q while another encoder E_c transforms a source code document into its vector representation. More formally, $E_q : \mathcal{Q} \rightarrow \mathbb{R}^d$ and $E_c : \mathcal{C} \rightarrow \mathbb{R}^d$ are embedding functions, where $\mathcal{Q}$ is the set of natural language queries, $\mathcal{C}$ is the set of source code documents, and $\mathbb{R}^d$ is the space of real-valued vectors of size d. Figure 3.1 depicts the architecture of a MEM for code search.

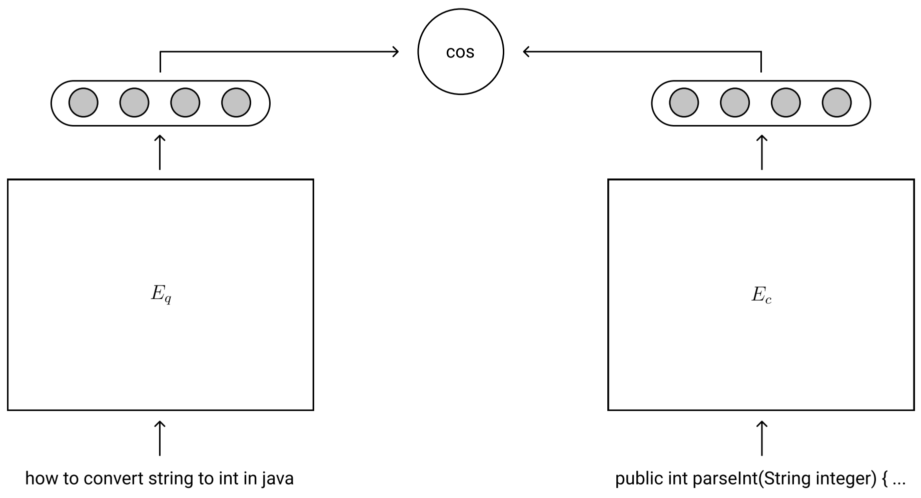
cos
E_q
E_c
how to convert string to int in java
public int parseInt(String integer) { ...

Figure 3.1: The multimodal embedding model architecture. A query is transformed into its vector representation by the query encoder E_q. Likewise, a source code document is encoded by the source code encoder E_c. Then, the similarity between query and source code is measured using cosine similarity between the two vector representations.

The encoder can be any model which converts the input data into its vector representation. In the past, recurrent neural networks (RNNs) were often used for the source code encoder [8, 17, 19–21]. In this work, we will use BERT [3] as the encoder architecture for both the source code encoder as well as the query encoder.

[1]The shared vector space is also called *semantic space*.

3.3 Transformers, BERT, and Transfer Learning

When the Transformer architecture was introduced, it replaced RNNs as the state of the art in neural machine translation (NMT) [2]. RNNs process each token in a sequence sequentially. This leads to loss of information of far-away tokens, i.e., by the time the RNN arrives at the last token, the signal from the first token has become very small. Attention mechanisms mitigate this problem, allowing the RNN to focus on arbitrary preceding tokens in the sequence. Despite attention, the nature in which RNNs processe data is still sequential. The Transformer changes this by removing recurrence and handling the entire input sequence in parallel. It achieves this by relying solely on attention, whereby a weight for each token pair in the input sequence is calculated. This component is called an *attention head* and lets the model represent relationships between tokens in the sequence. In fact, it has been shown, that attention heads learn syntactic features of a (natural) language, such as prepositions and their corresponding object or nouns and their determiner [22]. The Transformer proposed by Vaswani et al. uses eight attention heads, which means that more language features can be encoded than with a single attention head.

The parallel nature of the Transformer facilitates faster training, which in turn enables training on much larger datasets. This is a key aspect that BERT ("Bidirectional Encoder Representations from Transformers") [3] exploits. BERT was trained on an English corpus of 3.3 billion words. The training tasks were masked language modeling (MLM_{PT}) and next sentence prediction (NSP_{PT}). In MLM_{PT}, some tokens in the sequence are masked by as special [MASK] symbol and the model has to predict the token that was masked out. In NSP_{PT}, the model is given two random sentences from the corpus and it has to decide whether they appear in sequence of one another. These tasks guide BERT to learn a language model of the English language. A language model can be practical in itself, e.g., it can be used to give typing suggestions [23]. However, in the case of BERT, the language modeling tasks were only used as parameter initialization for different training tasks, such as question answering and language inference. The strategy of "pre-training" a model on a different task before training (or "fine-tuning") it on the target task is called *transfer learning.* The parameter weights learned during pre-training help the model to perform better on the target (or "downstream") task than with random weight initialization.

BERT uses the same architecture as the Transformer with one distinction: While the Transformer employs an encoder-decoder architecture, BERT only uses encoders[2]. What sets BERT apart from similar Transformer-based models is its bidirectionality. As opposed to Radford et al.'s GPT model [4] which processes the input sequence only in one direction (e.g., from left to right), BERT handles the input sequence in both directions (from left to right *and* from right to left) simultaneously. This enhances the capabilities of the attention heads as they can focus on preceding and subsequent tokens. Consequently, the token embeddings that BERT creates are dependent on the surrounding tokens and therefore called *contextualized* embeddings. Contextualized embeddings are more capable than context-free embeddings, such as those generated by word2vec or GLoVe [10, 24], as they can distinguish between words that spell the same but have different meaning (e.g., "minute" in "she pays attention to every minute detail" vs. "he was one minute late").

[2]The decoder is required for generating an output sequence, e.g., for machine translation. BERT does not generate output sequences and is instead designed to only analyze the input sequence.

Chapter 4

Approach to Evaluating Transfer Learning for Code Search

To examine the effectiveness of transfer learning for code search, we devised several experiments with different configurations of pre-training and fine-tuning data. To simulate a typical transfer learning scenario, in which the pre-training data differs from the fine-tuning data, we used two distinct datasets. The pre-training dataset consists of function definitions from open-source projects on GitHub, while the fine-tuning dataset contains StackOverflow questions and corresponding code snippet answers.

The following chapter includes a description of these datasets in Section 4.1. An explanation of the pre-training and fine-tuning setup, such as hyperparameters, training time, and hardware, is given in Sections 4.2 and 4.3, respectively. Then, Section 4.4 details how the trained models were evaluated. Finally, in Section 4.5, limitations of this work are discussed.

4.1 Datasets

4.1.1 Pre-training Dataset

For pre-training, we chose the CodeSearchNet [5] dataset, which was mined from GitHub repositories and consists of function definitions across six different programming languages (JavaScript, Java, Python, PHP, Go, and Ruby). We could have used any other raw source code dataset or mined one ourselves. We decided on the CodeSearchNet dataset because it readily provides a large set of source code samples in a machine readable format.

To reduce the noise in the pre-training data, all documentation and comments were removed using a parser[1]. Otherwise, the data was not further processed.

Table 4.1 lists the size of our pre-training dataset. In addition to the dataset sizes of the individual languages, the combined size of all datasets (All) as well as the combined size of the three largest datasets (JavaScript, Java, and Python) (Top_{GH}) are listed. We pre-trained models on the JavaScript, Java, and Python datasets, the Top_{GH} dataset, and the All dataset. To keep the number of experiments attainable, we forwent pre-training on the smaller PHP, Go, and Ruby datasets. Our largest dataset (All) contains around 350 million tokens. In comparison, BERT

[1]For parsing, we used TreeSitter (tree-sitter.github.io).

was pre-trained on a corpus of 3.3 billion words (0.8 billion words from the BooksCorpus and 2.5 billion words from English Wikipedia) [3].

	number of functions	number of tokens
JavaScript	**1 857 835**	**128 430 003**
Java	**1 569 889**	**75 654 447**
Python	**1 156 085**	**50 551 794**
PHP	977 821	53 352 522
Go	726 768	37 075 579
Ruby	164 048	5 495 442
Top_{GH}	**4 583 809**	**254 636 244**
All	**6 452 446**	**350 559 787**

Table 4.1: Size of the pre-training datasets (sorted by number of functions). The datasets that were used for pre-training are marked in bold.

4.1.2 Fine-tuning and Evaluation Dataset

For fine-tuning and evaluation, we decided to mine our own dataset consisting of question-answer pairs from StackOverflow. We use the question's title as the natural language query and the answer's code snippets as the source code document to be retrieved from the search corpus. Refer to Chapter 5 for a detailed description of the data mining process.

We believe that StackOverflow questions are a good proxy for search queries, especially since the platform is mostly used for the purpose of finding code solutions. Additionally, using StackOverflow data allows us to build a large enough dataset so that we can both fine-tune and evaluate our models, which would have been very difficult to achieve with human annotations.

For the fine-tuning, we applied 3-fold cross-validation by splitting the entire dataset into three equal folds and using two folds for training and one for evaluation. We further split the data from the two training folds into 90% training and 10% validation data, leaving us with the dataset sizes of Table 4.2.

4.2 Pre-training

The pre-training procedure on source code is identical to the pre-training of Devlin et al.'s $\text{BERT}_{\text{BASE}}$ model on natural language [3], with only a slight difference in the pre-training tasks: Instead of the NSP_{PT} task for pre-training on natural language, for source code, we apply next line prediction (NLP_{PT}). In this binary classification task, the model has to decide for any two given lines of source code `A` and `B`, whether or not `B` appears directly after `A`. To train the model on this task, it is fed with samples from our pre-training dataset, in which 50% of the time, line `B` actually follows line `A` and in the other 50% of the cases, `B` is a randomly chosen line from the corpus and does not immediately follow line `A`.

We call the MLM_{PT} task for source code *masked source code modeling* (MCM_{PT}) to make it clear that the model is pre-trained on source code data instead of natural language. Other than that, the MLM_{PT} and MCM_{PT} tasks are identical, i.e., the model has to predict masked out tokens in the input sequence. Like Devlin et al., we selected 15% of the tokens in the input sequence for masking. In contrast, we only used a maximum sequence length of 256 tokens, whereas Devlin

		train	valid	test	Total Size
JavaScript	fold 1	51 030	5 670	28 349	85 049
	fold 2	51 029	5 670	28 350	
	fold 3	51 029	5 670	28 350	
Java	fold 1	42 716	4 746	23 732	71 194
	fold 2	42 717	4 746	23 731	
	fold 3	42 717	4 746	23 731	
Python	fold 1	52 339	5 815	29 077	87 231
	fold 2	52 339	5 815	29 077	
	fold 3	52 339	5 815	29 077	
TOP$_{\text{FT}}$	fold 1	146 085	16 231	81 158	243 474
	fold 2	146 085	16 231	81 158	
	fold 3	146 085	16 231	81 158	

Table 4.2: Size of the fine-tuning datasets before and after the three-fold split.

et al. used 512. The reason for this is that longer sequences require exponentially more memory during training and would thus not have fit in our GPU memory without a drastic reduction in batch size. Moreover, as we will see in Section 5.1, the average sequence length in our fine-tuning dataset is less than 256, so most of the samples can be encoded by our model in their entirety. With a sequence length of 256, the maximum batch size that still fit in our GPU memory was 62.

Like Devlin et al., we tokenized the source code sequence using WordPiece tokenization [25] using a vocabulary size of 30 522 tokens. Similar to Husain et al. [5], we kept the case information. We second their choice to treat source code case-sensitively as case information carries a valuable signal, such as the distinction between constants and variables or between class declarations and method declarations.

Devlin et al. pre-trained their model for 1 million steps, which equals about 40 epochs on their dataset. Since our pre-training datasets are much smaller (see Section 4.1.1) and we used a different batch size and different sequence length, we adjusted the number of training steps accordingly in order to train for about 40 epochs as well. For example, our JavaScript dataset consists of 128 430 003 tokens. With a sequence length of 256 tokens and a batch size of 62 sequences there are 15 872 tokens in a batch. Thus, we reach 40 epochs after pre-training for 323 665 steps ($323\,665 * 15\,872/128\,430\,003$). Because of the smaller number of training steps we also reduced the number of warmup steps. Table 4.3 lists the hyperparameters we used for pre-training, while Table 4.4 shows the number of training steps for all model configurations.

Pre-training was executed on a single Nvidia Tesla V100 GPU with 32 GB of memory and took between 1.6 and 11 days. The individual pre-training times are shown in Table 4.4 along with the final performance of each model on the two pre-training tasks. The high accuracy values for both tasks suggest that pre-training was successful and the models learned useful abstractions of source code.

4.3 Fine-tuning

The fine-tuning procedure for code search closely follows the design by Husain et al. [5]. We use the same multimodal embedding architecture with two encoder models, one for the natural language

	BERT$_{\text{BASE}}$ [3]	ours
optimizer	Adam	Adam
learning rate	10^{-4}	10^{-4}
β_1	0.9	0.9
β_2	0.999	0.999
L2 weight decay	0.01	0.01
learning rate decay	linear	linear
dropout probability	0.1	0.1
activation function	gelu	gelu
masking rate	0.15	0.15
hidden size	768	768
intermediate size	3 072	3 072
attention heads	12	12
hidden layers	12	12
vocabulary size	30 522	30 522
maximum sequence length	**512**	**256**
batch size	**256**	**62**
learning rate warmup steps	**10 000**	**1 000**

Table 4.3: Pre-training hyperparameters compared to the original BERT$_{\text{BASE}}$ pre-training hyperparameters. The differing hyperparameters are marked in bold.

	JavaScript	Java	Python	TOP$_{\text{GH}}$	ALL
number of training steps	323 665	190 662	127 399	641 725	883 468
training time in days	4	2.4	1.6	8	11
MCM$_{\text{PT}}$ accuracy	90%	87%	86%	88%	88%
NLP$_{\text{PT}}$ accuracy	98%	96%	98%	95%	95%

Table 4.4: Pre-training steps, pre-training time, and model performance after pre-training.

query and one for the source code snippet (see also Section 3.2), and the same training objective, namely reducing the distance between query and code vector in the vector space.

To investigate our research questions from Chapter 1, we devised several experiments with different combinations of pre-trained models. All experiments were run on an Nvidia Tesla V100 with 32 GB of GPU memory. A single fold (10 epochs) took between 50 minutes and 3 hours, depending on the size of the fine-tuning dataset. We ran all experiments with 3-fold cross-validation using the datasets from Table 4.2.

4.3.1 Experiments with Pre-trained Models

Pre-trained query model (RQ1). First, we used Devlin et al.'s [3] pre-trained English model BERT$_{\text{BASE}}$ (uncased), which is publicly available[2], and applied it to the query encoder E_q. This means that the weights of the query encoder were initialized with the weights of the pre-trained English model. In this scenario, the code encoder E_c was not pre-trained, i.e., its weights were

[2]github.com/google-research/bert/#pre-trained-models

initialized with random values. Table 4.5 shows the experiments we conducted to assess the effect of using a pre-trained English model on code search performance.

pre-training E_q	pre-training E_c	fine-tuning	evaluation	experiment label
English	none	JavaScript	JavaScript	`[EN_no]-(JS)-{JS}`
English	none	Java	Java	`[EN_no]-(JA)-{JA}`
English	none	Python	Python	`[EN_no]-(PY)-{PY}`

Table 4.5: Experiments using a pre-trained English model for the query encoder.

Pre-trained code models (RQ2). Then, we used our own pre-trained source code models (see Section 4.2) to initialize the weights of the code encoder E_c. This time, the weights of the query encoder E_q were initialized with random values. Table 4.6 shows our experiments for assessing the effect of pre-trained source code models on code search performance. We limited the experiments to cases in which the pre-training was conducted on the same programming language as the fine-tuning. We note that cross-language learning, such as using a pre-trained Python model to fine-tune on Java data, could make sense in a scenario where the target language is so rare that there is not enough data available to justify pre-training. In such a scenario, however, we expect a pre-trained *multi-language* source code model, i.e., a model that was trained on a mix of programming languages, to yield better results. We examined multi-language source code models in RQ4.

pre-training E_q	pre-training E_c	fine-tuning	evaluation	experiment label
none	JavaScript	JavaScript	JavaScript	`[no_JS]-(JS)-{JS}`
none	Java	Java	Java	`[no_JA]-(JA)-{JA}`
none	Python	Python	Python	`[no_PY]-(PY)-{PY}`

Table 4.6: Experiments using a pre-trained source code model for the code encoder.

Pre-trained query and code models (RQ3). As a next step, we combined pre-trained query and code models to see how well they complement each other. For these experiments, both the weights of the query encoder E_q and the code encoder E_c were restored from the respective pre-trained model. Table 4.7 lists the different experiment configurations.

pre-training E_q	pre-training E_c	fine-tuning	evaluation	experiment label
English	JavaScript	JavaScript	JavaScript	`[EN_JS]-(JS)-{JS}`
English	Java	Java	Java	`[EN_JA]-(JA)-{JA}`
English	Python	Python	Python	`[EN_PY]-(PY)-{PY}`

Table 4.7: Experiments using a pre-trained English model and a pre-trained source code model.

Pre-trained multi-language code models (RQ4). Finally, we examined source code models that were pre-trained on several programming languages. We pre-trained two such models: one on JavaScript, Java, and Python data (TOP$_{GH}$) and another one on JavaScript, Java, Python, PHP, Go, and Ruby data (ALL). Again, we distinguished between only pre-training the query encoder E_q, only pre-training the code encoder E_c, and pre-training both. For these experiments, in addition to the single-language datasets, we fine-tuned and evaluated the models on a multi-language dataset consisting of JavaScript, Java, and Python samples (TOP$_{SO}$).

pre-training E_q	pre-training E_c	fine-tuning	evaluation	experiment label
English	none	TOP$_{SO}$	TOP$_{SO}$	`[EN_no]-(TOP)-{TOP}`
none	TOP$_{GH}$	JavaScript	JavaScript	`[no_TOP]-(JS)-{JS}`
none	TOP$_{GH}$	Java	Java	`[no_TOP]-(JA)-{JA}`
none	TOP$_{GH}$	Python	Python	`[no_TOP]-(PY)-{PY}`
none	TOP$_{GH}$	TOP$_{SO}$	TOP$_{SO}$	`[no_TOP]-(TOP)-{TOP}`
none	ALL	JavaScript	JavaScript	`[no_ALL]-(JS)-{JS}`
none	ALL	Java	Java	`[no_ALL]-(JA)-{JA}`
none	ALL	Python	Python	`[no_ALL]-(PY)-{PY}`
none	ALL	TOP$_{SO}$	TOP$_{SO}$	`[no_ALL]-(TOP)-{TOP}`
English	TOP$_{GH}$	JavaScript	JavaScript	`[EN_TOP]-(JS)-{JS}`
English	TOP$_{GH}$	Java	Java	`[EN_TOP]-(JA)-{JA}`
English	TOP$_{GH}$	Python	Python	`[EN_TOP]-(PY)-{PY}`
English	TOP$_{GH}$	TOP$_{SO}$	TOP$_{SO}$	`[EN_TOP]-(TOP)-{TOP}`
English	ALL	JavaScript	JavaScript	`[EN_ALL]-(JS)-{JS}`
English	ALL	Java	Java	`[EN_ALL]-(JA)-{JA}`
English	ALL	Python	Python	`[EN_ALL]-(PY)-{PY}`
English	ALL	TOP$_{SO}$	TOP$_{SO}$	`[EN_ALL]-(TOP)-{TOP}`

Table 4.8: Experiments using pre-trained multi-language source code models

Hyperparameters For the fine-tuning of our multimodal embedding model, we used the hyperparameters listed in Table 4.9. Since our fine-tuning procedure is very similar to the one by Husain et al., we kept their hyperparameters whenever possible. We increased the maximum sequence length of the code encoder to 256 because the average code snippet in our fine-tuning dataset has around 180 tokens (see Table 5.2) and because we pre-trained our code encoder with the same maximum sequence length of 256. We kept the maximum sequence length for the query encoder at 30 tokens as our average query contains only around 9 tokens. Thus, we do not expect better performance with a larger sequence length. We used the LAMB optimizer instead of Adam to reduce training time [26] and limited training to 10 epochs. In contrast, Husain et al. trained for a maximum of 500 epochs but applied early stopping, i.e., their training stopped if the MRR did not improve for 5 epochs ("patience" hyperparameter). 32 was the largest batch size that would fit in the GPU memory of our Nvidia Tesla V100 (32 GB).

The BERT-specific hyperparameter values were mostly dictated by our pre-trained models. For example, the English model provided by Devlin et al. was pre-trained on a vocabulary of 30 522 tokens. To keep the hyperparameters between the code and query encoder as similar as possible, we also pre-trained our source code model on a vocabulary size of 30 522 tokens. The same holds true for the hidden size and the intermediate size. The only hyperparameters we changed from our pre-trained models were the number of attention heads and number of hidden layers (both had a value of 12 during pre-training). We decided to use Husain et al.'s values (8 and 3, respectively) because we observed faster convergence of the models during training with those values, presumably because of the reduced model complexity.

	Husain et al. [5]	ours
multimodal embedding model hyperparameters		
learning rate	$5 \cdot 10^{-4}$	$5 \cdot 10^{-4}$
learning rate decay	0.98	0.98
momentum	0.85	0.85
dropout probability	0.1	0.1
maximum sequence length (query)	30	30
maximum sequence length (code)	**200**	**256**
optimizer	**Adam**	**LAMB**
maximum training epochs	**500**	**10**
patience	**5**	**10**
batch size	**450**	**32**
BERT-specific hyperparameters (apply to both code and query encoder)		
activation function	gelu	gelu
attention heads	8	8
hidden layers	3	3
hidden size	**128**	**768**
intermediate size	**512**	**3 072**
vocabulary size	**10 000**	**30 522**

Table 4.9: Fine-tuning hyperparameters compared to Husain et al. The differing hyperparameters are marked in bold. The BERT-specific hyperparameters are equal for both the query encoder and the code encoder and listed only once for simplicity.

Vocabulary, tokenization, and capitalization. One difference between Husain et al. and our approach is the tokenization and vocabulary building process. Because we used pre-trained models in our experiments, we had to use the vocabulary learned by the pre-trained models since the models' pre-trained weights depend on their specific encoding of tokens. Husain et al., on the other hand, did not rely on parameter weights of pre-trained models, which is why they built a new vocabulary from the fine-tuning data (the training set). They used byte-pair encoding (BPE) [27] for that process, while the pre-trained English model ($\text{BERT}_{\text{BASE}}$) built its vocabulary using WordPiece tokenization [25]. Both BPE and WordPiece use subword information and work very similarly in creating the token vocabulary. Hence, we do not expect the choice between BPE and WordPiece tokenization to affect our results greatly. Still, to keep things consistent in our experiments, we also used WordPiece tokenization to build our vocabulary. For the pre-trained code models, the vocabulary was built from the pre-training data, while the non-pre-trained baseline models built their vocabulary from the training set of our fine-tuning data.

Like Husain et al. we converted all query input to lowercase and kept the case information of the source code input. The same is true for the pre-trained models: We used the uncased version of Devlin et al.'s English model and pre-trained our own source code models case-sensitively.

4.3.2 Baselines

First, we built a simple Elasticsearch baseline with default parameters. Elasticsearch is a widely used, open-source search engine and retrieves documents using an inverted index structure and term frequency–inverse document frequency (tf-idf) weighting between query and document. By default, Elasticsearch converts all text to lowercase and splits tokens based on grammar. The intention behind this baseline is to give an estimate of what is possible with a low-effort and low-cost, "out-of-the-box" solution and to assess the usefulness of the multimodal embedding model.

Second, we trained the multimodal embedding model *without* any pre-training. We used the same hyperparameters as in Table 4.9 to make our baseline comparable to the experiments with pre-trained models. This baseline allows us to measure the effect of transfer learning, i.e., how much better the pre-trained models perform compared to a model trained from scratch.

Table 4.10 shows our baseline experiments. Note that the Elasticsearch model does not require any training. It simply indexes all code snippets from the test set and retrieves them during evaluation.

pre-training E_q	pre-training E_c	training	evaluation	experiment label
none	none	none	JavaScript	`ES-{JS}`
none	none	none	Java	`ES-{JA}`
none	none	none	Python	`ES-{PY}`
none	none	none	TOP$_{SO}$	`ES-{TOP}`
none	none	JavaScript	JavaScript	`[no_no]-(JS)-{JS}`
none	none	Java	Java	`[no_no]-(JA)-{JA}`
none	none	Python	Python	`[no_no]-(PY)-{PY}`
none	none	TOP$_{SO}$	TOP$_{SO}$	`[no_no]-(TOP)-{TOP}`

Table 4.10: Experiments for the Elasticsearch and non-pre-trained baselines.

4.4 Evaluation

To test our model performance, we applied the same evaluation strategy as Husain et al. [5], i.e., for each query in our test set we search for the correct answer among 1 000 code snippets (the correct code snippet and 999 distractor snippets). The distractor snippets are selected randomly from our test set. While a search corpus of 1 000 code snippets is small, a fixed search corpus size makes our results comparable. As an evaluation measure we use MRR (see Section 3.1.1).

4.5 Threats to Validity

The following section discusses some factors that may limit the validity of our experimental design.

Internal validity. The biggest limitation to our experimental design comes from the nature and quality of our evaluation dataset. While using StackOverflow questions and code answers allows us to gather large amounts of evaluation data, we cannot be sure that they are a valid proxy for measuring code search performance. It is possible that we measure something else instead, such as how well our model can find the right answer among multiple possible answers to a StackOverflow question.

Furthermore, not all questions ask for a code answer to a concrete implementation problem. Some questions touch on more high-level, abstract topics, such as programming style or best practices. The answer to these kinds of questions may still contain code examples for demonstration purposes. In these cases, we observe a big semantic discrepancy between the query and the corresponding code snippet.

Related to this is the fact that code snippets alone might not give a comprehensive answer to the question posed and it only make sense in the context of the surrounding natural language explanations of the answer post. This is especially true because, for answers which contain several code snippets, we concatenate them into one, which makes the code snippets less cohesive.

Lastly, the code snippets can contain comments, which we did not remove during pre-processing. While we would want the comments to be included in the search results returned to the user, they may be considered noise to our code encoder, which was pre-trained on source code where comments were removed. The same is true for console outputs, which are not removed from the evaluation dataset.

External validity. Our results are limited in the way that they can be generalized to other source code analysis tasks. While problems, such as code summarization and code generation are very similar to code search, we did not evaluate those problem tasks experimentally. This limitation is especially true, because both those problems require *generative* models that produce an output sequence (a natural language sequence in code summarization and a source code sequence in code generation). The model we developed is only capable of finding code snippets from a corpus of *existing* snippets.

Chapter 5

Mining StackOverflow Data for Code Search Evaluation

While, with the CodeSearchNet dataset [5], we had a large enough dataset for pre-training, we needed a different dataset for fine-tuning and evaluation. We could have used the same dataset for all three phases (pre-training, fine-tuning, and evaluation) but not only would that have reduced the amount of data available for each phase, it would also not reflect a typical transfer learning scenario, in which the pre-training dataset *differs* from the fine-tuning dataset. For example, in machine translation, we do not have a lot of aligned data between Urdu and English but a lot of aligned data between French and English [28]. Thus, to build a translation model between Urdu and English, we use the large French-English corpus for pre-training and the small Urdu-English corpus for fine-tuning.

Furthermore, we believe that method-docstring data, of which the CodeSearchNet dataset consists, is not very well suited for simulating code search, because docstrings are very different from code search queries. Not only are they usually much longer than search queries, but they are also commonly formulated only *after* the code has been written. The latter is fundamentally different from a search query formulation, where, typically, the query is formulated without prior knowledge of what a relevant search result looks like.

For the reasons above, we decided to mine our own dataset from StackOverflow. We chose question titles as the natural language data and answers that contain code snippets as the source code data. We can thus ensure that the pre-training data is different from the fine-tuning data (in accordance with a typical transfer learning scenario) and that the natural language examples, i.e., question titles, resemble search queries that would be sent to a code search engine. For answers that contain more than one code snippet, we concatenated all code snippets into one (separated by a new line character). We remove all text that is not contained in the code snippet itself, i.e., text surrounding the code snippet.

To gather examples that are specific to a programming language, we filtered questions by “javascript”, “java”, and “python” tags. In order to gather more data, we included partial matches as well, which resulted in questions with tags such as “javascript-framework”, “javabeans”, or “python-3.6” to be included in our corpus. A full list of included tags can be found in Appendix A.

We selected only question-answer pairs whose answer was an accepted answer. Because only the question poster can mark an answer as accepted, we can assume that an accepted answer reflects the solution that the question poster was looking for. In other words, the question poster finds that answer relevant to his question. This is exactly the behavior we expect of a search engine: It should return results that are relevant to the user’s query. We could have selected the highest upvoted answer to build question-answer pairs. However, because every StackOverflow user can upvote an answer, we do not know anything about the relevance of that answer with regards to the poster's question intent. Thus, we believe that for a code search application, accepted answers are better than highest upvoted answers to build question-answer pairs.

Like Husain et al. [5], we filtered out code answers which have fewer than three lines of code as these are quite noisy. Many of them contain only library import statements or they contain code that is not written in the target programming language, such as SQL queries, regular expressions, or command line instructions (e.g., setting environment variables, how to start a Python program, how to deploy a Java application to a Tomcat server). To further increase the quality of our sample, we removed any question-answer pairs in which either the question or the answer received fewer than three upvotes. Overall, our data mining process included the following steps:

1. Filter StackOverflow questions by "javascript", "java", and "python" tags.
2. Remove questions that do not have an accepted answer.
3. Remove questions whose accepted answer does not contain a code snippet (using the `<pre><code>` tags). Concatenate several code snippets of the same answer into one. Discard text outside `<pre><code>` tags.
4. Remove question-answer pairs where either the question or the answer has fewer than three upvotes or where the answer contains fewer than three lines of code.

In the subsequent Section 5.1, we will discuss some quantitative measures of our dataset. Then, in Section 5.2, a brief qualitative analysis of the dataset will be presented.

5.1 Data Statistics

Filtering all StackOverflow questions by the "javascript", "java", and "python" tags resulted in about 2 million JavaScript-, 1.8 million Java-, and 1.8 million Python-related questions, of which roughly half had an accepted answer. After removing questions whose accepted answer did not contain at least one code snippet, questions that had fewer than 3 upvotes, whose accepted answer had fewer than 3 upvotes, and whose code in the accepted answer had fewer than 3 lines, we were left with 85 049 JavaScript, 71 194 Java, and 87 231 Python question-answer pairs. Table 5.1 lists the number of samples remaining after each filtering step.

	JavaScript	Java	Python
total questions	2 045 114	1 841 296	1 884 571
questions with accepted answer	1 105 690	934 062	984 989
accepted answer contains code snippet	861 273	533 217	655 430
3+ upvotes and 3+ lines of code	**85 049**	**71 194**	**87 231**

Table 5.1: Number of StackOverflow questions after each filtering step. Every row includes the filtering criteria of the preceding rows. Thus, the last row accounts for questions that have an accepted answer *and* whose answer contains a code snippet *and* that have three or more upvotes and three or more lines of code. The numbers in the last row represent our final dataset sizes.

When analyzing the effects of the last filtering step (removing question-answer pairs with fewer than three upvotes or fewer than three lines of code) in Table 5.2, we realize that, even though we only removed questions and answers with fewer than three upvotes, the average number of upvotes increased for each programming language by at least a factor of five for the questions and at least a factor of four for the accepted answers. Furthermore, while the average question length became slightly smaller, the average answer length became noticeably larger, both in number of tokens and number of lines.

		JavaScript	Java	Python
before filtering	avg. question upvotes	2.94	3.18	3.51
	avg. question length (tokens)	8.74	8.62	9.08
	avg. answer upvotes	4.75	5.14	5.34
	avg. answer length (tokens)	175.61	203.50	165.15
	avg. answer length (lines)	29.73	29.64	25.89
after filtering	avg. question upvotes	21.16	16.64	18.37
	avg. question length (tokens)	8.48	8.49	8.66
	avg. answer upvotes	28.96	22.61	24.13
	avg. answer length (tokens)	207.39	262.99	205.90
	avg. answer length (lines)	34.43	37.73	32.63

Table 5.2: Dataset statistics before and after filtering.

5.2 Qualitative Data Analysis

Because JavaScript is mostly used in web development, the JavaScript dataset contains extensive amounts of HTML and CSS code. We observe similar noise in Java, where several answers contain XML code (XML is a commonly used in Java programs for configuration). In Python, the most noticeable noise are code snippets of interactive sessions, which start with `>>> `. Not only are the `>>>` symbols different from pure code snippets, but often, these lines are followed by console output instead of source code.

Sometimes, the code answer contains comments, which further explain the answer. It should be noted that comments were explicitly removed from the pre-training data using a parser. Because the code snippets in StackOverflow answers are not necessarily syntactically correct, we cannot use a parser to remove comments from the answer snippets. We could simply exclude non-parsable answers from the dataset, which previous work has done. But not only would that reduce the size of our dataset, it is not necessary for our model to receive syntactically correct code because our model is purely token based. This is one advantage over models which make use of syntactic structure in the code such as abstract syntax trees (ASTs). Nevertheless, the fact that our pre-trained model has not seen comments will likely affect its performance during fine-tuning.

Chapter 6

Results

In this chapter, the results from the experiments described in Chapter 4 are presented. Section 6.1 introduces the results of all experiments involving a single-language pre-trained model. Then, the results of the pre-trained multi-language models are exhibited in Section 6.2. For each experiment, the significance of its results compared to its baselines was assessed by a two-tailed heteroscedastic *t*-test. A summary of all experiments together with their statistical evaluation can be found in Appendix B.

6.1 Pre-trained Single-language Models (RQ1–RQ3)

In this section, the results of the experiments involving models that were pre-trained on English natural language or on a single programming language are presented.

Pre-trained query encoder (RQ1). Figure 6.1 shows the evaluation results of the Elasticsearch baselines (light gray), the non-pre-trained multimodal embedding model (MEM) baselines (dark gray), and the model with the pre-trained query encoder (red). The black dots mark the mean reciprocal rank (MRR) values of each fold in the 3-fold cross-validation. It becomes clear that the Elasticsearch baseline performs better than the MEMs across all programming languages, reaching MRR scores of around 0.2. In the case of JavaScript and Java, the results of the pre-trained model are significantly lower than those of the Elasticsearch model. Compared to the non-pre-trained baseline, however, the pre-trained model shows a slight improvement on the JavaScript and Python dataset and a significant improvement on the Java dataset.

Pre-trained code encoder (RQ2). In Figure 6.2, the same Elasticsearch and non-pre-trained baselines are presented as before (gray), but this time they are compared to the models with the pre-trained code encoder (blue). We see that the pre-trained models reach similar MRR values as the Elasticsearch models, but they exhibit higher variance across the three folds. The pre-trained models outperform the non-pre-trained baselines on all datasets and do so significantly in the case of JavaScript and Java.

Pre-trained query and pre-trained code encoder (RQ3). As Figure 6.3 shows, when combining the pre-trained query encoder with the pre-trained code encoder (green), the MEM outperforms both the Elasticsearch and the non-pre-trained baselines. All MRR values of the pre-trained models are significantly higher than the values of the non-pre-trained baselines, and significantly higher than the values of the Elasticsearch baselines in the case of JavaScript and Java.

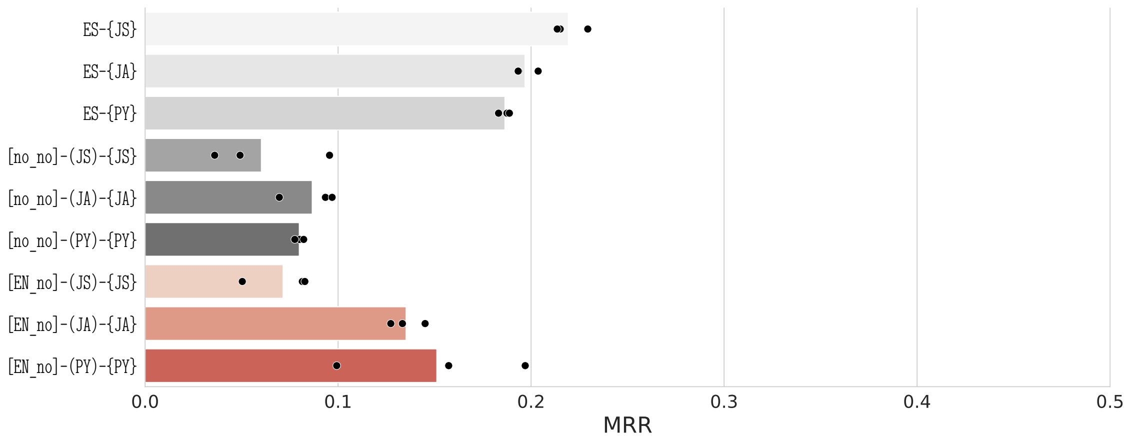

Figure 6.1: Evaluation of the pre-trained query encoder model.

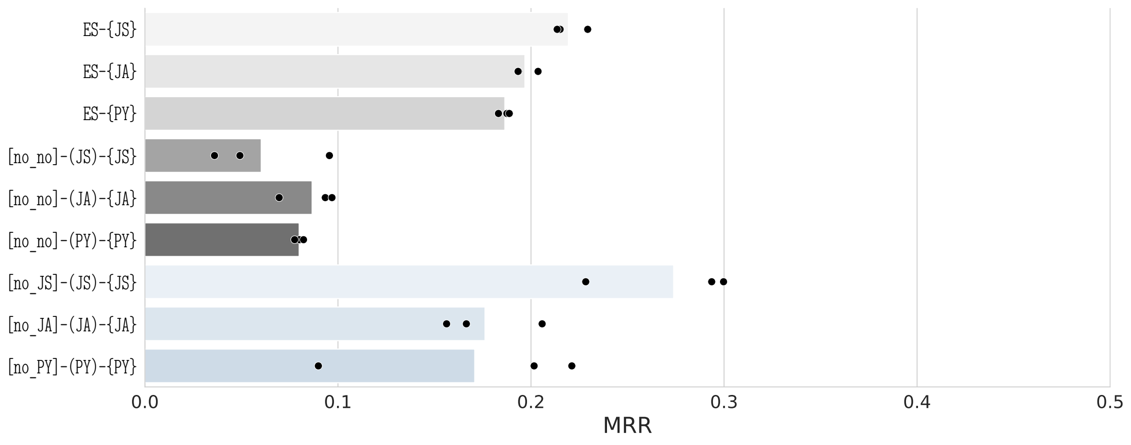

Figure 6.2: Evaluation of the pre-trained single-language code encoder models.

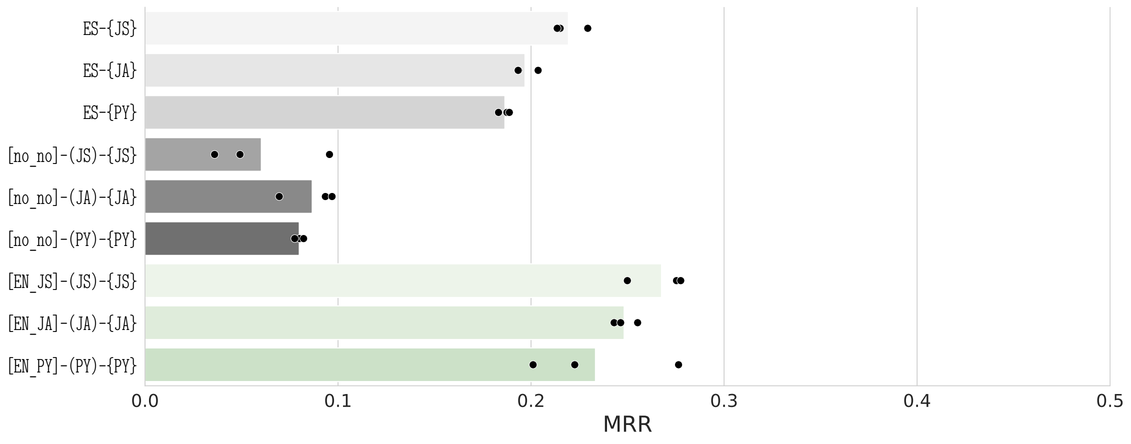

Figure 6.3: Evaluation of the pre-trained single-language query and code encoder models.

6.2 Pre-trained Multi-language Models (RQ4)

In this section, the results of the MEMs that were pre-trained on several programming languages, are presented. First, we will focus on the models that were pre-trained on the combination of JavaScript, Java, and Python data (TOP$_{GH}$ dataset). Then, we will show the results of the models that were pre-trained on the combination of JavaScript, Java, Python, PHP, Go, and Ruby data (ALL dataset).

While the single-language models from Section 6.1 were only evaluated on single-language corpora, the experiments on multi-language models were additionally tested on the multi-language corpus TOP$_{SO}$ consisting of StackOverflow question and answers for JavaScript, Java, and Python.

Pre-trained on TOP$_{GH}$ dataset. Figure 6.4 shows the results of the models that were pre-trained on the TOP$_{GH}$ dataset and evaluated on single-language corpora. We included the pre-trained query models [EN_no] again for comparison (their results were already shown in Section 6.1). We observe that all models in which the code encoder was pre-trained ([no_TOP], [EN_TOP]), show significant improvements over the non-pre-trained baselines. Compared to the Elasticsearch baselines, the combined pre-trained models [EN_TOP] reach slightly higher and — on Python data — significantly higher MRR values. With the exception of search on Java data, the models in which pre-training was applied only to the code encoder ([no_TOP]) achieve slightly higher MRR scores, albeit with higher variance and no statistical significance.

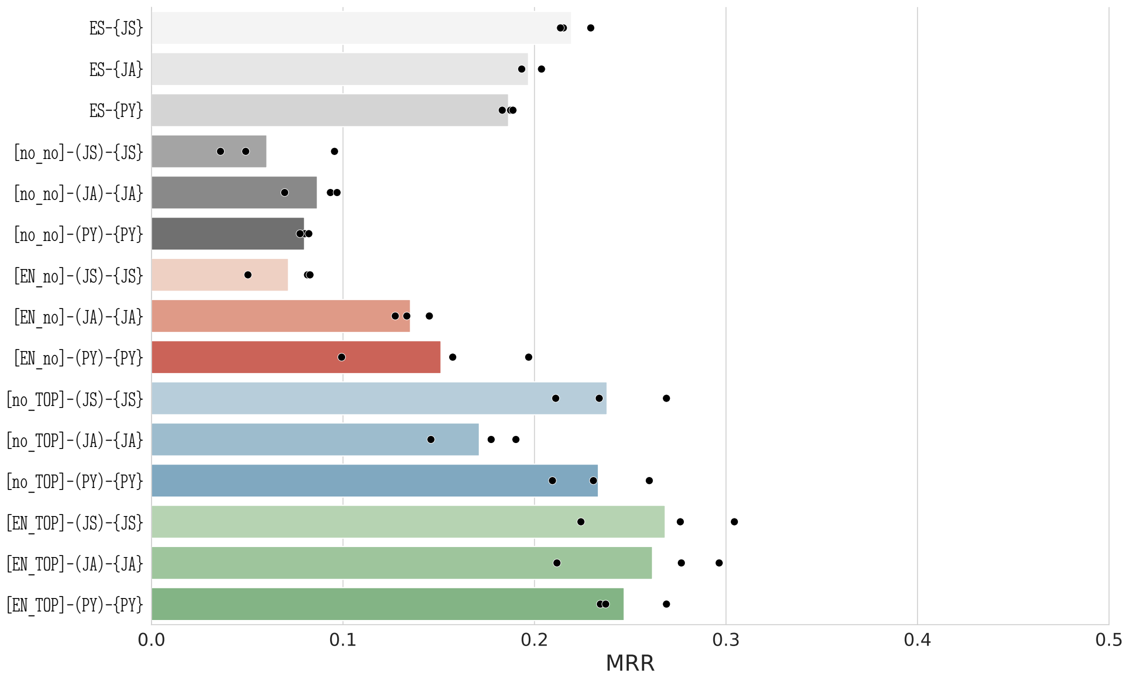

Figure 6.4: Single-language corpus evaluation of the code models pre-trained on the TOP$_{GH}$ dataset.

In multi-language search (Figure 6.5), it stands out that both the pre-trained and non-pre-trained MEMs reach MRR values above 0.3 and significantly outperform the Elasticsearch baseline. Compared to the non-pre-trained baseline [no_no]-(TOP)-{TOP}, pre-training improves multi-language search in all cases (query encoder pre-trained, code encoder pre-trained, both pre-trained), however, only the results of the model in which both encoders are pre-trained ([EN_TOP]) are significant. This model is also the only one that reaches an average MRR greater than 0.4.

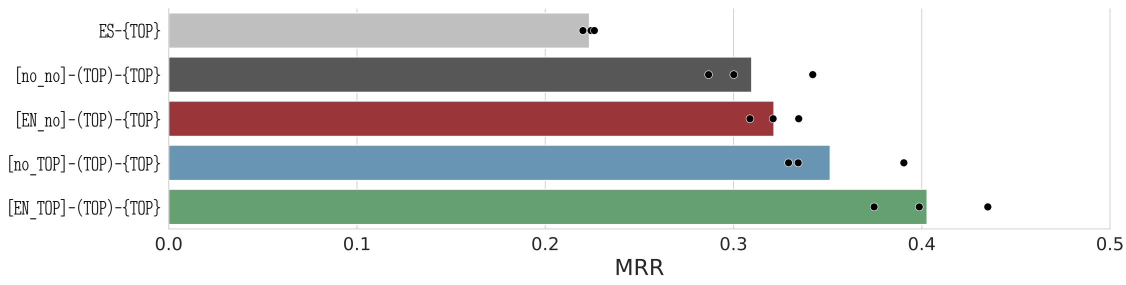

Figure 6.5: Multi-language corpus evaluation of the code models pre-trained on the Top_{GH} dataset.

Pre-trained on ALL dataset. Figure 6.6 summarizes the results of the models that were pre-trained on the All dataset and evaluated on single-language search corpora. Again, the combination of pre-training the query encoder and the code encoder yields the best results. All of these combined pre-trained models significantly outperform their non-pre-trained baselines. Yet, compared to the Elasticsearch baseline, they only show significant improvements in the JavaScript and Python case. Still significantly better than the Elasticsearch baseline is the [no_ALL] when it comes to search on Python data.

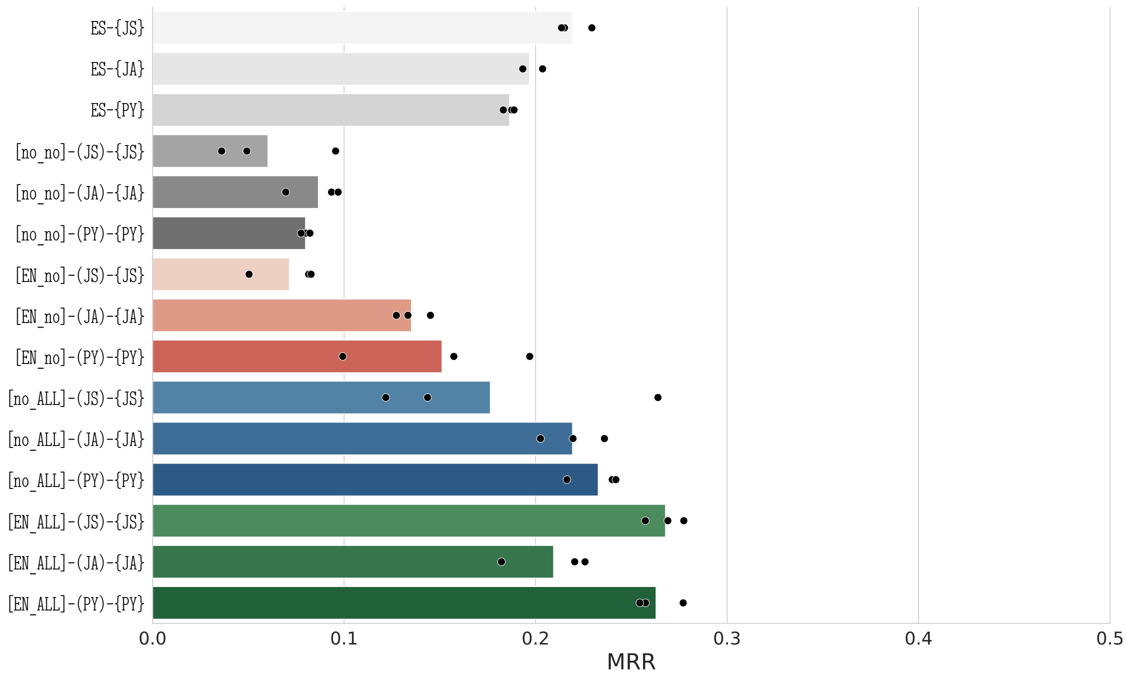

Figure 6.6: Single-language corpus evaluation of the code models pre-trained on the All dataset.

When looking at the results of multi-language search (Figure 6.7), we notice that the results are nearly identical to the results of the models pre-trained on Top_{GH} (Figure 6.5). Again, all MEMs achieve significantly higher MRR values than the Elasticsearch baseline, with the model that was pre-trained on both natural language and source code ([EN_ALL]) also significantly outperforming the non-pre-trained baseline [no_no].

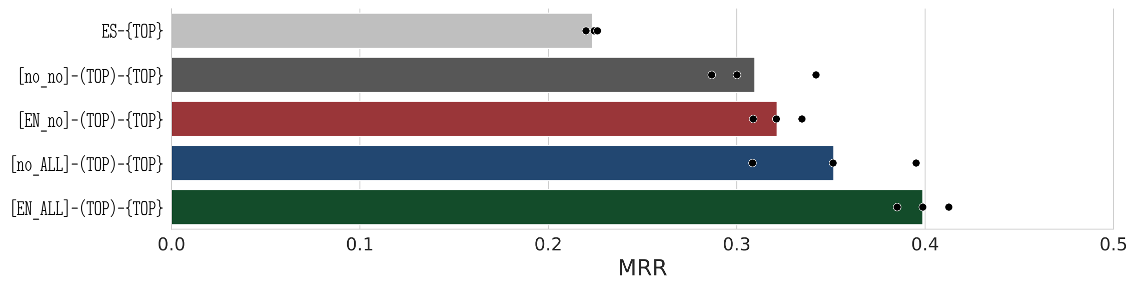

Figure 6.7: Multi-language corpus evaluation of the code models pre-trained on the All dataset.

Chapter 7

Discussion

In the following Chapter, we discuss the results of our code search experiments. The discussion is divided into two Sections: In Section 7.1, the quantitative results presented in Chapter 6 are analyzed. Then, in Section 7.2, the results from different models in our experiments are analyzed qualitatively on the basis of search result examples.

7.1 Quantitative Analysis

The effect of transfer learning. As has been demonstrated in Chapter 6, the pre-trained models outperfom their non-pre-trained baselines in all experiments, thus supporting the hypotheses posed by our research questions in Chapter 1. While models with only a pre-trained query encoder showed minor improvements over the non-pre-trained baselines (only one result was significant), the evidence is particularly strong in the case where a pre-trained code encoder was involved. In 7 out of 11 cases in which only the model's code encoder was pre-trained, the results significantly improved over the non-pretrained baselines. When both the query encoder and the code encoder were pre-trained, the resulting MRR was always significantly higher than the non-pre-trained baselines (11 out of 11 cases). The fact, that pre-training has a positive effect on model performance means that our pre-training tasks masked source code modeling ($\mathrm{MCM}_{\mathrm{PT}}$) and next line prediction ($\mathrm{NLP}_{\mathrm{PT}}$) are suitable pre-training tasks for source code and that a sequence-to-sequence model, such as BERT, can build effective source code models.

Pre-trained query encoder vs. pre-trained code encoder. Across all experiments, the effect of pre-training the query encoder on natural language is smaller than the effect of pre-training the code encoder on source code, even though the query encoder was pre-trained for much longer and on a much larger dataset than the code encoder. We attribute this difference to the fact that queries tend to be much shorter (around 9 tokens) than the code snippets (around 225 tokens, see Table 5.2). BERT and the pre-training tasks (masked language modeling and next line prediction) are designed to learn the relationships between tokens in a sequence. The shorter the sequence is, the less impactful the learned contextual embeddings from the pre-training become. By this logic, we expect the effect of using a pre-trained query encoder to become larger with longer query lengths. While this assumption was not tested in this thesis, it could be verified with our StackOverflow dataset by replacing the question title with the question body as the query sequence.

Training dataset size and performance. The pre-trained models converged at about the same rate as the non-pre-trained model. Across all experiments, the MEM reached its minimum validation loss after one epoch and the validation MRR values plateaued after three epochs or earlier.

This quick convergence is in part due to the small training datasets. A small training dataset means that the model runs out of a training signal quickly, i.e., the training loss reaches its minimum after a few iterations and does not decrease anymore. On one hand, quick model convergence is desirable as it reduces training time. On the other hand, it means that the model cannot improve its performance any further. Nevertheless, the pre-trained models outperformed the non-pre-trained models in all our experiments, i.e., they reached higher MRR values in the same training time. We conjecture that pre-training acts as a form of data augmentation: The pre-training dataset, despite being different from the fine-tuning dataset, enlarges the fine-tuning dataset, thereby extending the training signal and allowing the model to learn for longer. This claim is supported by the fact that the difference between the pre-trained and the non-pre-trained model is much smaller in the case where the models were fine-tuned on the larger multi-language datasets Top_{SO} and All (Section 6.2). The larger datasets mean that the training signal persists long enough for the model to achieve good performance, even if it was not pre-trained. We observe the same behavior in the work by Husain et al. [5]. When they train their BERT-based Self-Attention model on the large PHP, Java, and Python datasets, each consisting of more than 500 000 training samples, it outperforms the other models. However, when they train it on the smaller Ruby dataset with only 57 393 training samples, the Self-Attention model falls behind. We conclude that for problems in which enough training data is available, pre-training may not be necessary. Finally, *if* pre-training is applied, we assume that, the more similar the pre-training dataset is to the fine-tuning dataset and the more similar the pre-training tasks are to the downstream task, the better the performance will be during fine-tuning.

7.2 Qualitative Analysis

In the following section, we present some examples of queries and code snippets retrieved by our best performing model [EN_TOP] and compare them to the code snippets retrieved by the Elasticsearch baseline.

Listing 7.1 shows the correct JavaScript snippet for the query "Understanding Backbone.js event handler". Our pre-trained model retrieved the snippet at the first position, while the Elasticsearch model did not retrieve the snippet at all. This example demonstrates the problem of lexical gaps between query and code snippet and shows the limitations of classic token-matching models like Elasticsearch. If there is no overlap between the tokens in the query and the tokens in the code, Elasticsearch is not able to retrieve the code snippet.

Furthermore, the example validates the robustness of a purely token-based sequence model with regards to code syntax. We observe that this code snippet does not have valid JavaScript syntax as none of the opening braces `{` are matched by a closing brace `}`. Nevertheless, our model is able to build a semantic representation of the code snippet that leads to a successful retrieval. Examples like these are problematic for models that rely on syntactic features of the code snippet, such as abstract syntax trees (ASTs).

```
fullsize: function(ev) {
    target = $(ev.currentTarget);
fullsize: function(ev) {
    var target = ev.currentTarget;
    var self = this;
    $('.drop-shadow').click(function(inner_ev) {
          console.log(this.id); // the same as inner_ev.currentTarget
          console.log(self.cid); // the containing view's CID
```

Listing 7.1: Correct code snippet for query "Understanding Backbone.js event handler"

In the next example, we can observe that, even when the MEM does not return the correct code snippet, the retrieved snippet might still be relevant to the query. Listing 7.2 shows the correct snippet to the query "Is it possible to make POST request in Flask?", which the MEM returned at rank 3, whereas Listing 7.3 presents the code snippet which the MEM returned at rank 1. We notice, that both results describe an API endpoint responding to HTTP POST requests. Elasticsearch returned the correct snippet only at rank 58, despite the matching tokens "POST" and "request".

```
@app.route("/test", methods=["POST"])
def test():
    return _test(request.form["test"])
@app.route("/index")
def index():
    return _test("My Test Data")
def _test(argument):
    return "TEST: %s" % argument
```

Listing 7.2: Correct code snippet for query "Is it possible to make POST request in Flask?"

```
if __name__ == '__main__':
    app.debug = True
    app.run()
TypeError: 'dict' object is not callable
def api_response():
    from flask import jsonify
    if request.method == 'POST':
        return jsonify(**request.json)
```

Listing 7.3: Code snippet retrieved by the MEM at rank 1 for query "Is it possible to make POST request in Flask?"

When examining cases in which the MEM performs poorly, we mainly see examples in which the query is about something abstract, such as a programming concept, and less about a concrete implementation. Furthermore, our model struggles with queries that are very specific. Listing 7.4 shows the correct code snippet to such an abstract and very specific query ("When inserting objects into a type-safe heterogeneous container, why do we need the class reference?"). The MEM model retrieved this snippet at rank 165. In this particular example, the code snippet does not relate to the query at all and was presumably used by the author to get a point across with an example. We argue that this is more a limitation of our evaluation dataset than of our model as we expect queries and code snippets to be much more closely related in a real world application of code search.

```
Number num = new Integer(4);
System.out.println(num.getClass());
class java.lang.Integer
```

Listing 7.4: Correct code snippet for query "When inserting objects into a type-safe heterogeneous container, why do we need the class reference?"

Chapter 8

Conclusion and Future Work

We have demonstrated that transfer learning is an effective method for improving code search performance of neural networks. The impact of transfer learning is particularly noticeably in cases where limited training data is available. Because many problems dealing with the analysis of source code are limited by the size of the training dataset and because large code corpora can easily be obtained from open source platforms such as GitHub, we advocate that transfer learning can lead to improvements in other source code analysis tasks as well.

Furthermore, we have shown that state-of-the-art sequence-to-sequence models such as BERT that were originally designed for natural language processing (NLP) tasks can successfully be applied to problems dealing with source code data. However, due to the large number of parameters of such models, they require extensive amounts of pre-training and fine-tuning data. In cases where the training dataset is small, an Elasticsearch model achieves similar or better results in code search.

Moreover, we found some evidence that BERT, while being effective at modeling long sequences with hundreds of tokens, may be limited in modeling very short ones (fewer than 10 tokens). As search queries tend to be short, this might be a limiting factor of BERT when applied to code search.

To enable further research into code search and transfer learning for source code analysis, we publish our code search dataset, the pre-trained source code models, as well as the source code for data mining, pre-training, and fine-tuning.[1]

The main contributions of this thesis are:

1. Demonstrated that transfer learning improves code search performance.
2. Showed that sequence-to-sequence models such as BERT are effective at source code modeling if enough training data is used.
3. Pre-trained source code models on custom masked source code modeling and next line prediction tasks and made the pre-trained models available to the public.[1]
4. Mined a code search dataset, which consists of natural language questions and code answers and made it available to the public.[1]

Despite these findings, there are still open questions to address in the future. For one, our code encoder treats source code the same as natural language, namely as a sequence of tokens. While we have demonstrated that such a token-based model can yield good results on code search, we expect it to perform even better if the model makes use of the highly structured nature of source

[1]chrisly-bear.github.io/tlcs

code. This can be achieved, for example, by replacing or augmenting the token-based input to the code encoder with input features that represent the structural information of source code, such as ASTs or control flow graphs (CFGs).

Finally, it would be insightful to inspect BERT's attention heads when it processes source code. For natural language, it has been shown that the attention heads focus on specific language constructs, such as verbs and their objects or delimiter tokens [22]. In a similar fashion, it would be interesting and useful to understand to what source code tokens BERT attends during code search. Such understanding of the model's inner workings can drive the development of better model architectures for code search and other source code analysis tasks.

Bibliography

[1] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 5998–6008. [Online]. Available: http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[4] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*, 2018.

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

[6] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[7] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *International conference on machine learning*, 2015, pp. 2123–2132.

[8] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[9] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 404–415.

[10] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[11] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, pp. 31–41.

[12] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[13] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2019, pp. 13–25.

[14] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[15] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and localizing change requests for mobile apps based on user reviews," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE, 2017, pp. 106–117.

[16] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval.* Cambridge university press, 2008.

[17] W. Guo, J. Wang, and S. Wang, "Deep multimodal representation learning: A survey," *IEEE Access*, vol. 7, pp. 63 373–63 394, 2019.

[18] T. Baltrušaitis, C. Ahuja, and L.-P. Morency, "Multimodal machine learning: A survey and taxonomy," *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 2, pp. 423–443, 2018.

[19] Y. Hussain, Z. Huang, Y. Zhou, and S. Wang, "Deep transfer learning for source code modeling," *arXiv preprint arXiv:1910.05493*, 2019.

[20] R.-M. Karampatsis and C. Sutton, "Maybe deep neural networks are the best choice for modeling source code," *arXiv preprint arXiv:1903.05734*, 2019.

[21] Y. Hussain, Z. Huang, S. Wang, and Y. Zhou, "Codegru: Context-aware deep learning with gated recurrent unit for source code modeling," *arXiv preprint arXiv:1903.00884*, 2019.

[22] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning, "What does BERT look at? an analysis of BERT's attention," in *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP.* Florence, Italy: Association for Computational Linguistics, Aug. 2019, pp. 276–286. [Online]. Available: https://www.aclweb.org/anthology/W19-4828

[23] M. X. Chen, B. N. Lee, G. Bansal, Y. Cao, S. Zhang, J. Lu, J. Tsay, Y. Wang, A. M. Dai, Z. Chen *et al.*, "Gmail smart compose: Real-time assisted writing," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2287–2295.

[24] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).* Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://www.aclweb.org/anthology/D14-1162

[25] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016. [Online]. Available: http://arxiv.org/abs/1609.08144

[26] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large batch optimization for deep learning: Training bert in 76 minutes," *arXiv preprint arXiv:1904.00962*, 2019.

[27] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. [Online]. Available: https://www.aclweb.org/anthology/P16-1162

[28] B. Zoph, D. Yuret, J. May, and K. Knight, "Transfer learning for low-resource neural machine translation," *arXiv preprint arXiv:1604.02201*, 2016.

Appendix A

Included StackOverflow Tags

The following appendix contains the tags of all StackOverflow questions that we considered in our data mining process. The number of occurrences of each tag is denoted to the left of the tag and the tags are ordered by that number. Please note, that the total number of samples in our final dataset is smaller than the sum of all occurrences here because we applied some additional filtering. See Chapter 5 for the specifics.

A.1 JavaScript Tags

1 090 854	javascript	20	clojurescript-javascript-interop
6 621	javascript-events	20	javascript-security
2 977	javascript-objects	18	javascript-globalize
2 061	facebook-javascript-sdk	16	javascript-api-for-office
628	javascript-framework	13	javascript-inheritance
347	unobtrusive-javascript	11	adobe-javascript
324	youtube-javascript-api	9	javascript-marked
269	javascriptserializer	9	esri-javascript-api
209	serverside-javascript	9	ews-javascript-api
134	javascript-debugger	9	javascript-audio-api
134	isomorphic-javascript	8	amazon-javascript-sdk
130	javascriptcore	6	javascript-scope
108	rethinkdb-javascript	5	javascript-oscillator
104	javascriptmvc	5	javascript-1.7
89	javascript-automation	5	javascript-decorators
88	javascript-namespaces	5	javascript.net
77	javascript-injection	4	android-webview-javascript
76	javascript-intellisense	4	external-javascript-library
69	javascript-engine	4	shopify-javascript-buy-sdk
36	javascript-databinding	4	javascript-1.8
30	javascript-import	3	javascript-build
29	log4javascript	3	javascript-interop
26	asynchronous-javascript	2	javascriptservices
25	parse-javascript-sdk	1	flash-javascript-api
24	google-javascript-api	1	braintree-javascript
23	embedded-javascript	1	evaluatejavascript

1	neo4j-javascript
1	javascriptools
1	telerik-javascript

A.2 Java Tags

848 945	java
15 768	javafx
12 104	java-8
5 900	java-stream
4 311	java-native-interface
3 457	rx-java
3 149	java-me
3 024	javafx-8
2 953	java.util.scanner
2 800	javafx-2
1 978	javamail
1 914	rx-java2
1 882	javabeans
1 569	java-7
1 490	javadoc
1 236	javac
1 140	java-ee-6
851	java-io
777	java-9
746	java-web-start
730	java-time
687	java.util.concurrent
561	java-2d
543	java-11
531	mongodb-java
506	javax.imageio
505	javacard
496	java-ee-7
438	google-api-java-client
403	javasound
401	java-6
377	java-bytecode-asm
357	datastax-java-driver
345	javacv
339	java-threads
318	java.util.logging
303	spring-java-config
303	r.java-file
261	javacc
259	javassist
237	wsdl2java
229	cucumber-java
227	java-3d
223	scala-java-interop
220	java-module
215	spark-java
212	java-10
209	java-home
201	java-memory-model
195	javadb
192	grpc-java
185	java-api
173	rjava
167	clojure-java-interop
159	javaagents
156	java.util.date
155	mongo-java
152	aws-java-sdk
151	java-websocket
150	javafxports
131	graphql-java
130	effective-java
125	java-metro-framework
124	java-ee-5
119	javax.mail
116	java-5
115	java-security
115	java-date
109	elasticsearch-java-api
104	mongo-java-driver
102	javafx-3d
100	jira-rest-java-api
96	java-server
89	javafx-webengine
88	javafx-11
80	java-compiler-api
79	maven-javadoc-plugin
74	java-annotations
72	java-canvas
71	drjava
71	java-stored-procedures
70	java.nio.file
68	java.util.calendar
68	java.time
67	azure-java-sdk
64	javap

64	java1.4
62	java-ee-8
60	javax
59	javaparser
56	ewsjavaapi
54	javafx-css
52	couchbase-java-api
52	php-java-bridge
51	javah
51	javapoet
51	java-12
50	google-oauth-java-client
48	java-service-wrapper
47	java.lang.class
46	protobuf-java
46	javalite
46	java.library.path
45	acm-java-libraries
45	java.time.instant
42	java-client
42	javaw
40	java-http-client
39	javacpp
39	javacompiler
38	java-14
37	javax.crypto
37	java-wireless-toolkit
36	javax.sound.sampled
36	java-13
36	java-batch
35	java-opts
35	java-melody
35	javax.comm
34	javapos
34	realm-java
33	java-ws
32	javax.sound.midi
31	facebook-java-api
31	javapns
31	javax.xml
31	java-interop
31	java-package
30	javahelp
30	javahl
28	aws-sdk-java-2.0
27	functional-java
27	im4java
25	neo4j-java-api
25	java-calendar
24	javax.ws.rs
24	javapackager
23	javax.persistence
22	java-mission-control
22	webdrivermanager-java
21	luajava
21	javax.validation
21	javax.script
21	java-collections-api
21	handlebars.java
20	java.lang
19	web3-java
19	aws-java-sdk-2.x
19	java-money
19	javafx-9
17	java-binding
17	openid4java
17	javax.swing.timer
17	javafx-tableview
17	gcloud-java
17	jruby-java-interop
17	usb4java
17	javaplot
16	hbm2java
15	stringbyevaluatingjavascr
15	pljava
15	docker-java
14	java-record
13	biojava
13	pact-java
13	java-access-bridge
13	javafx-1
13	socket.io-java-client
13	java-synthetic-methods
12	javax.json
12	oci-java-sdk
12	javahg
12	javax.activation
12	java-bridge-method
12	p4java
12	javaspaces
12	skype4java
11	java-print
11	javabuilders
11	javax.swing.text
11	javafx-bindings
11	dnsjava
10	javaapns
10	javaloader
10	java-pair-rdd
10	java2wsdl

10	asterisk-java
10	java-security-manager
10	mongodb-java-3.3.0
10	smartsheet-java-sdk-v2
10	javalin
9	javafx-gradle-plugin
9	real-time-java
9	deployjava
9	prometheus-java
9	java-runtime-compiler
9	facebook-java-sdk
9	matlab-java
8	javaquery
8	java-custom-serialization
8	java-memory-leaks
8	jslint4java
8	loadjava
7	java-micro-editon-sdk3.0
7	java-war
7	java-persistence-api
7	gdata-java-client
7	hyperledger-fabric-sdk-java
6	node-java
6	javapolicy
6	raml-java-parser
6	kotlin-java-interop
5	java-gstreamer
5	java-flow
5	java-native-library
5	java.util.random
5	rx-javafx
5	smartsheet-java-sdk-v1
5	java-deployment-toolkit
5	javafx-webview
4	javaexec-gradle-plugin
4	java-scripting-engine
4	javarosa
4	javarebel
4	java2word
4	javafx-datepicker
4	google-java-format
3	java-assist
3	java-communication-api
3	unirest-java
3	rocksdb-java
3	underscore-java
3	java-console
3	javacameraview
3	java-heap
3	rx-java3
2	javax-inject
2	sqlite4java
2	java-aot
2	javax.mail.address
2	java-failsafe
2	java-transaction-service
2	javaoptions
2	jinjava
2	java-audio
2	java-attach-api
2	javaexe
2	javafx-packager
2	sql2java
2	javasymbolsolver
2	java-text-blocks
2	mdnsjava
2	java-ee-mvc
2	java.security
1	ballerina-java-interop
1	arangodb-java
1	dropbox-java
1	oracle-java-cloud-service
1	netlib-java
1	kusto-java-sdk
1	rsocket-java
1	javax.inject
1	java-process-runtime
1	azure-java-tools
1	aws-sdk-java
1	javac-compiler-plugin
1	java-test-fixtures
1	mongodb-java-3.8
1	aws-java-sdk-dynamodb
1	portal-java
1	javax.annotation
1	pdf-java-toolkit
1	java-resources

A.3 Python Tags

767 781	python
112 319	python-3.x
49 107	python-2.7
5 814	python-requests
3 941	wxpython
3 020	ipython
2 967	python-imaging-library
2 340	python-3.6
2 027	python-import
1 808	python-asyncio
1 768	python-3.5
1 588	python-2.x
1 389	python-3.4
1 281	python-3.7
1 250	python-multiprocessing
1 247	ironpython
1 234	python-sphinx
1 224	mysql-python
1 205	python-multithreading
1 108	python-unittest
1 057	python-decorators
1 021	ipython-notebook
772	python-2.6
708	python-module
671	boost-python
650	python-3.3
605	python-datetime
594	python-idle
554	cpython
518	python-c-api
493	python-internals
488	biopython
479	python-unicode
416	python-xarray
373	google-api-python-client
348	python-docx
346	pythonanywhere
298	pythonpath
263	google-app-engine-python
241	python-telegram-bot
228	python-2.5
220	python-social-auth
219	python-tesseract
213	python-click
206	python-mock
204	python-3.2
204	python-pptx
201	python-packaging
195	python-wheel
192	python-dateutil
185	python-3.8
174	python-venv
170	gitpython
164	mod-python
150	python-c-extension
146	python-2.4
140	graphene-python
140	python-os
138	mysql-connector-python
132	python.net
131	python-behave
126	python-dataclasses
125	python-embedding
123	python-extensions
119	plotly-python
111	micropython
103	mechanize-python
100	python-sockets
98	python-typing
90	wxpython-phoenix
90	kafka-python
88	python-curses
87	python-2to3
86	python-regex
86	python-ggplot
84	opencv-python
81	vpython
80	python-cffi
79	python-importlib
77	python-webbrowser
76	python-mode
75	ipython-magic
71	rethinkdb-python
71	python-logging
69	google-cloud-python
69	ipython-parallel
68	python-ldap
68	python-hypothesis
68	plpython
67	python-jira
64	azure-sdk-python
63	python-twitter
62	python-db-api
58	python-interactive

58	python-gstreamer	23	pythoninterpreter
58	python-collections	23	python-nonlocal
57	python-rq	22	python-magic
56	python-elixir	22	python-iris
56	qpython	22	python-2.3
55	python-datamodel	22	pythonbrew
54	python-textprocessing	21	python-sounddevice
53	python-stackless	21	pythonista
53	python-watchdog	21	cefpython
51	python-fu	21	python-django-storages
51	python-descriptors	20	gdata-python-client
51	python-sip	19	python-jedi
50	python-cryptography	19	python-applymap
49	python-appium	19	qpython3
48	revitpythonshell	19	python-mss
47	gremlinpython	19	python-exec
45	python-unittest.mock	17	python-class
45	python-poetry	17	cocos2d-python
45	python-requests-html	17	python-bytearray
44	python-daemon	17	python-keyring
38	couchdb-python	16	python-chess
38	google-python-api	16	python-cmd
38	python-attrs	16	twitterapi-python
37	python-pika	16	python-huey
36	influxdb-python	15	pythoncard
36	p4python	15	python-bigquery
36	dnspython	15	python-standalone
36	python-imageio	14	arrow-python
36	rpython	14	python-dedupe
34	pythonxy	14	kubernetes-python-client
33	python-trio	13	ironpython-studio
33	python-coverage	13	vscode-python
32	python-turtle	13	cassandra-python-driver
32	parallel-python	13	python-performance
32	python-object	13	python-dragonfly
32	gdb-python	13	pythonmagick
32	grpc-python	12	python-camelot
31	pythoncom	12	python-jsons
28	python-memcached	12	gae-python27
27	python-install	11	python-cloudant
26	activepython	11	objectlistview-python
26	gcloud-python	11	python-redmine
26	dronekit-python	11	python-textfsm
26	python-bindings	10	pythonnet
25	python-newspaper	10	six-python
24	python-socketio	10	python-black
24	python-jsonschema	10	python-moderngl
23	python-pdfkit	9	python-beautifultable
23	pythonw	9	python-markdown
23	bpython	9	python-s3fs

9	python-server-pages
9	python4delphi
9	zen-of-python
9	python-3.1
9	python-sql
8	pvpython
8	rubypython
8	python-assignment-expression
8	python-2.2
8	epd-python
8	python-gearman
7	python-pool
7	python-nose
7	portable-python
7	python-envoy
7	python-netifaces
7	mne-python
7	python-sacred
6	python-aiofiles
6	pythonqt
6	python.el
6	python-responses
6	python-egg-cache
6	python-crfsuite
6	python-siphon
6	python-vlc
6	python-arrow
6	python-pbr
6	facebook-python-business-sdk
5	python-nvd3
5	python-pulsar
5	python-config
5	python-can
5	python-module-unicodedata
5	datastax-python-driver
5	python-zappa
5	python-openid
5	python-manylinux
4	python-language-server
4	python-zip
4	neo4j-python-driver
4	snap-python
4	nxt-python
4	oci-python-sdk
4	dbus-python
4	python-jose
4	intel-python
4	python-2.1
4	pythonplotter
4	python-tenacity
4	dlib-python
3	pp-python-parallel
3	re-python
3	python-fire
3	pact-python
3	python-routes
3	python-holidays
3	python-mro
3	python-pdfreader
3	adafruit-circuitpython
3	python-contextvars
2	spectral-python
2	graphql-python
2	postgres-plpython
2	python-records
2	datasift-python
2	telepathy-python
2	lifetimes-python
2	python-docker
2	openstack-python-api
2	python-visual
2	python-hdfs
2	python-antigravity
2	python-winshell
2	python-gitlab
2	python-paste
2	protobuf-python
2	python-decimal
2	python-green
2	python-onvif
2	python-pattern
2	python-requests-toolbelt
2	python-parallel
2	python-inject
1	python-control
1	intellij-python
1	keyboard-python
1	python-mockito
1	python-plyplus
1	needle-python
1	python-cachetools
1	python-billiard
1	python-blessings
1	pythonce
1	python-parsley
1	python-py
1	python-iptables
1	python-3.9
1	python-hunter
1	python-scoop

1	python-templates
1	python-windows-bundle
1	python-arango
1	python-igraph
1	python-simple-crypt
1	python-iso8601
1	python-pex
1	python-novaclient
1	confluent-kafka-python
1	python-shove
1	cf-python-client
1	hydra-python
1	python-nolearn
1	python-schematics
1	empythoned
1	python-architect
1	pybricks-micropython
1	python-bob
1	analytics-engine-python-sdk

Appendix B

Results Overview

This appendix provides an overview of the results of all experiments described in Chapter 4. Table B.1 contains the average MRR for each experiment across its three folds. Experiments which achieved significantly higher or significantly lower MRR scores than one of the baselines are marked. The significance was determined by a two-tailed heteroscedastic *t*-test between each pre-training experiment and its corresponding baselines. For instance, the results from experiment [EN_TOP]-(JS)-{JS} were compared to the results from the Elasticsearch baseline ES-{JS} and the non-pre-trained baseline [no_no]-(JS)-{JS}. Figure B.1 shows a visual representation of the results.

experiment	MRR	experiment	MRR
ES-{JS}	0.219★	[no_TOP]-(PY)-{PY}	0.233★
ES-{JA}	0.197★	[no_TOP]-(TOP)-{TOP}	0.351◆
ES-{PY}	0.186★	[no_ALL]-(JS)-{JS}	0.176
ES-{TOP}	0.223☆	[no_ALL]-(JA)-{JA}	0.219★
[no_no]-(JS)-{JS}	0.060◇	[no_ALL]-(PY)-{PY}	0.233◆★
[no_no]-(JA)-{JA}	0.087◇	[no_ALL]-(TOP)-{TOP}	0.352◆
[no_no]-(PY)-{PY}	0.080◇	[EN_JS]-(JS)-{JS}	0.268◆★
[no_no]-(TOP)-{TOP}	0.310◆	[EN_JA]-(JA)-{JA}	0.248◆★
[EN_no]-(JS)-{JS}	0.072◇	[EN_PY]-(PY)-{PY}	0.233★
[EN_no]-(JA)-{JA}	0.135◇★	[EN_TOP]-(JS)-{JS}	0.268★
[EN_no]-(PY)-{PY}	0.151	[EN_TOP]-(JA)-{JA}	0.262★
[EN_no]-(TOP)-{TOP}	0.321◆	[EN_TOP]-(PY)-{PY}	0.247◆★
[no_JS]-(JS)-{JS}	0.274★	[EN_TOP]-(TOP)-{TOP}	0.403◆★
[no_JA]-(JA)-{JA}	0.176★	[EN_ALL]-(JS)-{JS}	0.268◆★
[no_PY]-(PY)-{PY}	0.171	[EN_ALL]-(JA)-{JA}	0.209★
[no_TOP]-(JS)-{JS}	0.238★	[EN_ALL]-(PY)-{PY}	0.263◆★
[no_TOP]-(JA)-{JA}	0.171★	[EN_ALL]-(TOP)-{TOP}	0.399◆★

◇ significantly lower compared to the Elasticsearch baseline ($p < 0.05$)

◆ significantly higher compared to the Elasticsearch baseline ($p < 0.05$)

☆ significantly lower compared to the non-pre-trained baseline ($p < 0.05$)

★ significantly higher compared to the non-pre-trained baseline ($p < 0.05$)

Table B.1: Statistical evaluation of all experiments.

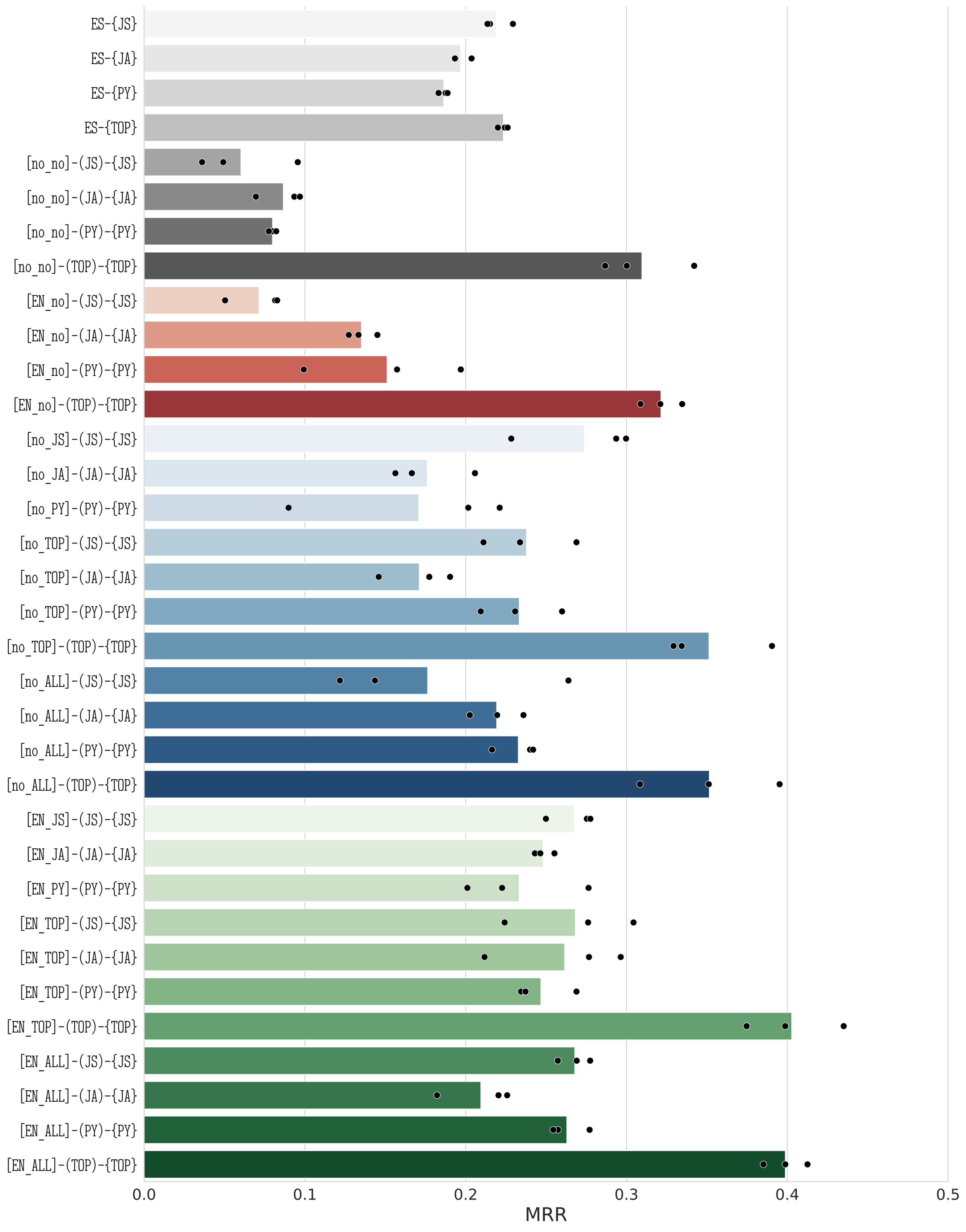

Figure B.1: Evaluation results of all experiments. The black dots represent the MRR of each individual fold, while the length of the bar equals the average MRR across all folds.