technische universität
dortmund

Master Thesis

Enriching Scientific Source Code by means of Mining Relations between Papers and their Implementation

Pierre Haritz

April 2021

Reviewers:
Prof. Dr. Katharina Morik
Dr.-Ing. Helena Kotthaus

Technical University Dortmund
Faculty of Computer Science
Chair of Artificial Intelligence
http://ls8-www.cs.tu-dortmund.de

"Words—so innocent and powerless as they are, as standing in a dictionary, how potent for good and evil they become in the hands of one who knows how to combine them."

Nathaniel Hawthorne (1804-1864)

Contents

Chapter 1

Introduction

This introductary chapter will give an overview of the motivation that inspired this thesis, as well as a plan what can be done to tackle some of the problems.

1.1 Motivation and Background

With the amount of research papers published daily getting exponentially bigger over recent years[1], the need for tools that help navigating through the ever-growing ocean of scientific articles becomes a necessity. Finding the right articles, as well as minimizing reading time spent on each paper, are key aspects of today's natural language processing research. As a result, several search engines, such as Semantic Scholar and summary tools have already come to light in recent years. Over the last five years, submissions for popular machine learning conferences, such as ICML[2] and NeurIPS[3] have increased by over 300%, while acceptance rate stayed at about 25% for both, albeit slowly decreasing recently. Reviewers have little time to decide whether a paper is accepted, and in the field of machine learning, where implementations of the presented algorithms often accompany those papers, the effort of verifying the implementations and results is at times not possible. At the same time, researchers, that want to apply those algorithms, do not want to spend hours or days working through the paper just to see what the parameters stand for, which is often not well documented in the code itself. Therefore, a tool that automatically finds structures and relations between papers and their implementations and subsequently is able to enrich the code in a way that helpful information from within the paper are present as comments, could be a massive improvement over current standards.

[1]https://arxiv.org/stats/monthly_submissions
[2]https://www.openresearch.org/wiki/ICML
[3]https://www.openresearch.org/wiki/NIPS

1.2 Contribution of this Thesis

To outline and define the problem, this thesis will try to answer several questions:

1. Which relevant tools are available to the research community at the moment?
2. In which ways can existing tools provide help?
3. How would the pipeline of a new prototypical tool look like?
4. What kind of data is needed for a task like this?
5. Is the data, that is available, suitable for such a task?
6. Is it possible to create a prototype, based on available data, to tackle the problem?
7. Is the task of aligning paper and code knowledge learnable by a simple network?
8. Can the proposed prototype already generate helpful comments?

1.3 Thesis Structure

After having explained the motivation and stated the resulting open questions, this thesis will first give an introduction to the relevant fields of *Natural Language Processing*, *Code Search*, *Encoders*, *Embeddings* and *Transformer-based Neural Network Architectures*, before introducing related works, tools, in both *Semantic Text Analysis* and *Semantic Code Analysis*. Furthermore, datasets, including the requirements suited data should fulfill and how data can be transformed to fit the needs will be explored and an assessment of using translation approaches to the given task is given. The next chapter will then propose a pipeline for such a prototypical tool and discuss its implementation step-by-step. Results will be evaluated in the following chapter, before the last chapter will finally give a conclusion and an outlook on what future research can look like.

Chapter 2

Foundations

This section will give an introduction to the theoretical foundations and the background needed for this thesis. First, a general overview of Natural Language Processing will be given. Then, the concept of Code Search will be explained. After that, a detailled description of Embeddings will be made before finally discussing Transformers and their architecture.

2.1 Natural Language Processing

In general, natural language processing refers to a subfield of machine learning that deals with making human language accessible for computers. Since both syntax and semantics play a big role in languages, the task is a most complex one. For example, having a high level of abstraction of language rules like the use of sarcasm as well as low level rules like appending "-ing" to a verb to transform it into a gerund in the English language, are equally part of it.

2.1.1 A Brief History of Natural Language Processing

The field of natural language processing evolved from a field called Machine Translation, at a time where Artificial Intelligence research did not exist yet [25]. Early ideas consisted of mechanical dictionaries, in as early as in the 17th century, that would serve as a multilingual translation tool. Since the technology was far from available, it took about 300 years for the issue to be adressed again. Warren Weaver, after developing the idea of using the recently invented computers for such a translation machine together with Andrew Booth, was the first to in theory tackle problems like ambiguity with the help of statistics in 1949. The fifties consisted of several efforts, mostly by MIT and some IBM, advancing the field, producing multiple books like [36].
It wasn't until the eighties that the approach for translation went from translating directly to the use of *text corpora*, meaning the view of language within a large body of naturally

occurring language, bringing the element of *context* into it.
With the development of the Internet in the mid 1990s, the popularity of MT surged again. The need for tools that translate web pages and emails grew suddenly, research interest peaked and brought us to where we are today. Companies started specializing in the processing of natural language and few grew to be some of the biggest players in today's economy.
Nowadays, NLP plays a huge role in everyday life. Not being limited to translation, applications such as spam detection for filtering emails, speech assistants like Apple's Siri and Amazon's Alexa, having phones auto-correcting user input, search engines like Google predicting the questions one wants to ask or even translating complex texts, have come to life. While these applications build upon different methods, it can be said that almost all build upon Machine Learning principles. The most prominent will hence be explored in the following section.

2.1.2 An Overview of NLP Tasks

A variety of tasks can be described within Natural Language Processing. Generally, they can each be classified as methods that analyze either language syntax or language semantics.

Text and speech processing

Processing raw data, such as text and speech signals, is a requirement for most other techniques of NLP. Text processing includes optical character recognition from images (e.g. by scanning book pages) and tokenization of sentences into words. In simple cases, this can be as easy as splitting words by spaces like in most languages that use Roman characters, or as hard as Japanese, where words are made up of sequences of syllables that are not separated from each other. Tokenizing latter thus requires an in-depth understanding of the language itself.

Morphological analysis

Naturally, grammar is hugely important to language structure. Words can be classified into *parts of speech* such as nouns and verbs. Since words can have multiple roles, mapping require additional information beyond the word itself. As an example, the word "fly" can be a noun, a verb "to fly" or even an adjective ("to be fly"). Hence, the task of *Part-of-Speech-Tagging*, assigning tags or labels to words in a sentence, also requires complex language models.

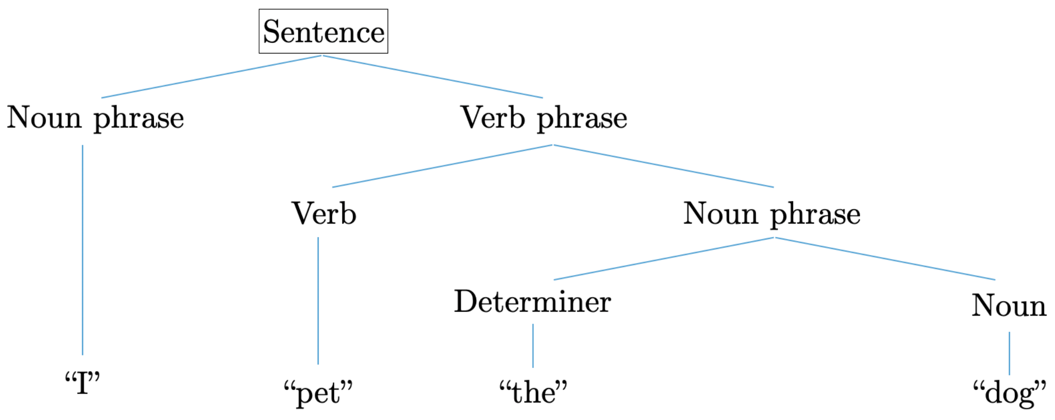


Figure 2.1: A constituency-based[2]parse tree for the sentence "I pet the dog".

Syntactic analysis

Language syntax is an important part of both natural languages and programming languages. While grammars for natural languages are ambiguous, grammars for programming languages can be either ambiguous or unambiguous. Briefly expanding beyond the bounds of natural languages, programming languages such as Python restrict their grammar to be $LL(1)$ [1] to avoid ambiguity, whereas in others, ambiguity can be induced by the Dangling-Else-Problem [1]. A LL grammar is a grammar that can be parsed by a LL parser, left-to-right, top-down.
For a terminal alphabet Σ, subset Σ^* is regular if it is a regular language over Σ. A partition π of Σ^* is regular if for all $R \in \pi$ the set R is regular. Then, a grammar G is LL-regular if a partition π of Σ^* exists, such that G is $LL(\pi)$. Furthermore, every $LL(k)$ grammar is also LL-regular and with all LL-regular grammars being unambiguous [45], restricting a programming language to be $LL(1)$ makes it retain the same property.
Parsing is a way of transforming a program or a sentence into a parse tree, an example of which can be seen in figure 2.1. Through *Dependency Parsing* the relationship of words in a sentence can be extracted. For the English language, this means having a sentence structure of subject-verb-object (SVO).
For programming languages, parsing code results in an Abstract Syntax Tree (AST). Such a tree is a representation of the source code's structure. To give a simple example, we will take a look at a simple algorithms and then its parsed tree in 2.2. ASTs can be edited and its nodes can be annotated or enhanced with additional information. One of its main uses is in compilation.

[1]https://www.python.org/dev/peps/pep-3099/
[2]https://web.stanford.edu/~jurafsky/slp3/13.pdf
[3]https://en.wikipedia.org/wiki/Abstract_syntax_tree

```
Require: $a, b$
  while $b \neq 0$ do
    if $a > b$ then
      $a := a - b$
    else
      $b := b - a$
    end if
  end while
  return $a$
```

Algorithm 2.1: Euclidean algorithm for computing the greatest common divisor of two numbers.

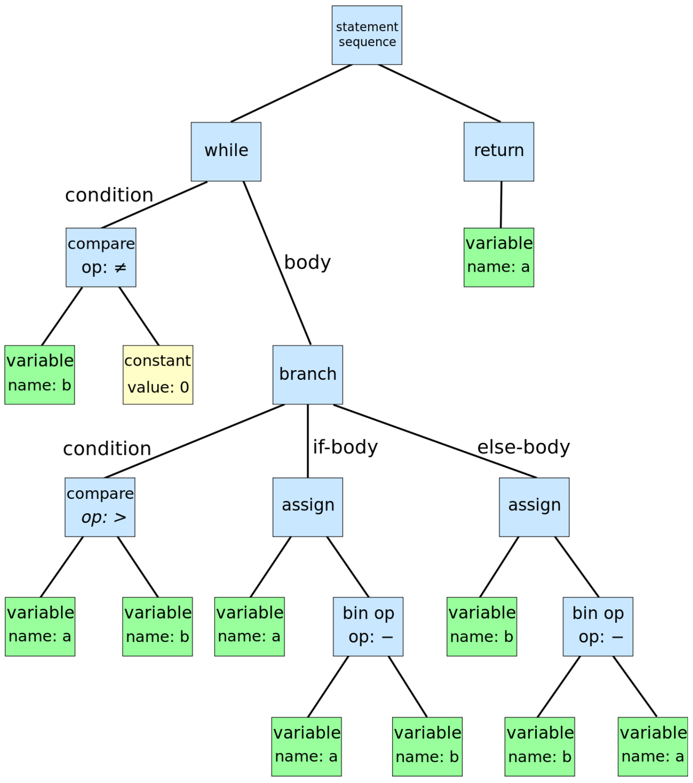


Figure 2.2: AST for the Euclidean algorithm for GCD computation[3]

Lexicosemantics

In lexical semantic analysis, words are not only seen as units but can be understood as composites of sub-words. A common task in NLP is the so-called *Named Entity Recognition (NER)* [39]. For this, a language model is trained on sentences with (foremost) labeled nouns. This makes it possible to annotate text such as

"Mozart wrote 41 symphonies between 1764 and 1788."

"[Mozart]$_{Person}$ wrote 41 symphonies between [1764]$_{Time}$ and [1788]$_{Time}$."

Another task that gained popularity with the rise of social media is *Sentiment Analysis*. By categorizing sentences into classes of connotations such as negative, positive and neutral by extracting subjective information. Applied to social media, sentiment analysis has shown to detect a wider range of emotions and sarcastic meaning of statements [15] by taking into account attached emojis.

Relational semantics

The analysis of relational semantics typically revolves around extracting meaning from a single sentence. This includes extracting relationships such as marital status with the use of ontologies. Another important concept is *Semantic Parsing* [29], the base of *Machine Translation*, *Question Answering* and *Code Generation*. An example of semantic parsing can be seen in figure 2.3.

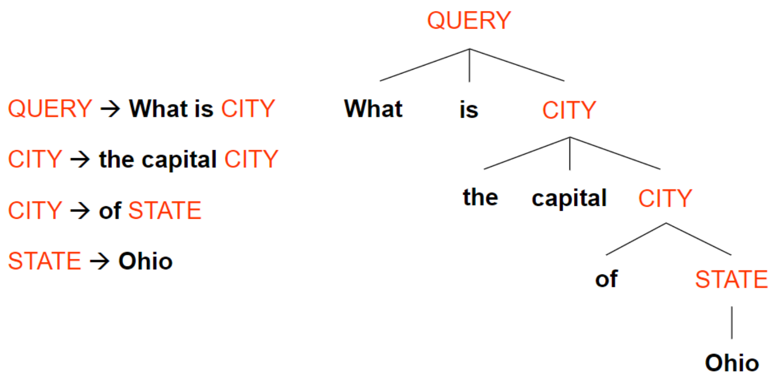


Figure 2.3: Example of a semantically parsed sentence[4].

Semantics beyond individual sentences

Imitating the human process of intuitively structuring bodies of text while reading, *Text Segmentation* has the computer split text into units: topics, sentences or words. With a

[4]https://www.cs.upc.edu/~ageno/anlp/semanticParsing.pdf

problem similar to that of *Tokenization*, this can require complex models when the text is written in a language that does not have sentence delimiters or the use of capital words at the start of each sentence.

Higher-level NLP applications

In reality, tools for more complex tasks such as *Text Summarization* and even *Question Answering* rely on the combination of several NLP subtasks. The complexity of the subtopic *Natural Language Understanding* has it being classified as an AI-hard problem[55].

2.1.3 Natural Language Understanding

As a combination of both syntactical and semantical analysis tools, teaching a machine to understand language can be described as the ultimate goal of Natural Language Processing research. Tasks like semantic parsing rely on the machine having learned an *understanding* of the human language. NLU generally refers to techniques that rearrange unstructured data to be learnable by machines.

2.2 Code Search

Programmers often need to search for code solutions to solve some of their own problems. Code Search is the task of retrieving relevant code from a query written using natural language. The concept is based upon *information retrieval*, but requires an understanding of both programming languages and natural languages, since the focus is on code semantic and not on its syntax like static code analysis tools such as linters [27].
While a model trained on natural languages could be evaluated with a GLUE (General Language Understanding Evaluation)[61] benchmark, models trained on code lacked a standardized evaluation dataset. With deep learning rising and dominating many machine learning fields, huge labeled datasets have cemented themselves as "standard" for evaluation, for example MNIST[5] for convolutional neural networks. Still, deep models struggle with highly structured data such as semantic code search. In 2020, GitHub initiated the *CodeSearchNet Challenge*[6] in order to evaluate the current state of Semantic Code Search by providing a baseline dataset for evaluating the task [24]. This dataset will be described in more detail in chapter 4.3. Existing tools for Code Search will be the topic of chapter 3. The authors also provide baseline models for code search. One idea was to join embeddings of code and natural language queries, as seen in figure 2.4. Both code and language get mapped onto vectors and in order for an encoded query to search for code snippets in the

[5]http://yann.lecun.com/exdb/mnist/
[6]https://github.blog/2019-09-26-introducing-the-codesearchnet-challenge/

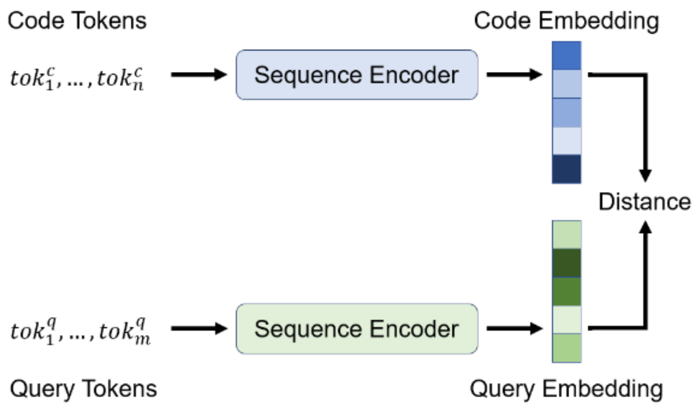


Figure 2.4: Code Search baseline model architecture

vicinity in this joint embedding space. To get a more in-depths understanding of both encoders and embeddings, they will be the main topic of the following sections.

2.3 Encoders

An encoder is a neural network that produces an encoded output, such as features maps, tensors or vectors, from an input. A common application in machine learning is the autoencoder [40].

2.3.1 Definition. Let $\phi : \mathcal{X} \to \mathcal{F}$, $\psi : \mathcal{F} \to \mathcal{X}$ be transition functions with ϕ representing the encoder part and ψ representing the decoder part. Then, an autoencoder should hold true to $\phi, \psi = \underset{\phi,\psi}{\arg\min} \|X - (\psi \circ \phi)X\|^2$.

The basic idea behind autoencoders is to simultaneously learn the encoder and the decoder in a way that the intermediate state $\phi(x) \in \mathcal{F}$ minizing the difference between x and $(\psi(\phi(x)) \in \mathcal{X}$. The error measuring the difference between the original data and the output is generally refered to as *reconstruction error*. For autoencoders, having a smaller intermediate state means reducing dimensionality of data [53].
Figure 2.5 shows a possible structure of a simple autoencoder. The mirrored design enables the model to reconstruct the input.

2.3.1 Sequence-to-sequence models

Sequence-to-sequence models [57], in short *seq2seq*, are the backbone of applications that implement machine translation, such as Google Translate, and those that handle speech recognition [46]. Seq2seq-models map inputs of fixed length to a fixed-length output, but contrary to LSTM-networks, can achieve this without requiring the same length for both.

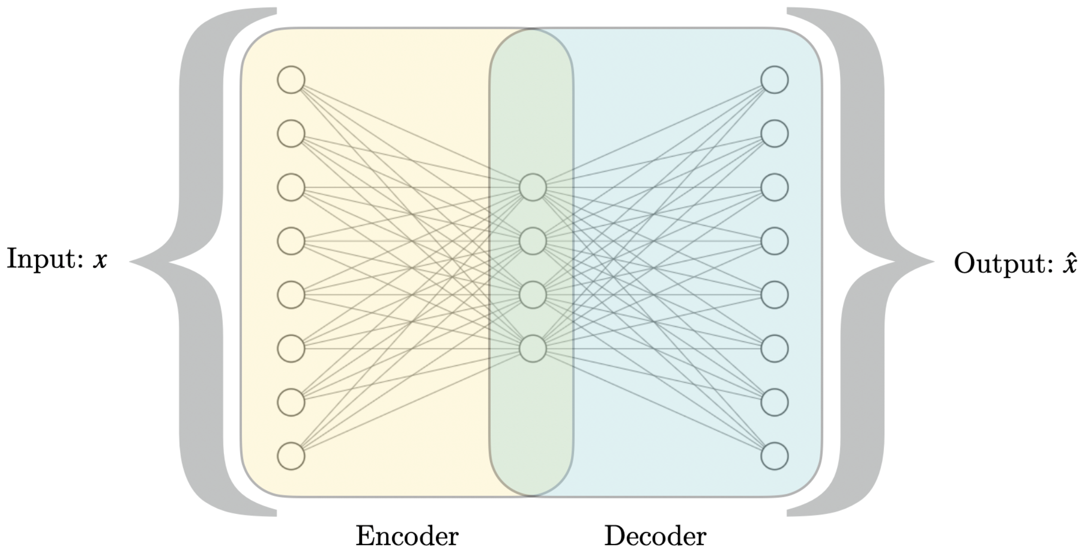


Figure 2.5: Simple Autoencoder Architecture.

Hence the application to translation: The task of translating natural language is rarely a one-to-one mapping of words.
In order to understand how those models work, there will be a brief section about Recurrent Neural Networks (RNNs) and Long-Short-Term-Memory networks (LSTMs) [4].

2.3.2 Definition. Given a sequence of inputs $(x_1, \dots, x_N)$, a RNN generates a sequence of outputs $(y_1, \dots, y_N)$ with

$$h_n = \sigma(W^{hx} x_n + W^{hh} h_{n-1}) \tag{2.1}$$

with W being a weight matrix, h_i the output of hidden layer i and

$$y_n = W^{yh} h_n \tag{2.2}$$

While the RRN is able to map sequences onto other sequences when the alignment and length is known, is struggles with tasks that don't fulfill those requirements. Additionally, RNNs fail to store longer temporal dependencies. This is especially problematic in the field of NLP, because we can not expect the network to predict a word, if the information from the beginning of the sentence is already gone.
LSTMs, with their ability to do just that, provide a solution [20]. The LSTM estimates the conditional probability $p(y_1, \dots, y_{N'} | x_1, \dots x_N)$, with N and N' not necessarily being the same length. With the use of special tokens for sentence delimiters (<*EOS*>), the model is able to define a distribution over sequences of all possible lengths. Sutskever et al. [57] showed that the combination of two different deep LSTMs, one for encoding, one for decoding, were able to learn a model for a different-length sequence-to-sequence task.

2.3.2 One-Hot encodings

The method of one-hot encoding makes it possible to vectorize categorial features.
"One-hot" describes a sequence of bits that have exactly one bit set to "1" and the rest to "0" [22]. This makes them different to the traditional use of bits, for example in binary. While a boolean state could easily be represented by a single bit that is either "0" or "1", a one-hot representation would result in using two bits: "01" and "10".
Therefore, instead of being able to represent 2^n entities with n bits, one-hot encoders are limited to exactly n entities.

Using one-hot encoding makes the required model simple and the encoding process fast. Since each category implies a new dimension, the trade-off is having large and sparse matrices, leading to the "curse of dimensionality" [6].

As an alternative to one-hot encoding, encoding to a lower dimension vector of real numbers has proven to be superior for a lot of tasks [8]. This alternative, embeddings, will be discussed in the next section.

2.4 Embeddings

In a mathemetical way, an embedding is an instance of a mathematical structure that is part of another structure.

2.4.1 Definition. [23] Let X and Y be objects. X is said to be embedded in Y if an injective and structure-preserving mapping function $f : X \to Y$ exists. For mapping functions f that embed, we use the notation $f : X \hookrightarrow Y$. This class of functions can also be refered to as *morphisms.*

Since embeddings preserve the structure of the original object while mapping onto a space of lower dimension, it can also be classified as a technique for dimension reduction.

In natural language processing, using embeddings is a way to map higher-dimensional data to lower dimensions while still retaining semantics of the language. The most commonly used class of embeddings in NLP are *word embeddings*, which will be explained in this section. Furthermore the problem of aligning embedding spaces and a way to tackle it, is discussed herein.

2.4.1 Word Embeddings

To obtain a representation of word meaning, the so-called word embeddings are used in NLP [28]. The idea is to learn features that allow mapping natural text to vectors of real numbers in order to obtain abstract representations of words.
The idea of learning representations dates back to 1986, the same paper that introduced

the concept of error back-propagation [52]. The authors state that a "network of neurone-like units" is able to construct appropriate internal representations of the input.
A way to learn word embeddings was developed by Mikolov et al. at Google research [37], a method dubbed *Word2Vec*. The following section will explore why and when word embeddings are useful.
If we consider sentences $s_{\mathbf{A}}$ = *"The weather is good"*, $s_{\mathbf{B}}$ = *"The weather is nice"* and $s_{\mathbf{C}}$ = *"The weather is bad"* and the task was to see how similar those sentences are, a one-hot encoder would generate outputs

$$e(s_{\mathbf{A}}) = [100000, 010000, 001000, 000100]$$
$$e(s_{\mathbf{B}}) = [100000, 010000, 001000, 000010]$$
$$e(s_{\mathbf{C}}) = [100000, 010000, 001000, 000001]$$

A similarity function would deem the semantically similar words, *"good"* and *"nice"* to be as different as *"good"* and *"bad"*, which have a very different meaning. An additional property is that all words in a one-hot encoded sentence are independent of each other, since the their relations can not be represented by such a binary encoding.
By generating distributed representations, word embeddings introduce dependencies between words. There are two methods for generating word embeddings with *Word2Vec*: *Continuous Bag Of Words (CBOW) Skip-Gram.*
The CBOW models learns to predict the words corresponding to the context by taking word context as input. Coming back to the example above, let's assume words *"weather"*, *"is"* as inputs to the neural network with a target of *"great"*. The one-hot encoded input words (also the context) then get compared to the one-hot encoded target and the error is measured. Through this, the model is able to learn the vector representation of the target word.
Figure 2.6 illustrates the architecture of both CBOW and Skip-Gram. Compared to the one of Skip-Grams, CBOW's architecture appears to be similar, albeit mirrored. Instead of aiming to predict the word based on its context, Skip-Gram maximizes a log-linear classifier of a word based on another word from the same sentence. Basically, each word gets used as an input and the models predicts the words before and the words that come after in a specific range. Both quality of word vectors and computational complexity increase with range [37].
Word embeddings are vectors of real numbers. An exemplary visualization is shown in figure 2.7. Values are scaled to be in $[-2, 2]$ with dark blue meaning the attribute is close to -2 and red meaning proximity to 2. On closer inspection, some attributes show similar values. While *"woman"* shares the same redness with *"girl"* in one of the features, in a different one it is similar to *"man"*. Without knowing the exact meaning of the features, it is intuitively clear, that there are relationships between those words and that they should be different ones.

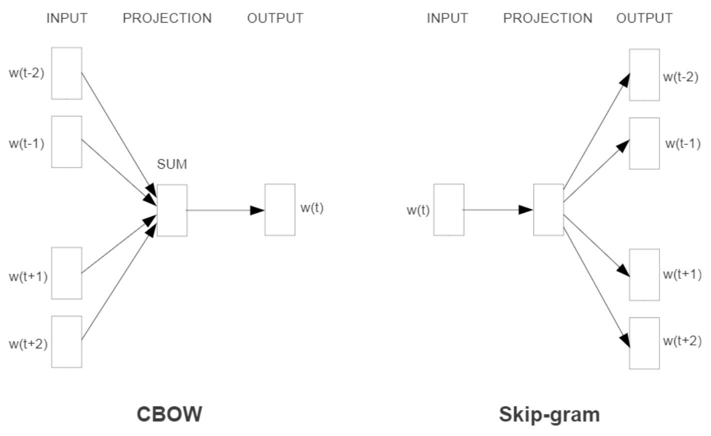


Figure 2.6: Comparison of the CBOW and Skip-Gram models [37]

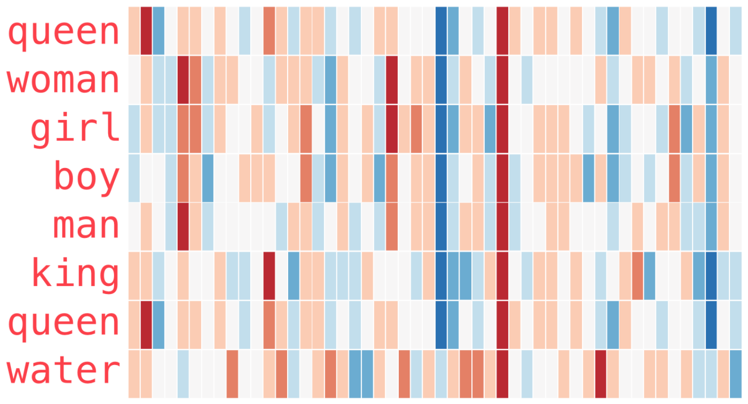


Figure 2.7: Visualization of word embeddings [7]

Another property of word embeddings is the concept of analogies. Since all real number vectors have the same length, it is possible to perform mathematical operations such as subtraction and addition in order to arrive at a different word. Going by intuition, starting with the word *"king"*, it should be able to subtract *"man"*, add *"woman"* and arrive at the word *"queen"*. As can be seen in figure 2.8, despite the resulting word being not exactly the same, it still is the word with the smallest distance to *"queen"* in the entire corpus.

Comparing two word vectors is commonly done by measuring the cosine similarity.

2.4.2 Definition. Let $\mathbf{A}$, $\mathbf{B}$ be two non-zero vectors of attributes. The cosine similarity between $\mathbf{A}$ and $\mathbf{B}$ is defined as $sim(\mathbf{A}, \mathbf{B}) = \cos(\theta) = \dfrac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$

[7] https://jalammar.github.io/illustrated-word2vec/
[8] https://jalammar.github.io/illustrated-word2vec/

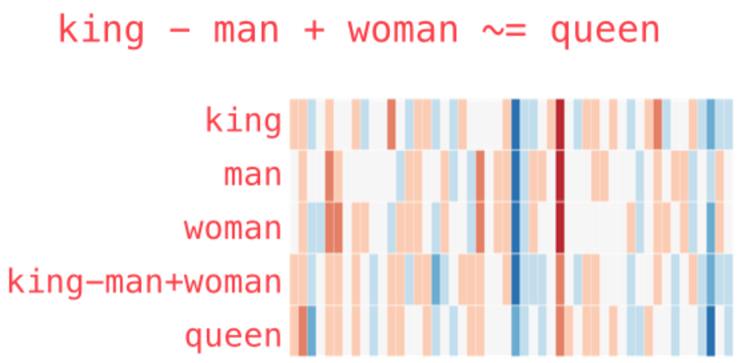


Figure 2.8: Visualizing the "king - man + woman ~= queen" analogy [8]

To illustrate this, a diagram of how the cosine similarity behaves is shown in figure 2.9.

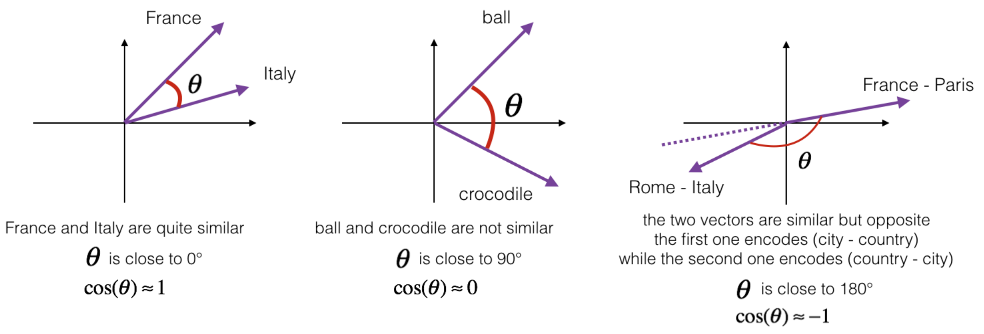

France
Italy
θ
France and Italy are quite similar
θ is close to 0°
cos(θ) ≈ 1
ball
crocodile
θ
ball and crocodile are not similar
θ is close to 90°
cos(θ) ≈ 0
France - Paris
Rome - Italy
θ
the two vectors are similar but opposite
the first one encodes (city - country)
while the second one encodes (country - city)
θ is close to 180°
cos(θ) ≈ −1

Figure 2.9: Cosine similarity between words [9]

One of the main problems of this approach, the limitation of those models only being able to represent a single meaning of a word, has since been solved by bi-directional transformers, which will be further discussed in the *Transformers* section of this chapter. An expansion of these word-level embeddings are *thought vectors* that allow for sentence- or even document-level embeddings. An embedding of former depth can also be refered to as *sentence embedding*.

2.5 Transformers

A transformer is a deep learning model that handles sequential data similarly to recurrent neural networks, but without the necessity to process the data in order [59]. Its main application is in NLP. The transformer architecture allows the network to be trained in parallel, which, since its first appearance, massively boosted the development of models with billions of parameters, mainly language models.
Transformers rely entirely on an attention mechanism to draw global dependencies between input and output [59]. The next part will go into *attention* in more detail.

[9]https://zhangruochi.com/Operations-on-word-vectors-Debiasing/2019/03/28/

2.5.1 Attention

Previous sections discussed how an encoder-decoder model processes sentences. A potential issue with using neural networks is that it has to compress all required information into a vector of fixed length. Since the appraoch is based on a training corpus, this can be problematic for sentences that are longer than the ones it is trained on. In [10], Cho et al. showed that the performance of such a model decreases with increasing input length.
Bahdanau et al. tackle this problem in [4] by learning sentence alignment and translation jointly.
Their architecture consists of a bidirectional RNN as encoder and a simple decoder. Different compared to a usual (unidirectional) RNN, a bidirectional RNN consists of two sub-RNNs, a forward one that reads the input in order $(x_1, \ldots, x_T)$ and calculates hidden states $(\overrightarrow{h_1}, \ldots, \overrightarrow{h_T})$ and a backwards one that reverses the input order into $(x_T, \ldots, x_1)$ to calculate backward hidden states $(\overleftarrow{h_T}, \ldots, \overleftarrow{h_1})$. For a word x_i, an annotation h_i is obtained by concatenating both hidden states, s.t. $h_i = [\overrightarrow{h_i}^\top; \overleftarrow{h_i}^\top]$. This way, the annotations contain summaries of both preceding and following words.
The decoder (next to the alignment model) uses this resulting sequence of annotations to compute a context vector. In the proposed architecture, the decoder defines a probability over a translation $\mathbf{y} = (y_1, \ldots, y_T)$ for a context vector c as

$$p(\mathbf{y}) = \prod_{t=1}^{T} p(y_t | \{y_1, \ldots, y_{t-1}\}, c) = \prod_{t=1}^{T} g(y_{t-1}, s_t, c) \tag{2.3}$$

The context vector c_i is computed by a weighted sum of annotations $(h_1, \ldots, h_T)$:

$$c_i = \sum_{j=1}^{T} \alpha_{ij} h_j \tag{2.4}$$

Weighted annotations let the model give more *attention* (hence the name) to certain parts of the sequence. By having a mechanism like this inside the decoder instead of the encoder, the information can be distributed among elements of the sequence. Those weights are computed by

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})} \tag{2.5}$$

2.5.2 Transformer Architecture

Figure 2.10 depicts the Transformer architecture. It combines several unique concepts. Firstly, it stacks layers for both encoder and decoder. The encoder stacks of identical layers each consist of a multi-head self-attention mechanism and a fully-connected feed-forward network. For the decoder stack, an additional multi-head self-attention mechanism is added. The concept of multi-head attention builds upon scaled-dot-product attention. In

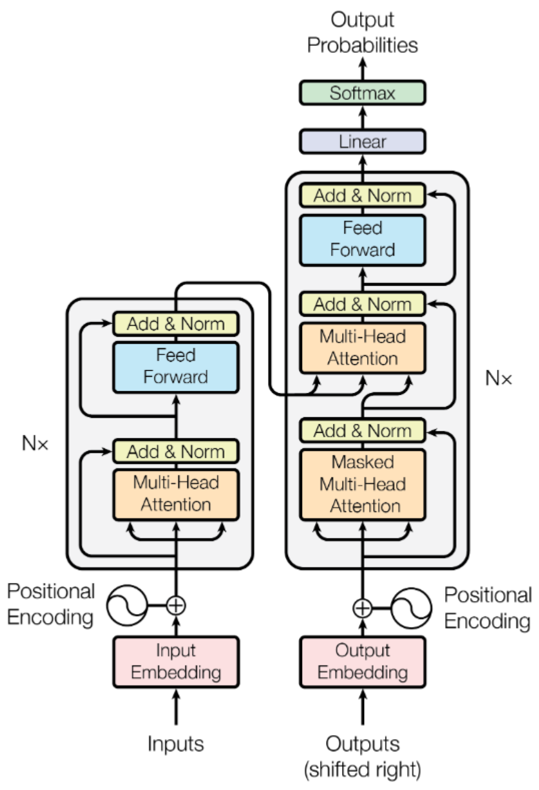


Figure 2.10: Architecture of the Transformer model [59]

general, an attention function maps a query and a key-value-pair to an output. For scaled-dot-product attention, since the Transformer architecture allows for parallel computation, for matrices of queries Q, keys K of dimension d_k and values V, the output matrix can be defined by

$$f_{Attention}(Q,K,V) = softmax\left(\frac{QK^\top}{\sqrt{d_k}}V\right) \tag{2.6}$$

Multi-head attention adds a dimension to the output by projecting queries, keys and values. Attention then gets calculated for each projection, concatenated and projected once again, resulting in projection dimensions $\{W_i^Q, W_i^K\} \in \mathbb{R}^{d_{model} \times d_k}$ and $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ with d_{model} being the dimension of embeddings.

$$f_{MultiHead}(Q,K,V) = concat(head_1, \ldots, head_h)W^O \tag{2.7}$$

$$head_i = f_{Attention}(QW_i^Q, KW_i^K, VQ_i^V) \tag{2.8}$$

This way, the multi-head attention model can reference information from multiple sources (representation subspaces) at different positions. A graphical comparison of the concepts is shown in figure 2.11,

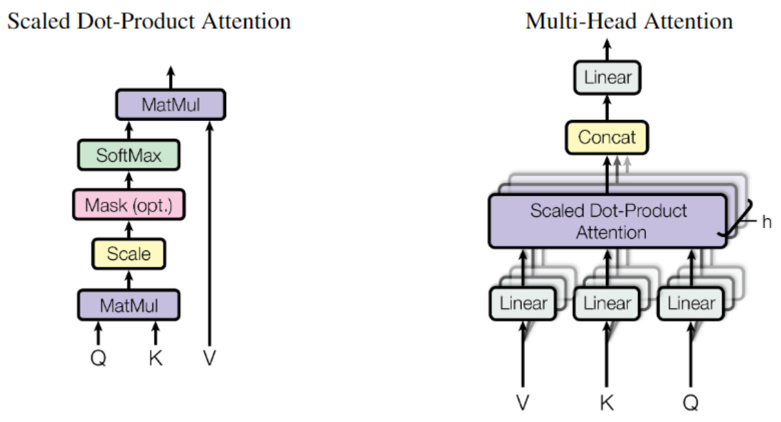


Figure 2.11: Architectural comparison of Scaled-Dot-Product Attention (left) and Multi-Head Attention (right)

2.5.3 Pre-trained Language Models

Pre-trained LMs, trained on large text corpora, enable the transfer to downstream tasks and prevent having to train a model completely from scratch [47]. This can be categorized as an instance of transfer learning, which already showed huge potential on several classification tasks [13]. ELMo, Embeddings from Language Models, is a deep contextualized word representation that models complex characteristics (word syntax and semantics) and how usage varies across different contexts. It is a bi-directional model, meaning a combination of a forward and backward LM (implemented by LSTMs), whose log likelihoods are jointly maximized during training [42]. As is shown in figure 2.12, its hidden states are

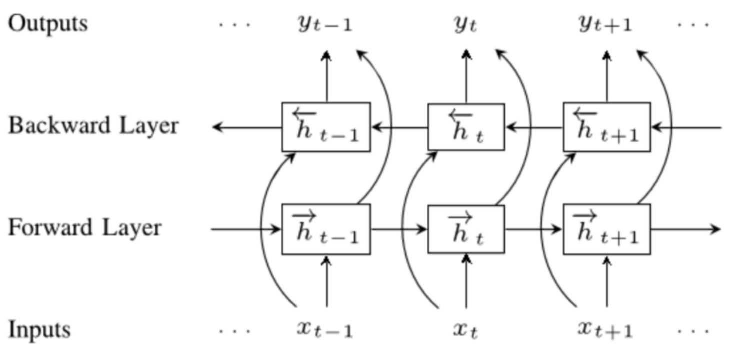

Outputs
$\cdots$ y_{t-1} y_t y_{t+1} $\cdots$
Backward Layer
$\overleftarrow{h}_{t-1}$ $\overleftarrow{h}_t$ $\overleftarrow{h}_{t+1}$
Forward Layer
$\overrightarrow{h}_{t-1}$ $\overrightarrow{h}_t$ $\overrightarrow{h}_{t+1}$
Inputs
$\cdots$ x_{t-1} x_t x_{t+1} $\cdots$

Figure 2.12: ELMo's bidirectional architecture [10].

combined to output context sensitive features.
With the release of Generative Pre-Training (GPT) [48], the problem of lacking labeled data for specific tasks was tackled and it increased performance of discriminately trained models.

[10]https://medium.com/duyanhnguyen_38925/create-a-strong-text-classification-with-the-help-from-elmo-e90809ba29da

The concept behind that approach is that models are trained in two steps: generatively for task-agnostic pre-training and discriminately for fine-tuning, meaning a specialization on a chosen target task. Unsupervised pre-training maximizes

$$L_1(\mathcal{U}) = \sum_i \log P(u_i|u_{i-k}, \dots, u_{i-1}; \Theta) \tag{2.9}$$

given a corpus of tokens $\mathcal{U} = \{u_1, \dots, u_n\}$, context window size k and a conditional probability P modeled by a neural network trained with gradient descent with parameters Θ.

2.5.4 Bidirectional Encoder Representations from Transformers

Devlin et al. enhanced the concept of using transformers for language understanding by developing a model named Bidirectional Encoder Representations from Transformers, short *BERT* [12]. The BERT model learns representations with bidirectional context on unlabeled text.

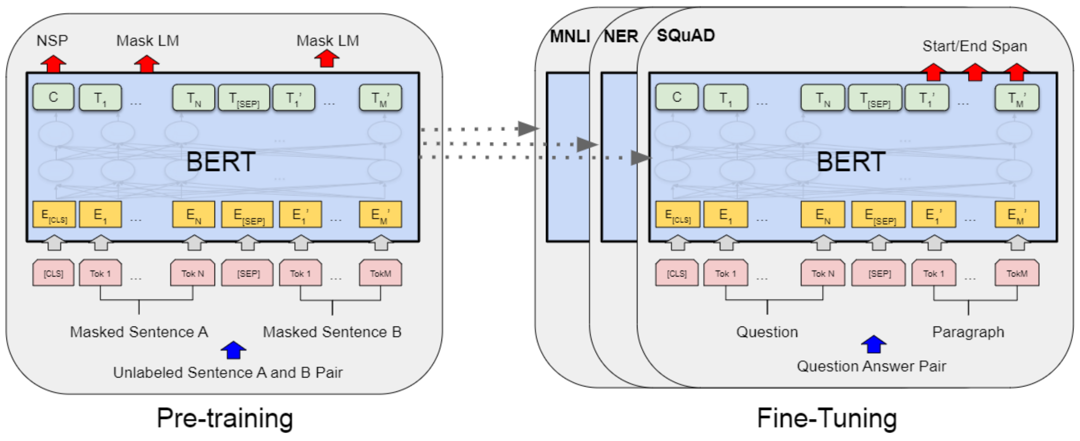


Figure 2.13: BERT pre-training (left) and fine-tuning (right). Output layers differ, all other layers are the same for both.[12]

As can be seen in figure 2.13, the BERT framework consists of two steps: Pre-training, in which the model gets trained on an unlabeled dataset over multiple tasks, and fine-tuning, where the parameters that resulted from the pre-training step are tuned. In the fine-tuning part, the model gets trained further on a (albeit significantly smaller compared to pre-training) labeled dataset from the downstream tasks.

BERT is trained on two unsupervised tasks: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). Both task will be described in the following section.

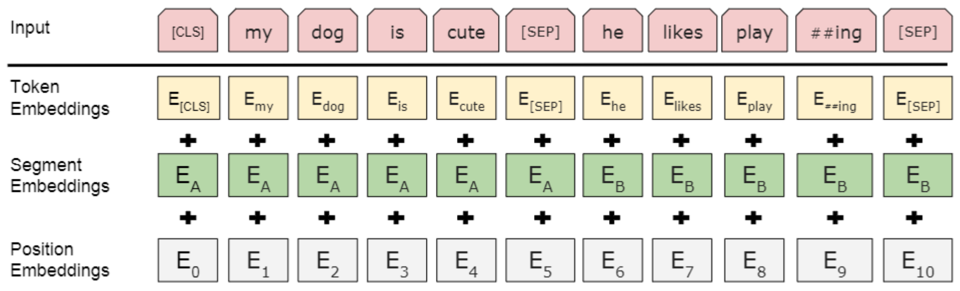


Figure 2.14: Input representation for BERT. Input embeddings consist of token embeddings, segment embeddings and positional embeddings. [CLS] preceeds every sample to mark its begin and [SEP] is a special token for representing sentence seperators.[12]

Masked Language Modeling

As mentioned above, unidirectional (left-to-right or right-to-left) models are not well suited for interpreting words in different contexts. To counter this, bidirectional models have been proposed. The issue with training a conditional LM to be bidirectional is impossible, due to the model being able to see itself and thus making any predictions pointless.
This leads to the task of prediction masked tokens. Certains parts of the input get hidden (*masked*) from the model by replacing it with a $[MASK]$ token and, in turn, making predictions of those original tokens non-trivial. To counterbalance discrepancies between pre-trained and fine-tuned models (where the $[MASK]$ token does not appear), the masked token sometimes gets replaced by a random token and at times not at all.

Next Sentence Prediction

The task of NSP is especially important for the task of Question Answering. Given a sentence s_A, the task is to predict the likeliest sentence s_B to follow s_A from a set of sentences. Pre-training of task is done one a dataset generated from a text corpus with half of the samples being pairs of (s_A, s_B), s_B being the actual next sentence, and the other half (s_A, s'_B) with s'_B being a random sentence from the full corpus.

[49] states, that the original transformer model is better suited for text-to-text tasks than either BERT or GPT architectures if scaled up and that all NLP tasks can be formulated as text-to-text tasks.

Chapter 3

Related Works and Tools

A multitude of tools have been released in recent years. This chapter will introduce some of them and evaluate if and how they are able to help with the proposed problem. At first, an overview of the current state of research and available tools for text analytics will be given, before secondly introducing research and exploring tools for analyzing code. Finally, this thesis will give an assessment of their applicability in the context of automatically enriching scientific source code.

3.1 Text Analytics

Text analytics, or text mining, is a research field that deals focuses on extracting information from natural language text.

3.1.1 Text Analytics Related Research

Knowledge Extraction from scientific sources:
Upadhyay and Fujii [58] proposed a software system that extracts sentences and keywords from large scale document databases. The system builds a hyper-graph as semantic representation scheme from PDFs, from which relations between sentences can be derived.
Ronzano and Saggion on Knowledge Extraction and Modeling from Scientific Publications [51], which is the base for the Dr. Inventor Framework [50].
In [56], Stathopoulos et al. assigned mathematical types to variables occurring in text from a manually annotated set of mathematical documents.
[54] gives a review of relevant knowledge extraction techniques, emphasizing the difficulty of finding relevant information amongst the rapidly growing amount of research publications.
GATE (General Architecture for Text Engineering[1]) is a community-driven open-source

[1]https://gate.ac.uk/family/

software toolkit for all-purpose text processing.

3.1.2 Text Analytics Tools

SemanticScholar[2] is a web application for navigating scientific literature. It supports automatic extraction of abstracts, tables, figures and citations from research papers. It also rates the papers impact based on volume and intent of their citations. Over 180 Million papers are currently available on SemanticScholar.
DeepDive [65] is a Text Mining tool developed by a research group from Stanford. It is a system that extracts information from so-called dark data, data that can be described as hidden data in entities like tables, figures and images, that are hard to process due to their lack of structure. DeepDive transforms unstructured dark data into a structured database. It has been most notably applied to domain-specific searches in geological data (GeoDeepDive) and documents from the dark web to fight human trafficking (MEMEX).

3.1.3 Detailed Overview: Dr. Inventor Framework

As mentioned in the previous section, the Dr. Inventor Framework[3] is a tool for automated analysis of scientific publications. The text mining framework supports identification of structural elements, enrichment of bibliographic entries, rhetorical characterization of sentences, linking of named entities and generating summaries. This section will give an overview of the framework's pipeline, which can be seen in figure 3.1.

Valid Input

The framework accepts both PDF and JATS XML documents. Depending on the input file format, additional pre-processing steps are necessary. Because of how XML contains raw text and environments for citations, these steps can be skipped.

Full Text Extraction

The pipeline makes use of the GROBID[4] library to extract the text from PDFs. GROBID is, at the time of writing, able to additionally extract information such as headers, references, or citations.

[2]https://www.semanticscholar.org/
[3]https://driframework.readthedocs.io/en/latest/
[4]https://github.com/kermitt2/grobid

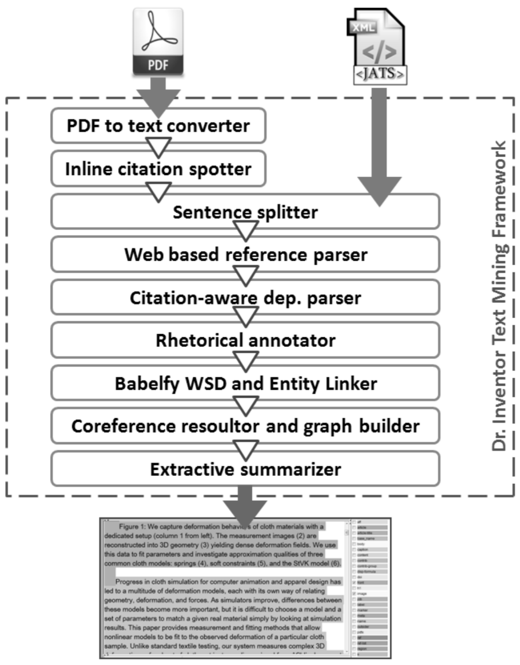

PDF
XML
<JATS>
PDF to text converter
Inline citation spotter
Sentence splitter
Web based reference parser
Citation-aware dep. parser
Rhetorical annotator
Babelfy WSD and Entity Linker
Coreference resoultor and graph builder
Extractive summarizer
Dr. Inventor Text Mining Framework

Figure 3.1: Architecture of the Dr. Inventor Framework Modules [51].

Inline Citations

Inline citations are spotted with the use of the Java Annotation Patterns Engine(JAPE) from the GATE toolkit. JAPE operates over annotations based on regular expressions and is particularly suited for pattern-matching and extraction of semantics.

Sentence Splitting

For sentence splitting, DRI utilizes ANNIE (a Nearly-New Information Extraction System), also from the GATE family. The splitter is independent from both domain and application. It is described as "a cascade of finite-state transducers which segments the text into sentences"[5].

Reference Parsing

The next step, web-based reference parsing, is done with FreeCite. The open-source library has since been retired. An alternative could be anystyle.io, though it has not been tested in this context. Within AnyStyle, the user can train parser and finder models.

[5]https://gate.ac.uk/sale/tao/splitch6.html#sec:annie:splitter

Citation-Aware Dependency Parsing

Open-source toolkit MATE[6] provides several tools for natural language analysis. It mainly consists of a dependency parser and a semantic role labeler. The Dr. Inventor Framework makes use of MATE for citation-aware dependency parsing.

Rhetorical Annotation

This part of the pipeline is realized within the framework through a language model based on a text corpus of 40 manually annotated documents. The papers belong to the field of computer graphics and each sub-group deals with a specific sub-field: Skinning, Motion, Fluid Dynamics and Cloth Simulation[7]. A general discoursive structure of research papers in computer graphics is assumed [17].
The Rhetorical annotator is a model trained in the DRI framework. The authors did not intend for other users to use the framework for training their own classifiers.

Knowledge Graph Builder

The term *Knowledge Graph* has become popular since the introduction of the Google Knowledge Graph[8], a knowledge base that enhances the search results, but the term stayed somewhat vague until the definition of Ehrlinger et al. in [14]:

3.1.1 Definition. A knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge.

The Dr. Inventor Framework builds a knowledge graph based on rules defined within the framework. The output format is a String with a RDF (Resource Description Framework) graph.
Visualization not built-in, since the framework is mainly intended for scientific data analysis, according to the authors. A possible way of visualizing the output graph is Gephi[9]. A different approach for language representation with knowledge graphs has since been proposed by Liu et al. [34].

Relevance for this thesis

The erstwhile plan of this thesis was to build a pipeline on top of the knowledge graphs that the Dr. Inventor Framework produces as its output. Even though the framework was released under an open-source license during the course of this thesis, it has not been used beyond testing. Partly due to the immense computing cost of processing just a single

[6]https://code.google.com/archive/p/mate-tools/
[7]http://sempub.taln.upf.edu/dricorpus
[8]https://blog.google/products/search/introducing-knowledge-graph-things-not/
[9]https://github.com/gephi/gephi/wiki/SemanticWebImport

document, not being able to access its model and its out-of-date components, a different route was taken, which will be the main focus of chapter 4.
Another key aspect was the lack of progress on generating knowledge graphs in the context of source code. This will, amongst others, be discussed in the next section of this chapter.

3.2 Code Analysis

3.2.1 Semantic Code Analysis Related Research

Wong et al. address the scarcity of in-code documentation in [62]. They propose a way of automatically generating comments from code-description mappings obtained from Q&A websites, such as StackOverflow.
After the success with word embeddings in NLP, the technique has also been applied to source code. Chen and Monperrus give an overview in [9]. Source code embeddings can be categorized into five categories: Tokens, expressions, APIs, methods and others.
Allamanis et al. [2] introduce neural equivalence networks to learn continuous semantic

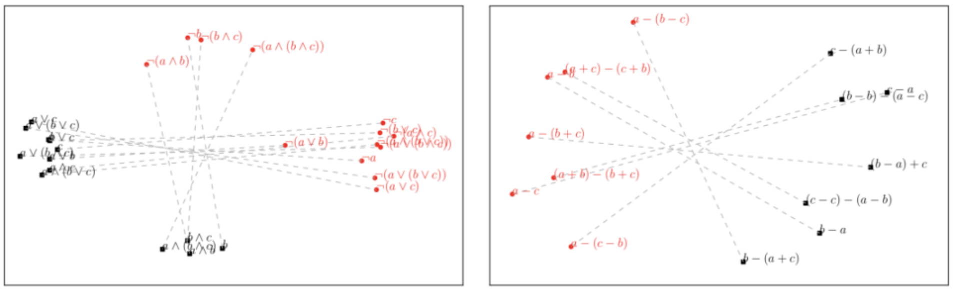
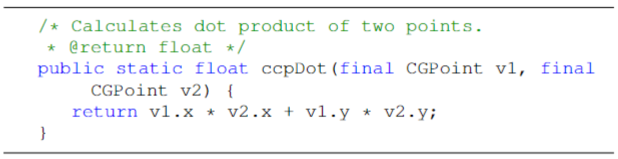


Figure 3.2: Visualization of an expression embedding space [2].

representations of algebraic and logical expressions. This is illustrated in 3.2.
In [32], Liang and Zhu propose a framework for automatic generation of descriptive com-



```
/* Calculates dot product of two points.
 * @return float */
public static float ccpDot(final CGPoint v1, final
    CGPoint v2) {
   return v1.x * v2.x + v1.y * v2.y;
}
```

Figure 3.3: Source Code documentation example of Code-GRU [32].

ments for blocks of source code. They use a RNN to extract features from the code and feed the embedded vector into a Gated Recurrent Unit (GRU). An output example can be seen in figure 3.3.
An approach to apply machine translation between source code and natural language has been published by Oda et al. in [41]. They use statistical machine translation to generate

pseudo-code from source code to improve understanding thereof, especially when coming across a previously unknown programming language.
Another recent contribution is [60] by Wan et al., in which the authors use Deep Reinforcement Learning to summarize source code. Their approach relies on the full AST rather than only the sequential code input.
Abstract Semantic Graphs as alternative to ASTs for semantic code analysis have been explored by Garner in [18].
Knowledge extraction from Source Code has been adressed by Azanzi and Camara in [3]. They adpot an ontology learning approach to generate knowledge graphs from code sources.
Jain et al. developed ContraCode [26], a self-supervised algorithm for learning semantic representations of programs with contrastive learning.
Pre-training contextual embeddings on a JavaScript corpus is done by Karampatsis and Sutton using the ELMo framework for the task of program repairing in [31].
Learning contextual embeddings of source code has been researched by Kanade et al. [30], which resulted in the CuBERT model.

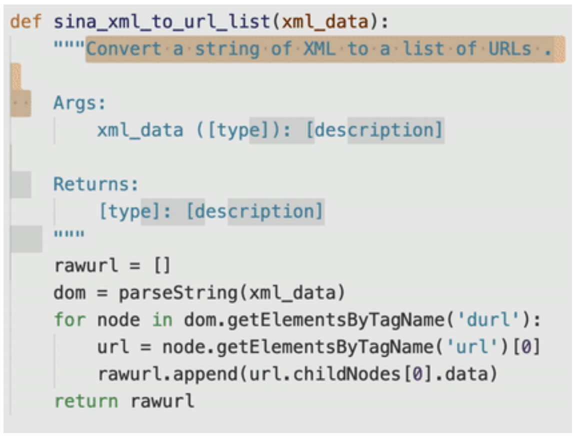
3.2.2 Code Analysis Tools and Resources

Tools for code analysis have been mostly syntactic. There is a multitude static program analysis tools or plugins for IDEs to help a programmer detecting errors before actually compiling (e.g. linters). Even having an output of precise compiler errors can be classified as a code analysis tool. While these kind of tools are certainly indispensable, they will not be the focus of this thesis.
For bringing together natural and programming languages, tools that do semantic code analysis, are paramount.
Github's *semantic* is a library for parsing, analyzing, and comparing source code across many languages[10] written in Haskell. It generates Haskell syntax types from tree-sitter grammar definitions. Tree-sitter[11] is a tool for generating parsers, which can build dynamic concrete syntax trees for a source code file.
Advances towards semantic versioning[12] have been a research topic for version control.
SourceGraph is a tool for universal code search and code navigation[13].
PapersWithCode[14] is a website dedicated to gathering machine learning research papers that have code repositories. Lately, automatic extraction of evaluation results has been the focus [15], although as of the time of writing, it still relies on its community to populate the databases. With the ability to compare results, this makes the website a tool for find-

[10]https://github.com/github/semantic
[11]https://github.com/tree-sitter/tree-sitter
[12]https://semver.org/
[13]https://github.com/sourcegraph/sourcegraph
[14]https://paperswithcode.com/
[15]https://github.com/paperswithcode/sota-extractor

ing current state-of-the-art approaches on specific tasks that have common datasets and evaluation methods.

Automatically generating basic descriptive DocStrings (figure 3.4) can be done with a VS



```python
def sina_xml_to_url_list(xml_data):
    """Convert a string of XML to a list of URLs .

    Args:
        xml_data ([type]): [description]

    Returns:
        [type]: [description]
    """
    rawurl = []
    dom = parseString(xml_data)
    for node in dom.getElementsByTagName('durl'):
        url = node.getElementsByTagName('url')[0]
        rawurl.append(url.childNodes[0].data)
    return rawurl
```

Figure 3.4: Automatic Generation of Python DocStrings with CodeBERT.

Code extension[16] powered by CodeBERT [16].

3.3 Translation Approaches

Neural Machine translation has its usual appliance in translations from one natural language to another natural language.

In general, approaches that learn vector representations of text that can be used for translations need parallel text or aligned documents, sentences or words. Barone [5] and Mohiuddin/Joty [38] propose adversarial autoencoders to automatically align those.

Pires et al. [44] showed that Multilingual BERT achieves good results at zero-shot cross-lingual tasks.

Wu et al. [63] researched unsupervised cross-lingual word alignments via contrastive learning, an example of which is illustrated in figure 3.5. In their approach, sentences are aligned beforehand and an alignment on a token level is learned during the training process.

While there are some instances of research for NL-PL translations in the previous section (3.2.1), the ones that automatically align are not suited for our specific task, since those approaches assume structural similarities between the translatable entities, as is the case for translations within natural languages or within programming languages. An approach that achieved this would open up a lot of possibilities in this field of research.

[16]https://github.com/graykode/ai-docstring

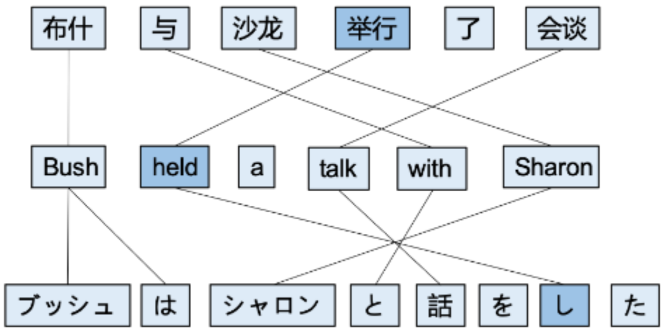


Figure 3.5: Example of cross-lingual (top to bottom: Chinese, English, Japanese) word alignment with SLUA [63].

Chapter 4

Implementation

This chapter will first introduce an outline for a pipeline for the task of extracting relations between research papers and their implementations and discuss ideas how to tackle each of the steps and potential issues that might arise. Then, the main idea will be described in detail, including libraries used, theory, pre-processing of data and the training setup. The implementation can be found on GitHub[1].

4.1 General Pipeline

One of the first, rough, drafts of the tool pipeline formulates as:

1. Input: Research paper (via arXiv link) and/or code repository (via GitHub link)
2. Extract knowledge representation from research paper
3. Extract knowledge representation from source code
4. Map knowledge representations into a shared space
5. Evaluate and rank mappings
6. Generate new comments from highest-ranking mappings
7. Insert comments into the original code
8. Output: Annotated version of the original source code

Since each of these steps pose challenging tasks, they will be further discussed below.

[1]https://github.com/pihari/enriching-ssc/tree/master

4.1.1 Step 1: Reading and Aligning Input

Reading input files varies in difficulty. Processing complex and information-rich data, such as Portable Document Format (PDF) files, requires resource-heavy tools. As described in chapter 3, the processing parts of the Dr. Inventor framework pipeline demonstrate the complexity. Multiple tools need to handle the data in sequence, but still, the output is then usable by subsequent libraries further down the chain.
Aligning paper sources with code sources requires either tools for finding references or a access to a database with present information about their alignment.
Assuming already processed data in *String* form, one possibility of former method is to look for said references in the sources. If we want to find a link to the code repository, it would be obvious to compare the strings in the research paper text with one containing, for example, "github" or "bitbucket". The problem with this is, that research papers do not only not necessarily include a linked repository, but also are not limited to a single referenced repository. This would make alignment impossible or unambiguous. Thus, additional information or a more complex method are required.
Another possibility would be to process the information contained in a code repository and find the referenced main paper there. Since a majority of papers get uploaded to arXiv, a good way to do this would be to look for and parse the raw *readme* markdown file, in order to search for a URL containing "arxiv.org".

4.1.2 Step 2: Knowledge Extraction from Research Papers

In chapter 3, we have already looked at some tools that are able to extract knowledge from both research documents. Building upon the foundations laid in chapter 2, a pipeline of NLP tools has to be constructed for this task. The extraction result should then be a representation of the knowledge that can be fed forward. These information can be represented by a knowledge graph or embeddings.

4.1.3 Step 3: Knowledge Extraction from Source Code

Since code analysis is mostly syntactically, extracting semantic information is challenging. Recent approaches to Code Search, as discussed in chapter 3, could prove useful.

4.1.4 Steps 4 and 5: Learning Knowledge Representations

The structural difference between natural languages and programming languages can intuitively be describes as less than the ones between a set of natural languages or a set of programming languages. Therefore, we can generally assume that the representations obtained from steps 2 and 3 are fundamentally different, even if they share the same type (e.g. a graph). Subsequently, the pipeline needs to map both of those representations into the same space.

4.1.5 Step 6: Generating Useful Comments

The usefulness of comments can be hard to quantify. The mapping from paper to code needs to be as exact as possible for a description of the code based on the research document. One of the possibilities is to analyze occurrences of symbols in both sources and add the containing sentence to the pool of comments, before in the end inserting them "as is" into the code.
An example of this would be hyperparameters. In most machine learning publications, there is at least some description of parameters that were tuned specifically. Now, if the source code contains a reference to a variable γ (gamma), it should be possible to search the paper for all occurrences of this variable and use the sentences to generate a comment.

4.1.6 Step 7: Enriching the Original Source Code

The generated comments need to get inserted into the original code base. A way to do this is by first parsing the original version of the code into an AST without discarding the comments (if available)[2]. After having generated the new comments, they could be inserted into the AST at the location of the specific function. Determining the location can be done by matching the function by an identifier such as its name. This step will not be part of this thesis due to a higher focus being on the previous steps.

4.2 Specific Pipeline Designs

In this section, the proposed pipeline will be illustrated in several steps.

4.2.1 Sample Generation Pipeline

The pipeline for generating samples from online data sources is shown in figure 4.1.
In the first step, data is gathered from online sources. This is done by cloning repositories to get the python code files and downloading the .tex source files from arXiv[3]. With user-defined criteria, the raw data gets processed in the next step. Finally, the training samples are generated by aligning the samples from each data source with user-defined alignment criteria.

[2]https://pypi.org/project/horast/
[3]https://arxiv.org/

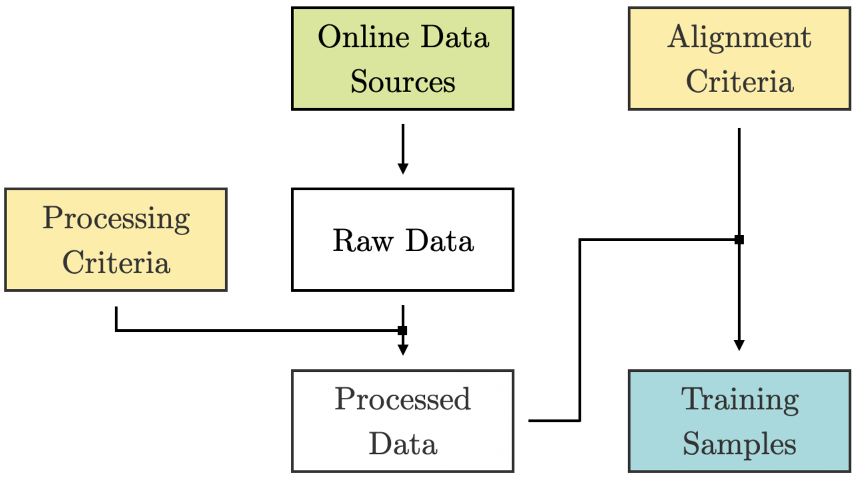


Figure 4.1: Sample Generation Pipeline

4.2.2 Training Pipeline

From the previously generated training samples and the encoders, the embeddings get produced. These then get fed batch-wise into a neural network that outputs the model. This is illustrated in figure 4.2.

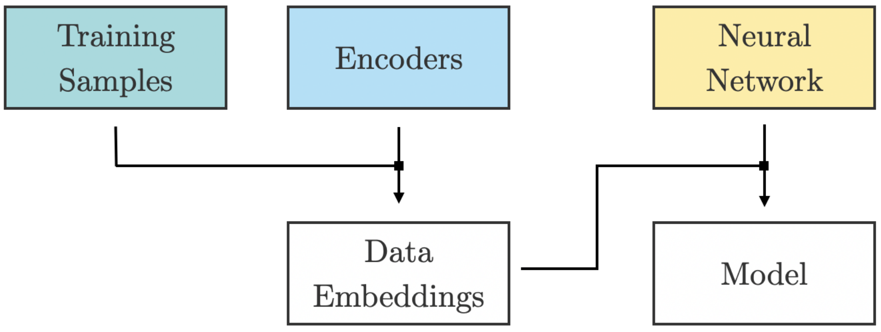


Figure 4.2: Training Pipeline

4.3 Data Acquisition

This chapter will explore a variety of datasets and emphasizes issues with utilizing them for the given task of enriching scientific source code. An overview of data processing will be given in the following part, before introducing the concept of tokenization in language processing and explaining how the pre-processing pipeline for the task looks like.

4.3.1 Data Requirements

For the task of jointly extracting information from research papers and code bases in order to find relations between them, certain requirements have to be met:

1. There has to exist a relation between a data pair of paper and code.
2. There has to be a way to align paper and code.
3. One of the following:
   (a) There has to exist a suited annotated dataset for the task.
   (b) Data has to be manually annotated for this thesis.
   (c) A self-supervised approach is taken.

Since, for item 3, there has not been found a suitable already annotated dataset and evaluating both of the others is beyond the scope of this thesis, we will proceed with focusing on item (c), the self-supervised approach.

4.3.2 Available Datasets

There is a broad availability of datasets for general-purpose natural language understanding. In this section, the most common classes will be introduced.
The first class is the *text corpus*. These corpora are large collections of structured natural language text and are at the foundation of the unsupervised pre-training step described in chapter 2. A selection was published online[4].
The second class is manually labeled data. Since the process of annotating data is tedious, the labeling can be crowd-sourced, meaning everyone within a community can contribute to the labeled dataset. An example of this is the Stanford Question Answering Dataset (SQuAD). It is a reading comprehension dataset, consisting of questions posed by users on a set of Wikipedia articles, where the answer to every question is a segment of text from the corresponding reading passage[5]. For NER, despite there being a collection of datasets in a curated list [6], none of these fit the scientific context.
While in some domains, labeled datasets at a large scale can be obtained through crowd-sourcing, such dataset in a scientific context is problematic due to the lack of experts required to annotate those [7]. A few exceptions for annotated research papers exist on GitHub[7] or PapersPlease[8].

[4]https://www.corpusdata.org/
[5]https://rajpurkar.github.io/SQuAD-explorer/
[6]https://github.com/juand-r/entity-recognition-datasets
[7]https://github.com/guptakhil/research-papers
[8]http://papersplease.ml/

This limits the possibilities of using supervised learning approaches and, for one, leads to a need for labeled data in research-related NLP tasks. At the same time, a different approach gained popularity in recent times: Self-supervised learning.
Self-supervised learning describes the process of learning to automatically annotate an unlabeled dataset by generating a supervisory signal. It is also used to learn a representation of the data. The autoencoder can be classified as self-supervised [19], since they are essentially unsupervised (with no class information being necessary for the training task), but at the same time mostly get used for classification (hence supervised) tasks. In language models, the previously mentioned text corpora get used as the source for self-supervised learning. For scientific literature, there exist corpora like the Semantic Scholar Open Research Corpus (S2ORC[9]) from which the vocabulary SciVocab[7] is created.

Datasets for specific tasks like code understanding, especially in a scientific context are sparse. While published annotations like *The Annotated Transformer*[10] are arguably very helpful to the machine learning community, the existence of such annotated datasets is certainly not sufficient to consider a supervised approach for the proposed task.

One approach could be to, instead of dealing with the code semantics, extract the variables and function parameters via the AST and to train a NER classifier on the task of classifying entities as variable. The issue is, that such a task needs a dataset with annotated entities, and, at the time of writing, such a dataset does not exist and would have to be created by manual annotation. Additionally, while having a language model that *understands* how variables fit into a sentence, the added value of such a classifier for just the variable detection is questionable if the extraction from papers can be done automatically.

4.4 Data Preparation

4.4.1 Tokenization

Tokenization[11] is an important part of processing text.

4.4.1 Definition. Given a character sequence $S = (S_0, \dots, S_n)$ and a defined document unit, tokenization is the task of splitting the sequence into tokens $s_i \in S_i$ for all $i \in 1, \dots, n$.

If we once again look at the example from figure 2.14, it is clear that the original sentence is "My dog is cute. He likes playing.". Splitting the sentences into its parts results in ["my", "dog", "is", "cute", ".", he", "likes", "playing", "."].

[9]https://github.com/allenai/s2orc
[10]https://nlp.seas.harvard.edu/2018/04/03/attention.html
[11][12]

Tokenizing words can be useful for language models, such as splitting "playing" into ["play", "ing"]. In order to make sense of such a split word, the relation is represented by adding a prefix "##". The final result then is ["play", "##ing"], the verb in its basic form and its -ing suffix.

Since there are no suitable datasets available for the proposed task, data has to be gathered first.
For learning the proposed task, the data has fulfill the requirements stated in chapter 4.3.1. Existing relations between data are provided by paperswithcode.com in form of a file that records the needed links between research publications and repositories with their respective implementations. For each paper, the following data can be accessed: paper url, paper title, arxiv id, abstract url, direct arxiv link to pdf, repository url, flags for references to each other in paper and on github and finally the used framework if available.
This makes it, for one, possible to automatically clone the repositories.
Additionally, from the arXiv id, with a script, the source files can be directly downloaded from "https://arxiv.org/e-print/{id}".
For this thesis, a limitation of only papers from January 2019 has been chosen.

4.5 Preprocessing

The preprocessing step is a big part of this pipeline. Since the data is gathered from two sources, the process is different for each of them.

4.5.1 Regular Expressions

A regular expression (RegEx) is a formal description of a pattern. The set of possible regular expressions for a given string is either empty or infinitely large.

4.5.1 Definition. For a given finite alphabet Σ, the empty set $\emptyset$, the empty string ϵ and all characters $\sigma \in \Sigma$ are regular expressions. Given regular expressions R_1 and R_2, (R_1R_2) is defined as their concatenation, $(R_1|R_2)$ as their union and R_1^* as smallest superset described by R_1 with $\epsilon \in R_1^*$.

Regexes are generally used in code for matching string patterns. Some of its metacharacters will be introduced in 4.1 below.

4.5.2 String Types in Python

In Python, regexes have to be described with raw strings. They can be declared by prepending the character "r" to the quotation mark that announces the beginning of a string.
A different type of strings in python are the f-Strings. They can be used to improve formatting of strings by making it possible to include variables within the string. The

Character	Meaning
.	Matches any character.
( )	Groups pattern elements together.
+	Preceding character has to occur at least once.
?	Prededing character can occur at most once.
*	Preceding character can occur for any number of times.
{M,N}	Specifies the range of number of occurences for the preceding character.
[ ]	Describes a set of possible character patterns.
|	Separates possible pattern matches. Equivalent of OR.
^	Matches the start of the string.
$	Matches the end of the string.

Table 4.1: Common RegEx characters.

prepended literal is an f. A variable or an expression has to be inside an environment of curly brackets.
To give an example of usage: `f"My name is {name}. I live in {city}."` would replace the placeholders with the strings stored in the variables `name` and `city`.
It is possible to combine r- and f-Strings by prepending both.

4.5.3 Processing LaTeX Files

As discussed in chapter 3, the available LaTeX parsers are not sufficiently modular to solve the proposed task. Therefore a pre-processing pipeline for .tex files has to be implemented. Since the idea is to create sentence/function pairs based on occurring variables, for this step, the sentences containing them have to be extracted. Generally, when writing a LaTeX document, refering to a variable v is done by embedding it in an environment `$`v`$`.
This assumption makes it possible to extract most variables used in a document by matching the syntax similarly to [43]. The corresponding r-string `r"(.*?)\$(.*?)\$(.*?)"` matches all strings in which two `$` symbols occur. To prevent unwanted matches, a filtering system should check possible mismatches. Typical mismatches that were found include:

- Numbers inside an inline match block, for example `$2$`
- Equation blocks `$$...$$`
- Sequential variable blocks in one sentence: `$`v_1`$...$`v_2`$`

In addition to identifying sentences that have variables in them, further modifications are needed. Because LaTeX syntax includes environments such as `\begin{...}`, parts inside specific environments are excluded. Specifically, figures, equations, tables, algorithms and align environments were all filtered out. With a combination of aforementioned r- and f-strings these can easily be found by matching the regular expression `rf"\\{cmd}{{{env}}}"`, where `cmd` can be `begin`, `end` or any of the structural commands such as `chapter` or `section`. Variables that are inserted via command (e.g. `\{alpha}`) get changed to just the variable name (`alpha`) and equally, math environments around words (`\mathcal{},...`) get removed.

Sentence splitting is necessary for the model to be trained on sentence level embeddings. It is important to define a regular expression that only splits at dots that are sentence delimiters and ignores all other occurrences. The given expression is defined by `r"(?<!˙.)(?<![A-Z][a-z])̇(?<=|̇)"`.

4.5.4 Processing Python Code Files

Python code files can be parsed into an Abstract Syntax Tree, in short: AST. The tree is a representation of the syntactic structure of the source code. We can traverse the AST to gather sets of node types, such as *functions*, *variables* and *imports*.
Since the idea is to align sentences and functions by variable, we need to create certain collections when processing the files.

1. Function dictionary: A collection of all functions available in the file with its name as the key.

2. Tokenized function dictionary: A tokenized version of the function dictionary.

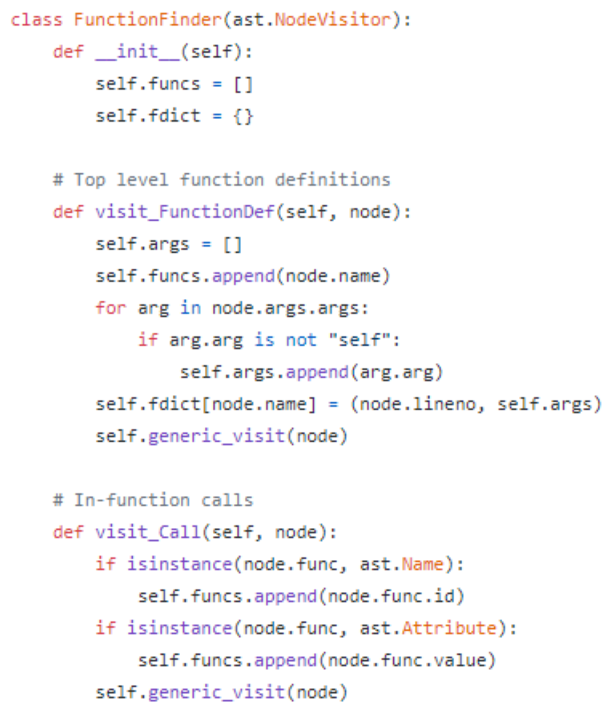
3. Variable dictionary: For each occurring variable, a list of tokenized functions that uses them.

The implementation is built on top of Python's *ast* module[13]. Custom implementations for function and variable extractions are shown in figures 4.3 and 4.4.

4.5.5 Aligning and Generating the Samples

Aligning and sample generation can be done in one step. The preceding processing step yields dictionaries grouped by variables, which now can be used to simply align them and generate samples for each combination.

[13]https://docs.python.org/3/library/ast.html



```
class FunctionFinder(ast.NodeVisitor):
    def __init__(self):
        self.funcs = []
        self.fdict = {}

    # Top level function definitions
    def visit_FunctionDef(self, node):
        self.args = []
        self.funcs.append(node.name)
        for arg in node.args.args:
            if arg.arg is not "self":
                self.args.append(arg.arg)
        self.fdict[node.name] = (node.lineno, self.args)
        self.generic_visit(node)

    # In-function calls
    def visit_Call(self, node):
        if isinstance(node.func, ast.Name):
            self.funcs.append(node.func.id)
        if isinstance(node.func, ast.Attribute):
            self.funcs.append(node.func.value)
        self.generic_visit(node)
```

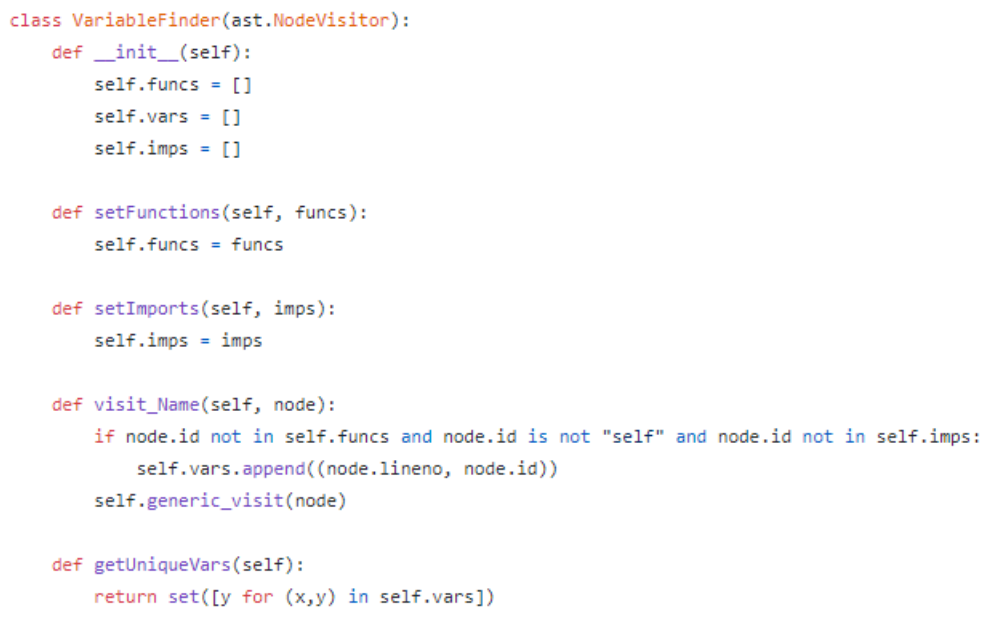
Figure 4.3: Custom implementation for extracting functions from a Python AST.

```
Require: $dict_s, dict_f$
1: $V_f \leftarrow dict_f.keys()$
2: samples $= \{\}$
3: for all $v \in V_f$ do
4:     if $v \in dict_s$ then
5:         for all combinations of entries $i, j$ do
6:             add $(dict_s[v][i], dict_f[v][j])$ to list of samples
7:         end for
8:     end if
9: end for
```

Algorithm 4.1: Sample Generation from Processed .tex and .py files

4.6 Training Pipeline

In this section, a simple setup for a pipeline for training the model will be presented. First, the backbone models of SciBERT and CodeBERT will be introduced. Then, the training architecture will be outlined.



```
class VariableFinder(ast.NodeVisitor):
    def __init__(self):
        self.funcs = []
        self.vars = []
        self.imps = []

    def setFunctions(self, funcs):
        self.funcs = funcs

    def setImports(self, imps):
        self.imps = imps

    def visit_Name(self, node):
        if node.id not in self.funcs and node.id is not "self" and node.id not in self.imps:
            self.vars.append((node.lineno, node.id))
        self.generic_visit(node)

    def getUniqueVars(self):
        return set([y for (x,y) in self.vars])
```

Figure 4.4: Custom implementation for variable finding in a Python AST.

4.6.1 Idea

The fundemental idea is to encode both the sentence and the function for each sample. Using Pre-trained models based on Bidirectional Transformers (see chapter 2) as encoders is a promising way to keep semantic information. The resulting embeddings then have to be combined into a shared space. A large collection of pre-trained language models is available on *Hugging Face*[14], an open-source website for facilitating building, training and deploying state of the art models.

4.6.2 SciBERT

SciBERT [7] is a pre-trained language model for scientific text. Compared to BERT [12], SciBERT is architecturally the same, but the training data consist of solely scientific text. The authors construct a vocabulary based on WordPiece, called SciVocab, that contains the most frequent words. The percentage of shared words between SciVocab and BERT's BaseVocab is only 42%. The vocabulary is used for unsupervised tokenization.
SciBERT is trained on a subset of over one million full-text papers from SemanticScholar, with 18% from the domain of computer science. SciBERT can be fine-tuned on several tasks: Named Entity Recognition (NER), Text Classification, Relation Classification, Dependency Parsing and PICO Extraction specifically for medical applications. Even without finetuning, SciBERT outperforms comparable models on tasks like NER for the computer

[14]https://huggingface.co/

science domain. With fine-tuning, SciBERT increases its F1 score up by up to 5.59%. The F1 score (also F-measure) is a weighted average of the precision and recall, where the closer the score is to 1, the better an evaluated method performs.
The pre-trained SciBERT model is provided by Hugging Face[15].

4.6.3 CodeBERT

CodeBERT [16] is, as the name suggests, a bimodal pre-trained model for programming and natural languages. It learns general-purpose representations that enable downstream applications that involve both programming and natural languages, such as code search and generation of documentation. In comparison to unimodal BERTs, like SciBERT, the model has to process data from multiple sources with inherently different structures. It is, however, able to process unimodal data (either NL or PL) as well, making it the first model to achieve this task.
CodeBERT is trained on data pairs of *(code, comment)* in 6 programming languages and its english documentation obtained from Github code repositories.
Microsoft's pre-trained CodeBERT model is being provided by Hugging Face[16]. Its architecture is based on the RoBERTa model architecture [35].
Similar to how natural language is interpreted by NL BERT models, SciBERT regards source code as a sequence of tokens. Input is a concatenation of two segments, one NL and one PL: $[CLS], w_1, w_2, \ldots, w_n, [SEP], c_1, c_2, \ldots, c_m, [EOS]$. CodeBERTs output representation consists of a contextual vector representation of each token for both NL and PL and a representaion of [CLS], which is the aggregated representation of the sequence.
During pre-training, the model uses two objectives, masked language modeling (MLM) and replaced token detection (RTD) [11].

MLM Objective

Given an input pair $x = (w, c)$ with t being a sequence of NL tokens and c a sequence of PL tokens, a set of positions m^w and m^c gets randomly replaced by a $[MASK]$ token with $m_i^w \sim \text{unif}\{1, |w|\}$ for $i \in \{1, \ldots, |w|\}$ and $m_i^c \sim \text{unif}\{1, |c|\}$ for $i \in \{1, \ldots, |c|\}$. The objective of MLM is a prediction of the masked out tokens, w^{mask} and c^{mask}. For a token-predicting discriminator $p^{D_{MLM}}$, it can be formulated as:

$$\mathcal{L}_{MLM}(\theta) = \sum_{i \in m^w \bigcup m^c} -\log p^{D_{MLM}}(x_i | w^{mask}, c^{mask}) \tag{4.1}$$

[15]https://huggingface.co/allenai/scibert_scivocab_uncased
[16]https://huggingface.co/microsoft/codebert-base

RTD Objective

Instead of using only bimodal data, the RTD task uses both unimodal and bimodal data. Two separate n-gram language models are used as generators, p^{G_w} for NL and p^{G_c} for PL. We draw $\hat{w}_i \sim p^{G_w}(w_i|w^{mask})$ for $i \in m^w$ and $\hat{c}_i \sim p^{G_c}(c_i|c^{mask})$ for $i \in m^c$ respectively. The discriminator $p^{D_{RTD}}$ is trained on classifying whether a word is the original word or a *fake* word $w^{corrupt}$. The objective loss is given by

$$\mathcal{L}_{RTD}(\theta) = \sum_{i=1}^{|w|+|c|} \Big(\delta(i) \log p^{D_{RTD}}(x^{corrupt}, i) + (1 - \delta(i)) \big(1 - \log p^{D_{RTD}}(x^{corrupt}, i)\big)\Big) \tag{4.2}$$

$$\delta(i) = \begin{cases} 1, & \text{if } x_i^{corrupt} = x_i \\ 0, & \text{else} \end{cases} \tag{4.3}$$

The total training objective can now be written as

$$\min_\theta \mathcal{L}_{MLM}(\theta) + \mathcal{L}_{RTD}(\theta). \tag{4.4}$$

An illustration of the RTD objective can be seen in figure 4.5.

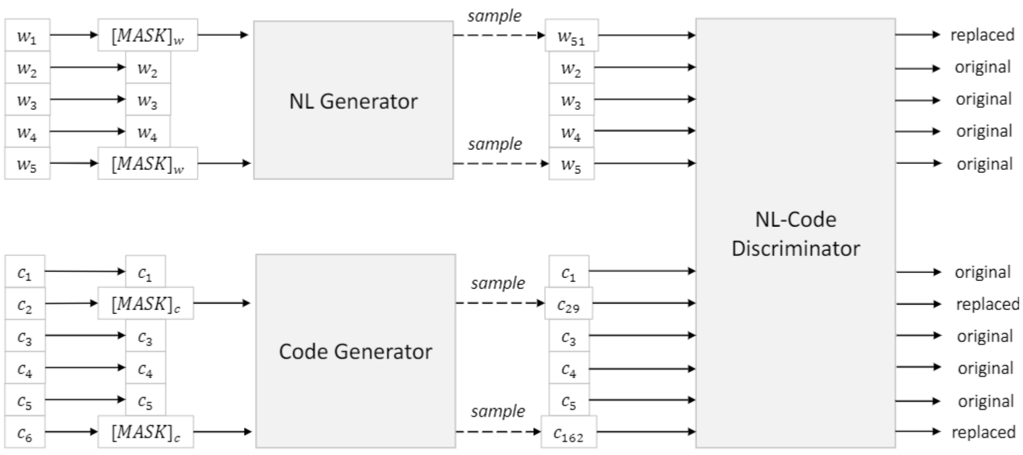


Figure 4.5: RTD objective within CodeBERT. Both generators are language models. The discriminator is the target pre-trained model.

4.6.4 Training Architecture Design

In order to train a model, the architecture has to be designed first.

Figure 4.6 illustrates how the design is meant to work. Training sample pairs get split into text and code samples. Text samples get encodes by SciBERT, while code samples get encoded by CodeBERT. Both result in embeddings of their respective type. An autoencoder is trained for both with a shared intermediate state space. The training gets controlled by a shared loss function that measures both reconstruction errors and similarity of the

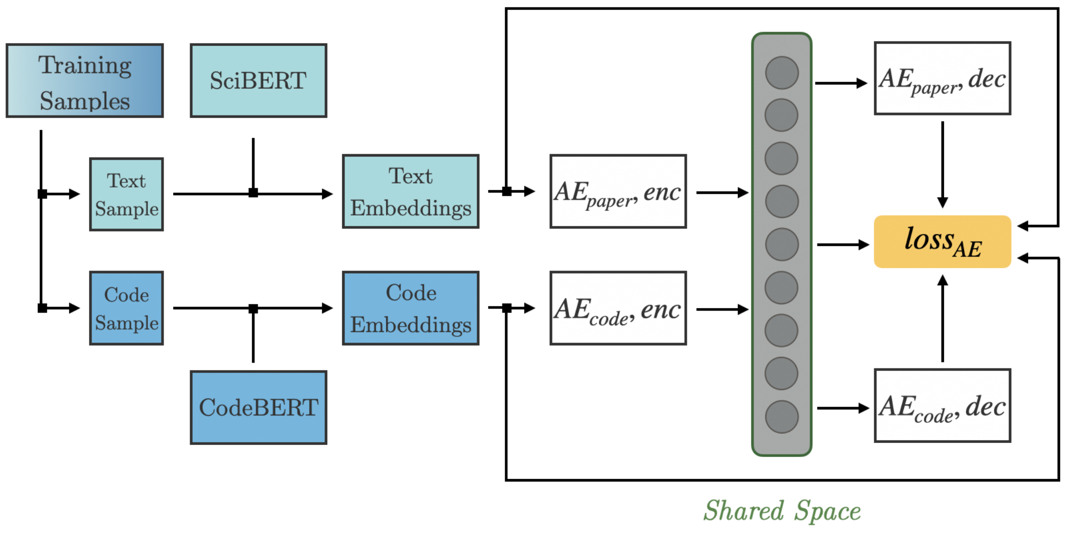


Figure 4.6: The proposed Training Setup

intermediate encoded state.

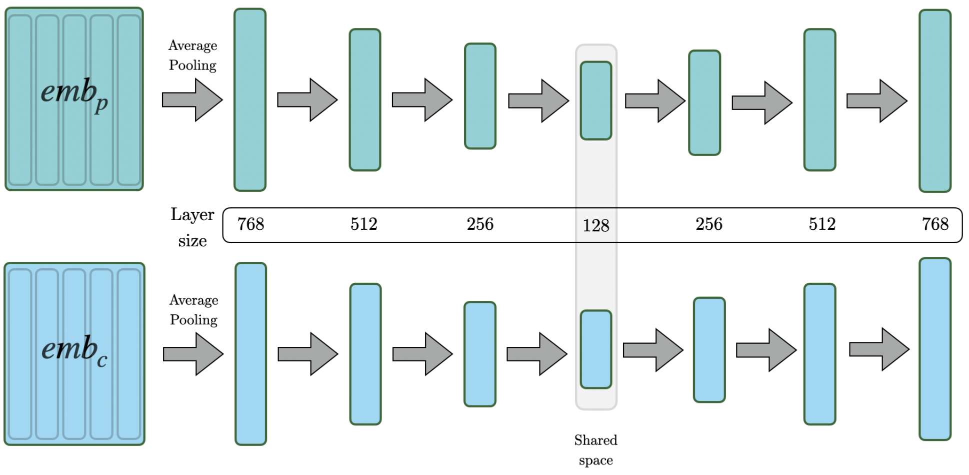


Figure 4.7: Autoencoder architecture for the task. Top: Paper embeddings. Bottom: Code embeddings.

Figure 4.7 shows the layers of the proposed simple parallel autoencoders. Both consist of two pooling layers, a state and two unpooling layers, each with a ReLU activation function.

$$f_{ReLU}(x) = \max(0, x) \tag{4.5}$$

After the final layer, a sigmoid activation function is used.

$$f_{sigm}(x) = \frac{1}{1 + e^{-x}} \tag{4.6}$$

The input embeddings get average pooled before getting put in the autoencoders, since averaging the complete sequence of hidden states of the embeddings is a better representation of semantics[17].

For the network, a custom loss function has been implemented. Because we want to evaluate the reconstruction of the two autoencoders at the same time and additionally evaluate the similarity of the encoded spaces, a composite loss function gets used. First, we define two mean-squared error loss functions for measuring the reconstruction error of the encoders/decoder system. A smaller error means that the autoencoder has learned a better abstract representation from which the original can be reconstructed.

$$\mathcal{L}_{paper} = \frac{1}{n}\sum_{i=1}^{n}(s_i^p - \hat{s_i^p})^2. \tag{4.7}$$

$$\mathcal{L}_{code} = \frac{1}{m}\sum_{j=1}^{m}(s_i^c - \hat{s_i^c})^2. \tag{4.8}$$

For measuring the similarity of the encodings, a cosine similarity function is used. It is defined by

$$\mathcal{L}_{sim} = 1 - \frac{e_p \cdot e_c}{\max(||e_p||_2 \cdot ||e_c||_2, \epsilon)} \tag{4.9}$$

A small ϵ is used to prevent division by zero errors. It is set to $\epsilon = 10^{-8}$ by default.
For tunable hyperparameters λ_1 and λ_2, the resulting composite loss function can then be formulated as:

$$\mathcal{L}_{ae} = \lambda_1(\mathcal{L}_{paper} + \mathcal{L}_{code}) + \lambda_2\mathcal{L}_{sim} \tag{4.10}$$

For a start, we set $\lambda_1 = \lambda_2 = 1$.

4.7 Inference Pipeline

After having trained the model, a design and implementation for a pipeline for inference is needed. This section will go over a proposed design, before its evaluation will be part of the next chapter. For simplicity, the inference part will only cover plugging the data into the model and generating comments as output. Obtaining the data and injecting them into the source code is not part of this pipeline, even though it could be added in the future.
Starting from a pair of research paper and code repository, first, all sentences get extracted from the paper and pre-processed in the same way as described in the previous section about pre-processing. Likewise, all available python files from the repository get processed and their functions (head and body) get extracted. As illustrated in figure 4.8, we calculate all embeddings with SciBERT for all sentences and encode those into the learned shared space with the encoder trained on research papers. Then, for each function, CodeBERT

[17]https://huggingface.co/transformers/model$_{d}oc/bert.html\#transformers.BertModel$

yields the code embeddings and through the encoder part of the auto-encoder trained on code samples, we get the encodings of the code. We calculate their similarities with the cosine similarity function, and output the sentence, whose encoded embeddings have the highest similarity to the encoded function embedding. From there, the resulting sentence could be used as a comment for the respective function in the source repository.

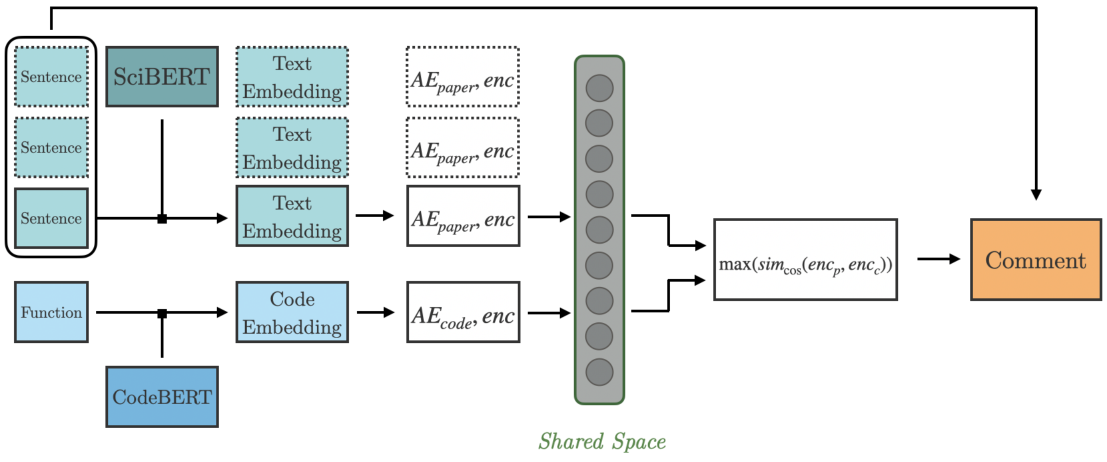


Figure 4.8: Proposed Inference Pipeline

Chapter 5

Evaluation

In this chapter, the proposed training pipelines are to be evaluated and an the inference results for a simple feed-forward model will be compared to human annotation.

5.1 Evaluation Limits

The typical evaluation methods usually done within machine learning to assess a newly developed method can not be realized in this context. As neither a standardized dataset to compare the proposed methods performance against other methods, nor even other methods that try to achieve the same goal exist, evaluation has to be done on a different level. This means, that the method has to be compared to itself on several tasks:

1. Is the shared space learnable?

2. Which factors play a role in learning the space shared by the encoders?

3. How well do (for manually checked examples) generated sentences represent the information in the paper?

5.2 Training Loss

There is an infinite amount of ways to build a neural network. Usually, certain factors play a bigger role in measuring the quality of a model than others, so there will be limitations of what can and will be realistically compared. In each of the upcoming subsections, a potentially influencing factor will be described and discussed.

5.2.1 Baseline Model

As a baseline to compare against, a model without normalization was trained and the loss recorded, which can be seen in figure 5.1. While the cosine similarity loss goes down rapidly, the reconstruction error for both paper and code stays the same throughout the batches.

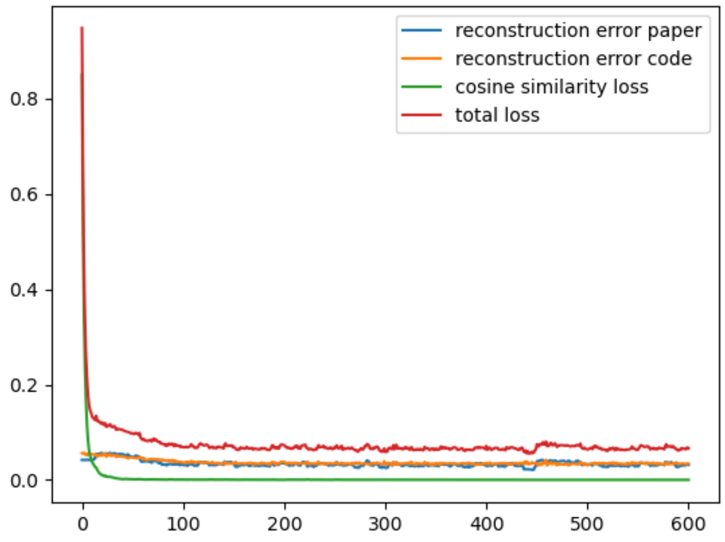


Figure 5.1: Baseline model training loss for n batches.

5.2.2 Effect of Data Properties

Normalization

Normalization brings down both reconstruction errors, as shown in figure 5.2. This leads to an overall lower total loss.

5.2.3 CodeBERT vs. GraphCodeBERT

The way that the pipeline is built, BERT models can be swapped out easily. This section will evaluate the training performance of GraphCodeBERT and compare it to CodeBERT. GraphCodeBERT [21] is a pre-trained model, like CodeBERT, that considers the inherent code structure by taking into consideration the semantic data flow instead of the syntactic AST.

An example of a target task for the model can be seen in figure 5.4. GraphCodeBERT gets used to detect similarity between implementations of function by extracting and comparing their semantic representations. GraphCodeBERT supposedly evaluates higher than CodeBERT on most tasks, meaning a potentially better retention of semantics for this task. Figure 5.5 shows the training loss of the pipeline trained with GraphCodeBERT with pooling of the hidden-state-sequence. Compared to the training performance of 5.3, the difference is marginal.

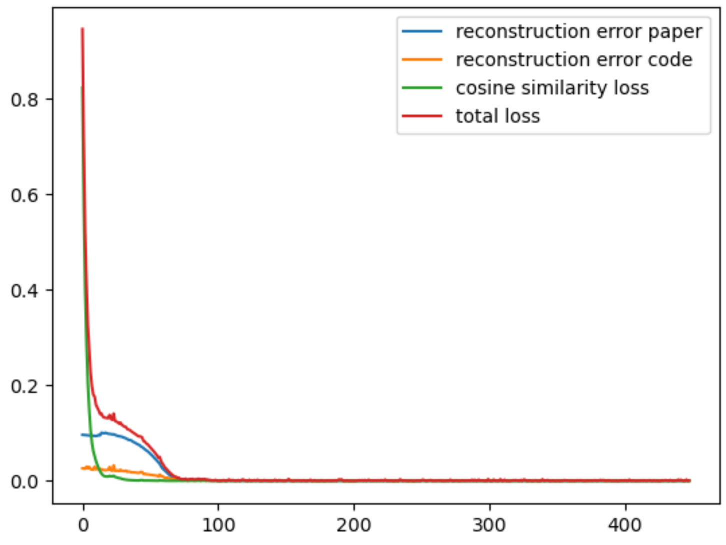


Figure 5.2: Training loss with normalized data for n batches.

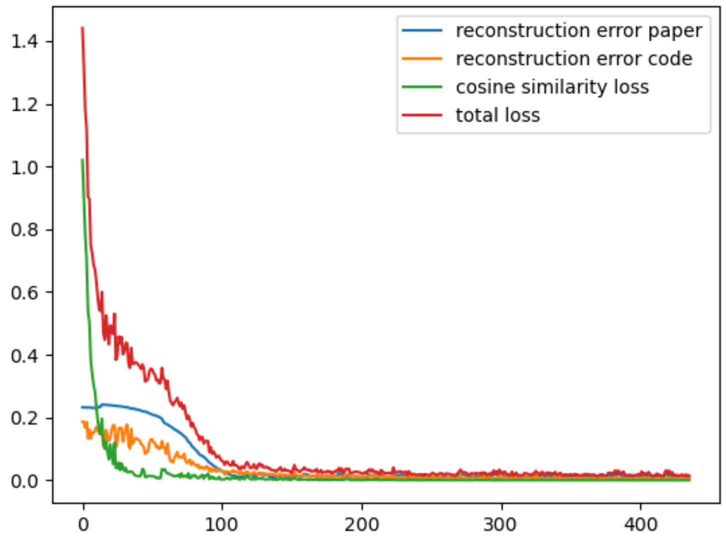
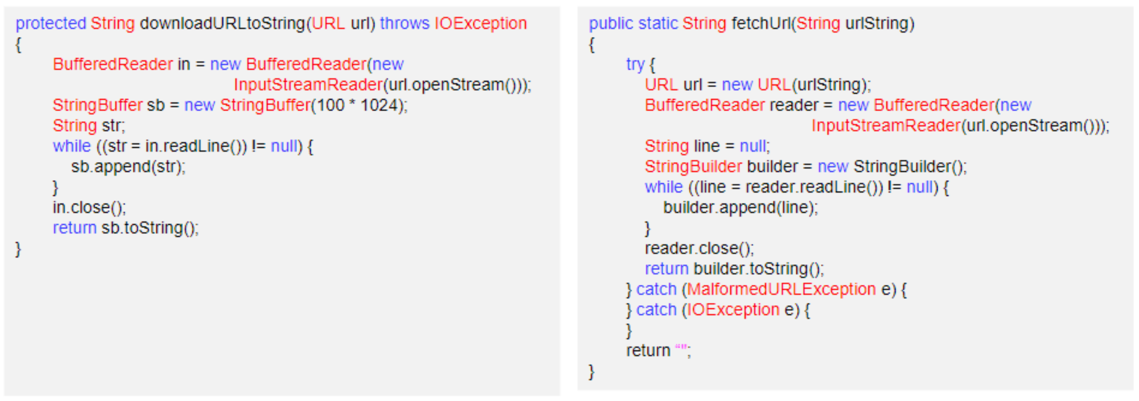


Figure 5.3: CodeBERT training loss for n batches with normalization and average hidden state sequence pooling.



```java
protected String downloadURLtoString(URL url) throws IOException
{
      BufferedReader in = new BufferedReader(new
                              InputStreamReader(url.openStream()));
      StringBuffer sb = new StringBuffer(100 * 1024);
      String str;
      while ((str = in.readLine()) != null) {
         sb.append(str);
      }
      in.close();
      return sb.toString();
}
```

```java
public static String fetchUrl(String urlString)
{
      try {
         URL url = new URL(urlString);
         BufferedReader reader = new BufferedReader(new
                              InputStreamReader(url.openStream()));
         String line = null;
         StringBuilder builder = new StringBuilder();
         while ((line = reader.readLine()) != null) {
            builder.append(line);
         }
         reader.close();
         return builder.toString();
      } catch (MalformedURLException e) {
      } catch (IOException e) {
      }
      return "";
}
```

Figure 5.4: An example of GraphCodeBERT on a clone detection task. The two sources have a reported semantic similarity of 0.983.

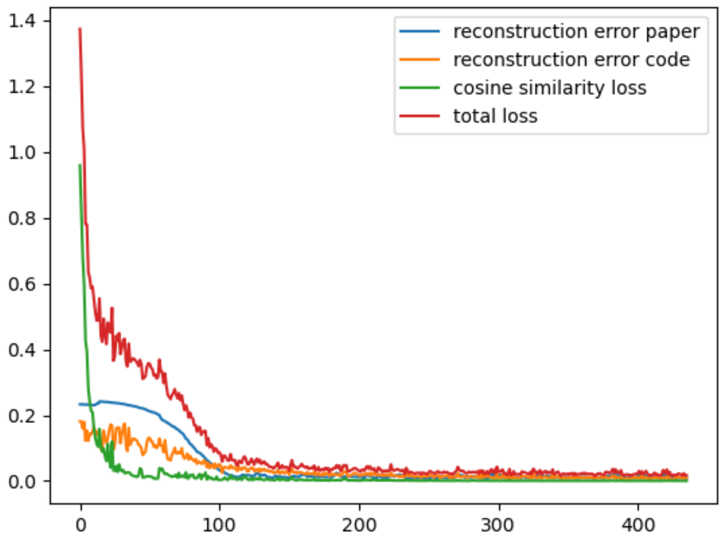


Figure 5.5: Training loss of GraphCodeBERT model for n batches.

5.3 Inference Evaluation

For evaluation of the inference pipeline, a small selected sample of papers will be processed by the pipeline and the extracted comments will be manually compared against how a reader might extract information from the source.

5.3.1 Example #1: Hierarchical Adaptive Contextual Bandits for Resource Constraint based Recommendation

The first chosen research paper is [64] by Yang et al.

Content of the research paper

From the paper's abstract: *"Contextual multi-armed bandit (MAB) achieves cutting-edge performance on a variety of problems. When it comes to real-world scenarios such as recommendation system and online advertising, however, it is essential to consider the resource consumption of exploration. In practice, there is typically non-zero cost associated with executing a recommendation (arm) in the environment, and hence, the policy should be learned with a fixed exploration cost constraint. It is challenging to learn a global optimal policy directly, since it is a NP-hard problem and significantly complicates the exploration and exploitation trade-off of bandit algorithms. Existing approaches focus on solving the problems by adopting the greedy policy which estimates the expected rewards and costs and uses a greedy selection based on each arm's expected reward/cost ratio using historical observation until the exploration resource is exhausted. However, existing methods are hard to extend to infinite time horizon, since the learning process will be terminated when there is no more resource. In this paper, we propose a hierarchical adaptive contextual bandit method (HATCH) to conduct the policy learning of contextual bandits with a budget constraint. HATCH adopts an adaptive method to allocate the exploration resource based on the remaining resource/time and the estimation of reward distribution among different user contexts. In addition, we utilize full of contextual feature information to find the best personalized recommendation. Finally, in order to prove the theoretical guarantee, we present a regret bound analysis and prove that HATCH achieves a regret bound as low as $O(\sqrt{T})$. The experimental results demonstrate the effectiveness and efficiency of the proposed method on both synthetic datasets and the real-world applications."*

Code repository content

The authors provide an implementation to accompany their paper on GitHub[1].
It contains three Python files: *bandit.py*, *util.py* and *evaluation.py*, neither of which have any documentation in form of comments or DocStrings.

Results of the inference

The inference pipeline was not able to generate comments on the full set of sentences without restrictions, despite the autoencoder managed train well with a small loss. Restricting to sentences with matched variables, with normalized CodeBERT embeddings, the following function/comment pairs were generated:

- `__init__`: ""

[1]https://github.com/ymy4323460/HATCH

- `_add_common_lin`: "For HATCH, we keep α as the same in both of resource allocation and personal recommendation level."
- `set_time_budget`: ""
- `fit`: "We set $r = x_t^\top \theta_{j,a}^* + \epsilon$, supposing that ϵ is 1-sub-gaussian independent zero-mean random variable, where $\mathbb{E}[\epsilon] = 0$ and $\theta_{j,a}^*$ denotes the expected value of θ."
- `_fit_single`: "The quality of user class j is captured by u_j and is called expected reward of j."
- `partial_fit`: "We set $r = x_t^\top \theta_{j,a}^* + \epsilon$, supposing that ϵ is 1-sub-gaussian independent zero-mean random variable, where $\mathbb{E}[\epsilon] = 0$ and $\theta_{j,a}^*$ denotes the expected value of θ."
- `predict`: ""

The same results were obtained with the GraphCodeBERT approach. With the training performance being very similar, this could mean that there is no noticeable difference in semantics being transported for this specific task.

Comparison against manual annotation

When comparing the results to the potential comments of manual annotation, there is a noticeable difference. Following the structure of the inference results, a selection of possibly helpful comments could be:

- `__init__`:
  - "We set the dimension of each generated context is $dim = 5$ and the value range of each dimension is in [0,1].
  - "For HATCH, we keep α as the same in both of resource allocation and personal recommendation level."
  - " We set $J = 10$ as the number of user classes and K=10 arms with the reward generated from a normal distribution.
- `_add_common_lin`: "For HATCH, we keep α as the same in both of resource allocation and personal recommendation level."
- `set_time_budget`: ""
- `fit`: "Here we choose Gaussian Mixture Model as mapping method in our system, denoted by $\mathcal{G}(x)$.

- `_fit_single`:

  - "We sampled 30000 synthetic contexts from data generator with the distribution ϕ and we denote X_j as context set of class j."
  - "The quality of user class j is captured by u_j and is called expected reward of j."

- `partial_fit`: "Here we choose Gaussian Mixture Model as mapping method in our system, denoted by $\mathcal{G}(x)$.

- `predict`: "The quality of user class j is captured by u_j and is called expected reward of j."

5.3.2 Example #2: Learning to Remove: Towards Isotropic Pre-trained BERT Embedding

The chosen research paper is [33] by Liang et al.

Content of the research paper

From the abstract: "*Pre-trained language models such as BERT have become a more common choice of natural language processing (NLP) tasks. Research in word representation shows that isotropic embeddings can significantly improve performance on downstream tasks. However, we measure and analyze the geometry of pre-trained BERT embedding and find that it is far from isotropic. We find that the word vectors are not centered around the origin, and the average cosine similarity between two random words is much higher than zero, which indicates that the word vectors are distributed in a narrow cone and deteriorate the representation capacity of word embedding. We propose a simple, and yet effective method to fix this problem: remove several dominant directions of BERT embedding with a set of learnable weights. We train the weights on word similarity tasks and show that processed embedding is more isotropic. Our method is evaluated on three standardized tasks: word similarity, word analogy, and semantic textual similarity. In all tasks, the word embedding processed by our method consistently outperforms the original embedding (with average improvement of 13% on word analogy and 16% on semantic textual similarity) and two baseline methods. Our method is also proven to be more robust to changes of hyperparameter.*"

Repository content

The provided implementation is available on the author's GitHub[2]. The python files in the repository are *config_file.py*, *evaluate.py*, *wr_ train.py*, *all_ test_forBERT.py*, *dataloaders.py*, *draw.py*, *loaddatasets.py*, *models.py* and *testForTask.py.*

Comment Generation Issues

Due to the difference in variable naming between the paper and the code, there were no matches between sentences and functions and therefore no generated comments. This part is included in the evaluation chapter to highlight a problem with this approach. Since the model is not able to infer without constraining the comment output to the matched variables, in cases like this, it does not work.

5.3.3 Discussion

While during training, the loss gives the impression that the shared space was learned by the neural network with the chosen approach and relevant comments can be generated with the mentioned restrictions, the quality and coverage of generated comments gets negatively impacted. The limitation of sequence length being 512 for the models also impacts large function bodies negatively, in that they do not get processed. The rather simple network for inference likely also plays a role in the result. Alternatives and possible improvements to various sections of the whole pipeline will be outlined in chapter 6.

[2]https://github.com/liangyuxin42/weighted-removal

Chapter 6

Conclusion and Outlook

The overall goal of this thesis was to explore how, given scientific publications and their provided implementations, the code could be annotated in a way that would help others understand and re-use the source code for their own needs. Chapter 2 gave an overview over recent developments, the current state of research in natural language processing and theoretical foundations to several concepts, such as embeddings and transformers. Chapter 3 explored specific research approaches for information extraction from research documents and available tools and libraries. The Dr. Inventor framework was described in more detail and an explanation was given on why it was not suited for the proposed task. Chapter 3 also presented relevant research on extraction from source code and tools, before briefly discussion issues with using approaches usually used for translation. A pipeline was drafted and each of its parts discussed in chapter 4. Moreover, chapter 4 went into detail about how a dataset for self-supervised learning approaches can be obtained. Additionally, implementations for pre-processing the data, training the model and using it for inference were presented. Evaluation of the training performance was shown in chapter 5 and the results of the inference was compared to a manually annotated exemplary paper. It showed that an alignment of embeddings generated by two BERT-type models, SciBERT for scientific documents and CodeBERT for code repositories was possible by learning a shared space with an autoencoder. However, the quality of generated comments were not as high as a user would need them to be, or impossible if alignment was different from the chosen scheme, and it seems as if the inference model was not sophisticated enough to be able to *understand* the relations of unlabeled data from the learned representations of the generated dataset, even though using the total set of sentences aligned by their variables as a selection of source code comments could be an option to improve understanding. All in all, this thesis can be understood as a step in the direction, a prototype, of at some point being able to automatically generate semantically rich documentation for scientific source code.

6.1 Outlook

First, in the future, the model and approach for learning the shared encoded space could be substituted. The concept proposed in this thesis could be expanded beyond just variables in the future. Another possibility could be to try different approaches (instead of an autoencoder) to learn the shared space. Also, future works could research different approaches to utilize the dataset proposed or use the approach of this thesis on a different dataset. A different approach to tackle the tackled task could be to generate descriptive natural text from functions and learn to align it with natural text from the papers in order to summarize a text (for comment generation) made from those descriptions. Comments from repositories could also be included in learning approaches, if available. Fine-tuning on manual annotations done by experts could also be a direction to take, if the amount of available labeled data, such as [1], grows over time. Furthermore, it could be possible to try a contrastive learning approach for this problem by training a model to learn differences between positive and negative pairs of code and paper segments, given a good enough retention of semantics through encodings. Fine-tuning the BERT-type models used in the proposed pipelines during training could be done as well. With the likely advancement of research regarding knowledge graph generation in the context of source code, a research opportunity could be to use a similar pipeline as proposed in this thesis to learn an alignment. It could also be possible to train a bimodal BERT model on source code and paper input to skip the problem of having to learn a shared space after generating embeddings.

[1]https://nlp.seas.harvard.edu/2018/04/03/attention.html

Appendix A

Further Information

List of Figures

List of Algorithms

Bibliography

[1] Abrahams, Paul W.: *A final solution to the Dangling else of ALGOL 60 and related languages.* Communications of the ACM, 9(9):679–682, Sep 1966.

[2] Allamanis, Miltiadis, Pankajan Chanthirasegaran, Pushmeet Kohli and Charles Sutton: *Learning Continuous Semantic Representations of Symbolic Expressions*, 2017.

[3] Azanzi, F. J. and G. Camara: *Knowledge Extraction from Source Code Based on Hidden Markov Model: Application to EPICAM.* In *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, pages 1478–1485, 2017.

[4] Bahdanau, Dzmitry, Kyunghyun Cho and Yoshua Bengio: *Neural Machine Translation by Jointly Learning to Align and Translate*, 2016.

[5] Barone, Antonio Valerio Miceli: *Towards cross-lingual distributed representations without parallel text trained with adversarial autoencoders.* arXiv preprint arXiv:1608.02996, 2016.

[6] Bellman, Richard E: *Adaptive control processes: a guided tour.* Princeton university press, 2015.

[7] Beltagy, Iz, Kyle Lo and Arman Cohan: *SciBERT: Pretrained Language Model for Scientific Text.* In *EMNLP*, 2019.

[8] Bengio, Samy, Jason Weston and David Grangier: *Label Embedding Trees for Large Multi-Class Tasks.* In *Neural Information Processing Systems (NIPS)*, 2010.

[9] Chen, Zimin and Martin Monperrus: *A Literature Study of Embeddings on Source Code.* CoRR, abs/1904.03061, 2019.

[10] Cho, Kyunghyun, Bart van Merrienboer, Dzmitry Bahdanau and Yoshua Bengio: *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*, 2014.

[11] Clark, Kevin, Minh-Thang Luong, Quoc V Le and Christopher D Manning: *Electra: Pre-training text encoders as discriminators rather than generators.* arXiv preprint arXiv:2003.10555, 2020.

[12] Devlin, Jacob, Ming-Wei Chang, Kenton Lee and Kristina Toutanova: *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019.

[13] Do, Chuong B and Andrew Y Ng: *Transfer learning for text classification.* Advances in Neural Information Processing Systems, 18:299–306, 2005.

[14] Ehrlinger, Lisa and Wolfram Wöss: *Towards a Definition of Knowledge Graphs.* SEMANTiCS (Posters, Demos, SuCCESS), 48:1–4, 2016.

[15] Felbo, Bjarke, Alan Mislove, Anders Søgaard, Iyad Rahwan and Sune Lehmann: *Using millions of emoji occurrences to learn any-domain representations for detecting sentiment, emotion and sarcasm.* arXiv preprint arXiv:1708.00524, 2017.

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

[17] Fisas, Beatriz, Horacio Saggion and Francesco Ronzano: *On the Discoursive Structure of Computer Graphics Research Papers.* In *Proceedings of The 9th Linguistic Annotation Workshop*, pages 42–51, Denver, Colorado, USA, June 2015. Association for Computational Linguistics.

[18] Garner, Richard: *An abstract view on syntax with sharing*, 2011.

[19] Gogna, Anupriya and Angshul Majumdar: *Semi supervised autoencoder.* In *International Conference on Neural Information Processing*, pages 82–89. Springer, 2016.

[20] Graves, Alex: *Generating Sequences With Recurrent Neural Networks*, 2014.

[21] Guo, Daya, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang and Ming Zhou: *GraphCodeBERT: Pre-training Code Representations with Data Flow*, 2021.

[22] Harris, David and Sarah Harris: *Digital design and computer architecture.* Morgan Kaufmann, 2010.

[23] Hocking, John: *Topology.* Dover Publications, New York, 1988.

[24] Husain, Hamel, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis and Marc Brockschmidt: *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*, 2020.

[25] Hutchins, J.: *Machine translation: a concise history.* 2006.

[26] Jain, Paras, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez and Ion Stoica: *Contrastive Code Representation Learning*, 2020.

[27] Johnson, S. C. and M. Hill: *Lint, a C Program Checker.* 1978.

[28] Jurafsky, Daniel and H James: *Speech and language processing an introduction to natural language processing, computational linguistics, and speech.* 2000.

[29] Kamath, Aishwarya and Rajarshi Das: *A survey on semantic parsing.* arXiv preprint arXiv:1812.00978, 2018.

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

[31] Karampatsis, Rafael Michael and Charles Sutton: *SCELMo: Source Code Embeddings from Language Models*, 2020.

[32] Liang, Yuding and Kenny Q. Zhu: *Automatic Generation of Text Descriptive Comments for Code Blocks*, 2018.

[33] Liang, Yuxin, Rui Cao, Jie Zheng, Jie Ren and Ling Gao: *Learning to Remove: Towards Isotropic Pre-trained BERT Embedding*, 2021.

[34] Liu, Weijie, Peng Zhou, Zhe Zhao, Zhiruo Wang, Qi Ju, Haotang Deng and Ping Wang: *K-BERT: Enabling Language Representation with Knowledge Graph*, 2019.

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

[36] Locke, William N. and A. Donald Booth: *Machine translation of languages.* American Documentation, 7(2):135–136, Apr 1956.

[37] Mikolov, Tomas, Kai Chen, Greg Corrado and Jeffrey Dean: *Efficient Estimation of Word Representations in Vector Space*, 2013.

[38] Mohiuddin, Tasnim and Shafiq Joty: *Unsupervised Word Translation with Adversarial Autoencoder.* Computational Linguistics, 46(2):257–288, June 2020.

[39] Nadeau, David and Satoshi Sekine: *A survey of named entity recognition and classification.* Lingvisticae Investigationes, 30(1):3–26, 2007.

[40] Ng, Andrew et al.: *Sparse autoencoder.* CS294A Lecture notes, 72(2011):1–19, 2011.

[41] Oda, Yusuke, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, T. Toda and S. Nakamura: *Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T).* 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 574–584, 2015.

[42] Peters, Matthew E., Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee and Luke Zettlemoyer: *Deep contextualized word representations*, 2018.

[43] Pfahler, Lukas and Katharina Morik: *Semantic Search in Millions of Equations.* In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, KDD '20, page 135–143, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Pires, Telmo, Eva Schlinger and Dan Garrette: *How multilingual is Multilingual BERT?*, 2019.

[45] Poplawski, David A: *Properties of LL-Regular Languages.* 1977.

[46] Prabhavalkar, Rohit, Kanishka Rao, Tara Sainath, Bo Li, Leif Johnson and Navdeep Jaitly: *A Comparison of Sequence-to-Sequence Models for Speech Recognition.* 2017.

[47] Qiu, Xipeng, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai and Xuanjing Huang: *Pre-trained Models for Natural Language Processing: A Survey*, 2020.

[48] Radford, Alec, Karthik Narasimhan, Tim Salimans and Ilya Sutskever: *Improving language understanding by generative pre-training.* 2018.

[49] Raffel, Colin, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li and Peter J. Liu: *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*, 2020.

[50] Ronzano, Francesco and Horacio Saggion: *Dr. inventor framework: Extracting structured information from scientific publications.* In *International conference on discovery science*, pages 209–220. Springer, 2015.

[51] Ronzano, Francesco and Horacio Saggion: *Knowledge Extraction and Modeling from Scientific Publications.* 2016.

[52] Rumelhart, D., Geoffrey E. Hinton and R. J. Williams: *Learning representations by back-propagating errors.* Nature, 323:533–536, 1986.

[53] Schmidhuber, Jürgen: *Deep learning in neural networks: An overview.* Neural networks, 61:85–117, 2015.

[54] Shahid, Abdul, M. Afzal, M. Abdar, Mohammad Ehsan Basiri, X. Zhou, N. Y. Yen and Jiawei Chang: *Insights into relevant knowledge extraction techniques: a comprehensive review.* The Journal of Supercomputing, 76:1695–1733, 2019.

[55] Shapiro, Stuart C: *Encyclopedia of artificial intelligence second edition.* John, 1992.

[56] Stathopoulos, Yiannos, Simon Baker, Marek Rei and Simone Teufel: *Variable Typing: Assigning Meaning to Variables in Mathematical Text.* In *NAACL-HLT*, pages 303–312, 2018.

[57] Sutskever, Ilya, Oriol Vinyals and Quoc V. Le: *Sequence to Sequence Learning with Neural Networks*, 2014.

[58] Upadhyay, R. and A. Fujii: *Semantic knowledge extraction from research documents.* 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), pages 439–445, 2016.

[59] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin: *Attention Is All You Need*, 2017.

[60] Wan, Yao, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu and Philip S Yu: *Improving automatic source code summarization via deep reinforcement learning.* In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407, 2018.

[61] Wang, Alex, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy and Samuel R. Bowman: *GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding*, 2019.

[62] Wong, Edmund, J. Yang and Lin Tan: *AutoComment: Mining question and answer sites for automatic comment generation.* 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 562–567, 2013.

[63] Wu, Di, Liang Ding, Shuo Yang and Dacheng Tao: *SLUA: A Super Lightweight Unsupervised Word Alignment Model via Cross-Lingual Contrastive Learning*, 2021.

[64] Yang, Mengyue, Qingyang Li, Zhiwei Qin and Jieping Ye: *Hierarchical Adaptive Contextual Bandits for Resource Constraint based Recommendation.* Proceedings of The Web Conference 2020, Apr 2020.

[65] Zhang, Ce: *DeepDive: a data management system for automatic knowledge base construction.* University of Wisconsin-Madison, Madison, Wisconsin, 2015.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den April 19, 2021

[illegible]

Pierre Haritz

Eidesstattliche Versicherung
(Affidavit)

Haritz, Pierre	159535
Name, Vorname (Last name, first name)	Matrikelnr. (Enrollment number)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem folgenden Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present Bachelor's/Master's* thesis with the following title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution.

Titel der ~~Bachelor~~-/Masterarbeit*:
(Title of the ~~Bachelor's~~/ Master's* thesis):

Enriching Scientific Source Code by means of Mining Relations between Papers and their Implementation

*Nichtzutreffendes bitte streichen
(Please choose the appropriate)

Dortmund, 19.4.2021	[illegible]
Ort, Datum (Place, date)	Unterschrift (Signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to €50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, section 63, subsection 5 of the North Rhine-Westphalia Higher Education Act (*Hochschulgesetz*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:**

Dortmund, 19.4.2021	[illegible]
Ort, Datum (Place, date)	Unterschrift (Signature)

**Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.