Sofia University, “St. Kliment Ohridski”
Faculty of Mathematics and Informatics
Computer Informatics

Source Code Identifier Suggestion

MASTER’S THESIS
IN
ARTIFICIAL INTELLIGENCE

Author: Anton A. Petkov, Faculty number: 25915

Supervisor: Assoc. Prof. Trifon A. Trifonov, PhD

July 2020

Table of Contents

1. Abstract

The amount of freely accessible open-source code is growing every day. Millions of man-hours go into the development of open-source projects that follow best practices for quality code.

When programming, naming abstractions with identifiers is one of the most common tasks. We hypothesize that the rules for constructing identifiers are hidden implicitly in the vast amounts of source code and can be learned and applied through the use of machine learning techniques.

In this thesis, we focus on the method name suggestion problem for the programming language Java. We create a dataset with Java methods from a curated list of well-maintained Java repositories. We use this dataset to train a deep learning language model that generates method names for a given source code snippet. After experimentation, the best model achieves a ROUGE-L F1 score of 0.5955 on the test set. We also create an IDE extension for real-world application that uses the model to suggest method names and summarize code in Java.

2. Introduction

In computer programming, source code has two complementary roles: it is written for machines to execute, but at the same time for people to read. Software is created and maintained by groups of people who express their ideas for solving a problem through the means of programming languages. In this sense, source code can be seen as a form of human communication.

Identifiers are an important component of any program, as they serve as names of program elements such as variables, functions, classes, objects, and more. As a rule, programmers carefully select identifiers so that they reflect the meaning of these elements as clearly as possible so that the code is easy to understand. Inventing and selecting appropriate identifiers is a creative and non-trivial process. Having a tool that helps in this process has the potential to increase the productivity of programmers.

Creating such an effective tool through symbolic methods (for example, using computational linguistics and formal grammars) is not easy due to the complex structure of identifiers. They are compound and often ambiguous phrases, which is why generating them by enumeration and exhaustive searching for appropriate identifiers leads to a combinatorial explosion. Furthermore, it is not clear whether such an approach could work well with unfamiliar source code.

On the other hand, machine learning methods can approximate an optimal solution to a problem without creating this combinatorial explosion and have the ability to generalize when working with previously unseen data, provided that an appropriate sample of data is used for training.

Over the last decade, significant progress has been made in the field of natural language processing through the application of deep learning. Natural language processing is in general a more complex task than processing of programming languages, which are formal languages. This is mostly due to the more complex grammar and semantics of natural languages, and mostly because of their inherent ambiguity. Despite this complexity, a number of language models exist which can outperform people in some linguistic tasks. [1] [2] Examples of such language models include BERT [3], GPT-2 [4], XLNet [5], among others.

In the last few years the number of open-source repositories and the number of people who support them in publicly accessible systems such as GitHub has increased significantly. Such code is freely available in huge quantities (often referred to as "Big Code") and some of it is used by many projects because of its good quality. Millions of lines of code contain implicit rules for writing quality code and, in particular, rules for creating and assigning appropriate identifiers. This data can be used to train probabilistic models to learn and approximate these rules based on a snippet of source code.

2.1. Thesis Structure

This subsection explains how the following sections in this thesis are organized. Chapter 3 defines the problem addressed in this thesis and provides a general direction for the solution. Chapter 4 reviews prior work in the field. Chapter 5 presents and analyses the dataset used for the language model. Chapter 6 explains the deep learning language model's structure, how it works and how it can be configured. Chapter 7 focuses on evaluation, so that the best model can be trained through the experiments carried out in the following Chapter 8, which also visualizes what the model has learned. Chapter 9 describes the developed IDE extension for suggestions of Java method names. Chapter 10 covers the implementation details of this thesis. Finally, Chapter 11 ends with a list of research contributions and ideas for future work.

3. Problem Statement

The main goal of this thesis is to develop a tool for suggesting appropriate identifiers in source code. The tool is intended to be utilized as an extension of an integrated development environment (IDE) where it can access existing source code and be queried for identifiers at specific locations in the code. Thus, it receives as input source code and outputs identifiers, ranked by estimated relevance.

3.1. Constraints

A number of constraints may be introduced to reduce the problem complexity.

Firstly, in the age of Big Data, there are millions of lines of open-source code available online, i.e., Big Code. There is a subset of quality open-source projects in GitHub which are widely used and well-maintained. They contain implicit knowledge, like rules for constructing appropriate identifiers to name program components. However, this knowledge not directly accessible and should somehow be extracted and made use of.

This abundance of data implies that a Machine Learning model can be developed to solve the problem. It can be used to automatically learn these rules for identifier construction from the already written and curated source code from the previous constraint. Machine Learning methods can build models that can generalize the relationship between identifiers and the source code they refer to and thus become applicable to previously unseen examples. Because of this, when the model suggests identifiers for a piece of code and the code is subsequently changed, the model will suggest a different set of identifiers. In particular, Deep Learning can build complex models that have a greater capacity to learn such rules.

This thesis is focused specifically on the suggestion of method names as a special case of identifier suggestion. This makes the problem easier to solve and the solution would require fewer resources (training data and computational power). This constraint can also be seen as reducing the problem to source code summarization.

We have selected the Java programming language as the target for the tool for method name suggestion. The choice of Java is due to its vast popularity and availability of many mature open-source projects written in Java. Additionally, it is a strongly typed language and typing information can be very useful for feature engineering.

To summarize, the problem has been reduced to developing a Machine Learning model for method name suggestion, trained on a large corpus of open-source Java code.

3.2. Examples

Given the Java method from Code Snippet 1 as input, the tool should suggest a list of appropriate identifiers for the method `f`, ranked by relevance.

```java
boolean f(Set<String> set, String value) {
    for (String entry : set) {
        if (entry.equalsIgnoreCase(value)) {
            return true;
        }
    }
    return false;
}
```

Code Snippet 1. Java method

A potential list of appropriate identifier suggestions may be:

1. containsIgnoreCase
2. hasIgnoreCase
3. checkForIgnoreCase

4. Related Work

4.1. Survey of Machine Learning on Source Code

The survey paper "A Survey of Machine Learning for Big Code and Naturalness" by M. Allamanis et al. [6] is a good entry point in the field of applying machine learning on source code. It classifies what existing problems there are in the field and it also classifies the types of models used for solving these problems.

One of the most interesting ideas in the paper is the Code Naturalness hypothesis that states that "software is a form of human communication". This is due to the fact that source code consists mainly of identifiers that carry the meaning of the code, so that it is easy for humans to understand. Research shows that up to 70% of all characters of a project's source code can be identifiers. [7]. This hypothesis implies that these natural language properties of source code can be exploited through the application of Natural Language Processing models, as it is done in this thesis.

Most notably, the survey paper presents a thorough analysis of the differences between source code and natural language:

- Source code is **executable** and small changes to it can lead to drastic changes in its behavior, while natural language is not that sensitive.

- Source code has much more **neologisms** compared to natural language. This is due to the way identifiers are constructed – they are compound phrases, concatenated using camelCasing, snake_casing, etc. This implies that a direct application of NLP models may not be ideal, instead more complex tokenization methods have to be considered, e.g. splitting identifiers like *binarySearchTree* into the sub-words *binary*, *search* and *tree*. The high number of neologisms also implies that identifiers should be generated as sequences, as opposed to simple classification of single tokens.

- Source code is written so that it can be **reused**. Programmers often apply the DRY (Don't Repeat Yourself) principle, so that the code can be as maintainable as possible. In comparison, natural language has a lot of repetitive words, often called "stop words", e.g. "the", "an", "of", etc. This means that stop word removal does not seem so appropriate for source code, compared to natural language.

- In order to be understood by the reader, source code can require more **background knowledge** compared to natural language. This is partly due to the reusability of source code. Programming languages provide tools for creating and composing abstractions. This can imply that, in order to learn properties of source code identifiers, the identifier's definition would not be sufficient, but instead all its usages should be considered as well. An example of background knowledge can be a boolean flag that is initialized to *true* and then used for taking a decision, but if it is updated in a loop, then the semantics of the code become more complex.

4.2. Extreme Code Summarization

Extreme Code Summarization is one of the problems in the field closely related to identifier suggestion, in particular method name suggestion. This is because the method name often summarizes the meaning of the method's body.

At the time of writing this thesis, code2seq [8] has shown impressive results in method name suggestions through the use of the popular Neural Machine Translation Seq2Seq architecture [9]. This paper was very influential for this thesis, however their dataset did not seem appropriate because the training set and test sets do not contain methods from the same projects, meaning that the distributions of the two sets may not be the same. This issue is addressed in this thesis by creating a new dataset, as explained in Section 5.

The code2seq model adds special AST (Abstract Syntax Tree) Path Embeddings that encode the token's path in the AST from the root of the tree. This is done so that the model can extract more semantic information about the source code.

"Suggesting Accurate Method and Class Names" by M. Allamanis et al. [10] is also a successful paper in the field because the developed model not only learns to suggest identifiers, but it also embeds source code identifiers into a vector space that has interesting properties. For example, identifiers for array operations are clustered together. This model in this paper accounts for the background knowledge property of the code (discussed in Section 4.1) by encoding all occurrences of an identifier into context vectors and aggregating them, then using them as features.

Finally, "A Convolutional Attention Network for Extreme Summarization of Source Code" by M. Allamanis et al. [11] is also a successful research paper in the field. The model they develop uses convolutional attention, i.e. it combines characteristics from Convolutional Neural Networks [12] and the Attention Mechanism [9].

5. Data

In the software engineering community, quality software code is developed and maintained by following coding conventions and rules. One of the hypotheses that this thesis is based on is that these conventions and rules are implicitly available in source code and can be automatically learned by a machine learning model. In order to develop such a machine learning model, it is necessary to build a sufficiently large data set that represents quality source code.

The number of publicly accessible source code repositories has grown rapidly in the recent years. GitHub and other source code hosting systems provide access to numerous open-source projects. It is estimated that millions of man-hours have been spent by the open-source community to provide high-quality software. For example, the effort gone into Apache Hadoop can be estimated up to 1.8 million of man-hours [13]. This freely available source code can be leveraged to create the data set for this problem.

Supervised learning problems require a dataset with labeled examples. However, in many cases, these examples have to be labeled by a human which is a very labor-intensive task. Fortunately, for the method name suggestion problem, we have access to millions of lines of code and the method name (i.e., the label) can be automatically extracted alongside the method body. Other useful properties can be extracted from the method signature as well, such as modifiers (e.g. public, private, static, etc.), annotations, parameters, return type, name of containing class, etc.

The next sections focus on how this data set is built.

5.1. Data Collection

The most popular Java source code repositories are considered to build a data set that reflects how method names should be constructed. GitHub exposes a REST API that allows searching for repositories by several criteria (e.g. most stargazers[1]) and retrieving metadata for these repositories, such as contributors count, commits count, etc.

The dataset is comprised of 29 Java repositories, which have been selected based on a number of properties:

- Number of stargazers – reflects the number of people “liking” the repository
- Number of contributors – only repositories with at least 50 contributors are considered
- Number of commits – only repositories with at least 500 commits are considered
- Number of forks – how many users have duplicated and modified the repository on their own
- The domain of application – the 29 repositories have been carefully selected to cover as many domains as possible

Repository	**Hash**	**Domain**
antlr/antlr4	0b35a76e	Language Parser Generator
apache/commons-io	cd778727	Core Libraries for IO
apache/commons-lang	115a6c64	Core Libraries
apache/dubbo	da6c3ebf	RPC
apache/flink	674817c1	Stream Processing
apache/groovy	57a48870	Programming Language
apache/hadoop	3ca15292	Distributed Processing
apache/kafka	d4ef46c6	Message Queue
bazelbuild/bazel	1af65cf2	Build System
bigbluebutton/bigbluebutton	18bb82f7	Web Conferencing
brettwooldridge/HikariCP	8a28f6eb	Database Connection Pool
clojure/clojure	30a36cbe	Programming Language
dbeaver/dbeaver	2aa7f923	Database Tool
elastic/elasticsearch	853dd1b8	Full-Text Search Engine
google/ExoPlayer	6bfdf8f8	Android Media Player
google/gson	ceae88bd	JSON Library
google/guava	2b5c096d	Core Libraries
jenkinsci/jenkins	2776f4dc	Automation Server
junit-team/junit4	50a285d3	Unit Testing
libgdx/libgdx	95b054d8	Game Development
mockito/mockito	b0e814b0	Mocking Framework
neo4j/neo4j	1a9468b0	Graph Database
netty/netty	b559711f	Network Applications
oracle/graal	cf16c076	Virtual Machine

[1] A stargazer is a user who has given a “star” to the repository, meaning that the user likes the repository. The most popular repositories on GitHub have the most stargazers.

ReactiveX/RxJava	da498c5c	Asynchronous Programming
scribejava/scribejava	fdef16c0	OAuth Library
SeleniumHQ/selenium	3912f49c	Browser Automation
spring-projects/spring-framework	a34f1e37	Web Applications
zxing/zxing	64ff8012	Barcode Scanning Library

Table 5.1. The list of all 29 Java source code repositories that comprise the final data set.

The final selection of repositories can be seen in Table 1. The second column contains the short commit hash representing a unique identifier of the specific snapshot of the repository that is used (ensuring reproducibility of the dataset). The third column is the application domain of the selected project.

Table 2 contains statistics for the repositories in the data sets. Most of the repositories have been maintained for almost a decade and all of them are being actively maintained and used in many projects.

Repository	**Commits**	**Contributors**	**Stars**	**Forks**	**Year Created**
antlr/antlr4	7448	194	7848	1950	2010
apache/commons-io	2358	57	696	455	2009
apache/commons-lang	5734	139	1930	1177	2009
apache/dubbo	4327	302	32568	21218	2012
apache/flink	22428	676	13177	7111	2014
apache/groovy	17202	302	3700	1450	2015
apache/hadoop	23913	253	10510	6506	2014
apache/kafka	7656	341	15950	8465	2011
bazelbuild/bazel	26335	568	14799	2552	2014
bigbluebutton/bigbluebutton	26942	111	4815	4677	2010
brettwooldridge/HikariCP	2728	97	13096	2038	2013
clojure/clojure	3298	143	8399	1305	2010
dbeaver/dbeaver	16159	131	13717	1248	2015
elastic/elasticsearch	52804	356	49249	16898	2010
google/ExoPlayer	7946	157	15660	4697	2014
google/gson	1485	103	17960	3494	2015
google/guava	5274	237	37594	8440	2014
jenkinsci/jenkins	30023	613	15623	6260	2010
junit-team/junit4	2432	143	7878	2987	2009
libgdx/libgdx	14161	344	17013	6036	2012
mockito/mockito	5260	189	10604	1889	2012
neo4j/neo4j	68479	188	7817	1839	2012

netty/netty	9880	359	23748	11346	2010
oracle/graal	44950	174	12670	897	2016
ReactiveX/RxJava	5750	253	42828	7165	2013
scribejava/scribejava	1145	103	5044	1657	2010
SeleniumHQ/selenium	25269	390	17826	5726	2013
spring-projects/spring-framework	20933	391	37510	25286	2010
zxing/zxing	3535	98	25491	8693	2011

Table 5.2. Statistics of the 29 Java repositories from the data set.

5.2. Feature Extraction

Once the Java source code files have been collected, they are fed into the **javalang** Java parser [14] which extracts the methods alongside their properties. These properties can be very useful for feature engineering which can improve the model's learning performance. Below is a list of the extracted features:

- Method body – e.g. `{ return stream.getSourceName(); }`
- Return type – e.g. `void, int, LinkedHashMap, String`, etc.
- Method parameters with their types – a list of tuples [parameter name, parameter type]. For example, `[['grammarName', 'String'], ['result', 'Collection']]`
- Annotations – a list of the method-level annotations, e.g. `[Override, ExportMessage]`
- Modifiers – e.g. `public, private, static`, etc. However, methods with `abstract` and `native` modifiers are ignored, since they don't have an implementation (a body).
- Documentation of the method – a string, e.g. `'/**\n   * A predicate that returns true for {@link SkyKey}s that have the given {@link SkyFunctionName}.\n   */'`
- Name of the class which contains the method, e.g. `SimpleCycleDetector, AbstractParallelEvaluator`

5.3. Data Analysis

The raw dataset consists of 791 986 Java method names that are extracted from 126 483 Java files that contain a total of 22 977 834 lines of code.

The preprocessing of the collected data strongly affects the hyperparameters of the machine learning model which will use this data. In this thesis, the Deep Learning Seq2Seq architecture [15] is used. It is composed of an encoder and decoder Recurrent Neural Network (RNN) [16] which work on sequences of tokens. The encoder encodes the Java method body and the decoder decodes the method name. The two most important hyperparameters affecting the data preprocessing step are the vocabulary size and the maximum sequence length for both the input and output sequences. The Seq2Seq model is explained in Chapter 6.

5.3.1. Maximum Sequence Length

The longer the sequences, the more hardware resources are required to train the model. The simplest (vanilla) RNNs are not capable of modeling long-term dependencies in long sequences. This flaw is partly fixed with more complex RNNs like the LSTM (Long-Short Term Memory) [16] which maintain a memory state. Thus, if the maximum sequence length is too long, the model will take longer to train, and if it is too short, the model might not be able to learn well enough.

Tokenization is one of the most important steps when preprocessing text sequences because it determines the maximum sequence length. For the method name suggestion problem, three tokenization approaches are considered for the input sequence. They are listed below with an example of how the sequence "`HTTPClient client = new ExternalClient();`" is tokenized:

1. Token-level

   This tokenizer produces shorter sequences at the cost of a larger vocabulary. The tokenizer is essentially splitting words by whitespace and non-alphanumeric characters (e.g. braces, operators, etc.).

Example: `{ HTTPClient, client, =, new, ExternalClient, (, ), ; }`

2. Character-level

   Treating each character as a token is the simplest way of tokenizing and it also produces the smallest possible vocabulary. However, this comes at the price of a very long maximum sequence length.

Example: `{ H, T, T, P, C, l, i, e, n, t, _, c, l, i, e, n, t, _, =, _, n, e, w, _, E, x, t, e, r, n, a, l, C, l, i, e, n, t, (, ), ; }`

3. Subtoken-level

   This is the most complex tokenizer of the three because it has to account for different styles of identifier capitalization, such as camel-casing (e.g. traverseTreePreOrder), snake-casing (e.g. EXAMPLE_GLOBAL_VARIABLE). It represents a compromise between vocabulary size and sequence lengths.

Example: `{ HTTP, Client, client, =, new, External, Client, (, ), ; }`

The method names which are the labels in the dataset are compound phrases that are concatenated using camel-casing and they are split into subtokens. For example, `buildHTTPResponse` is tokenized into `{ build, HTTP, Response }`.

The effect of these tokenization methods on the sequence lengths of the input sequences is illustrated on Figure 5.1. When sequence lengths are sorted in ascending order, the histogram of all sequence lengths approximates an exponential distribution. Generally, there are more samples with short sequences and less samples with long sequences. Token and subtoken level tokenization methods produce shorter sequences of lengths up to 1000, while character level tokenizers can produce sequences of lengths up to 4000 which is very long and hard for some models to learn from.

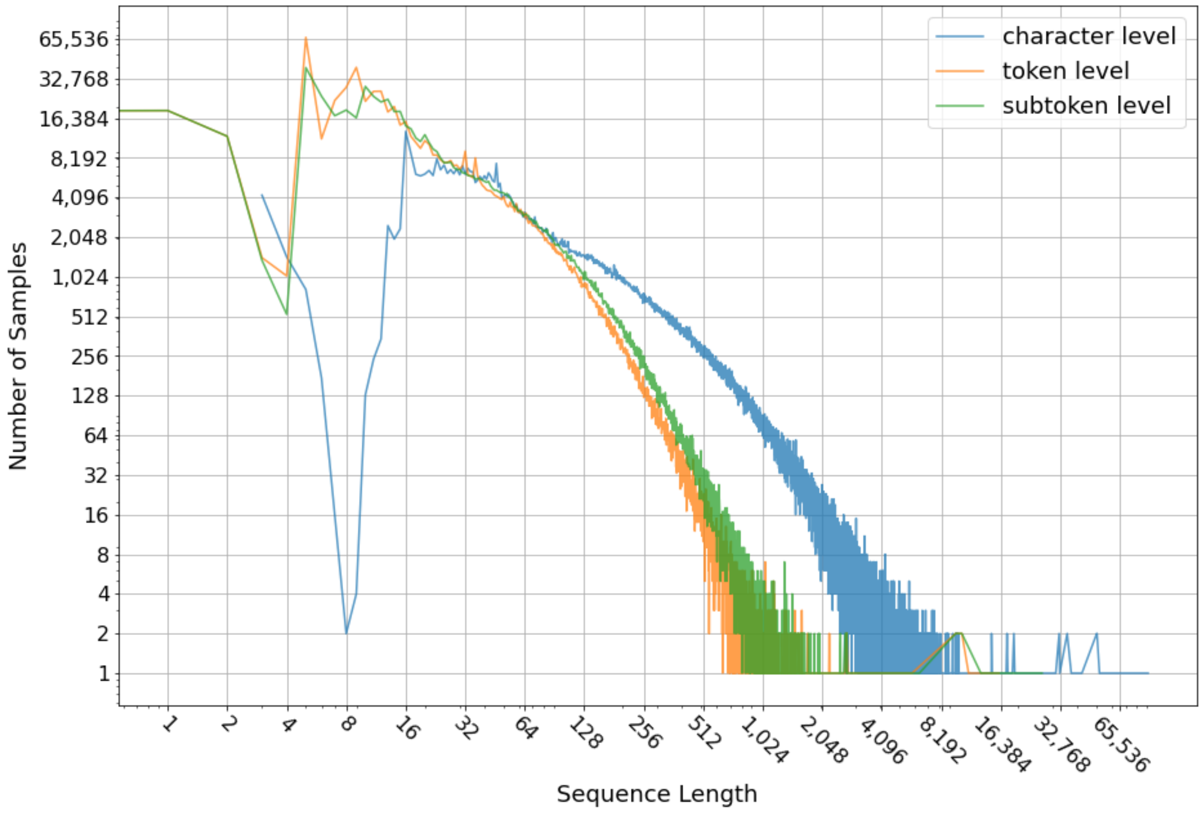


Figure 5.1. Logarithmic histogram plot of the sequence lengths for all tokenization methods.

The maximum sequence length is an important hyperparameter for sequence models. Figure 5.2 illustrates what proportion of the sequences would be used in their full length when a different maximum sequence length threshold is chosen. From the figure it can be seen that 90% of the token and subtoken level sequences can fit in a maximum sequence length of 192 and 90% of the character level sequences can fit in a maximum sequence length of 768. The latter is expected because, as it was seen in Figure 5.1, the character level sequences are roughly 4 times larger than the token level sequences.

Sequences which are too large to fit in the maximum sequence length are trimmed to their prefix.

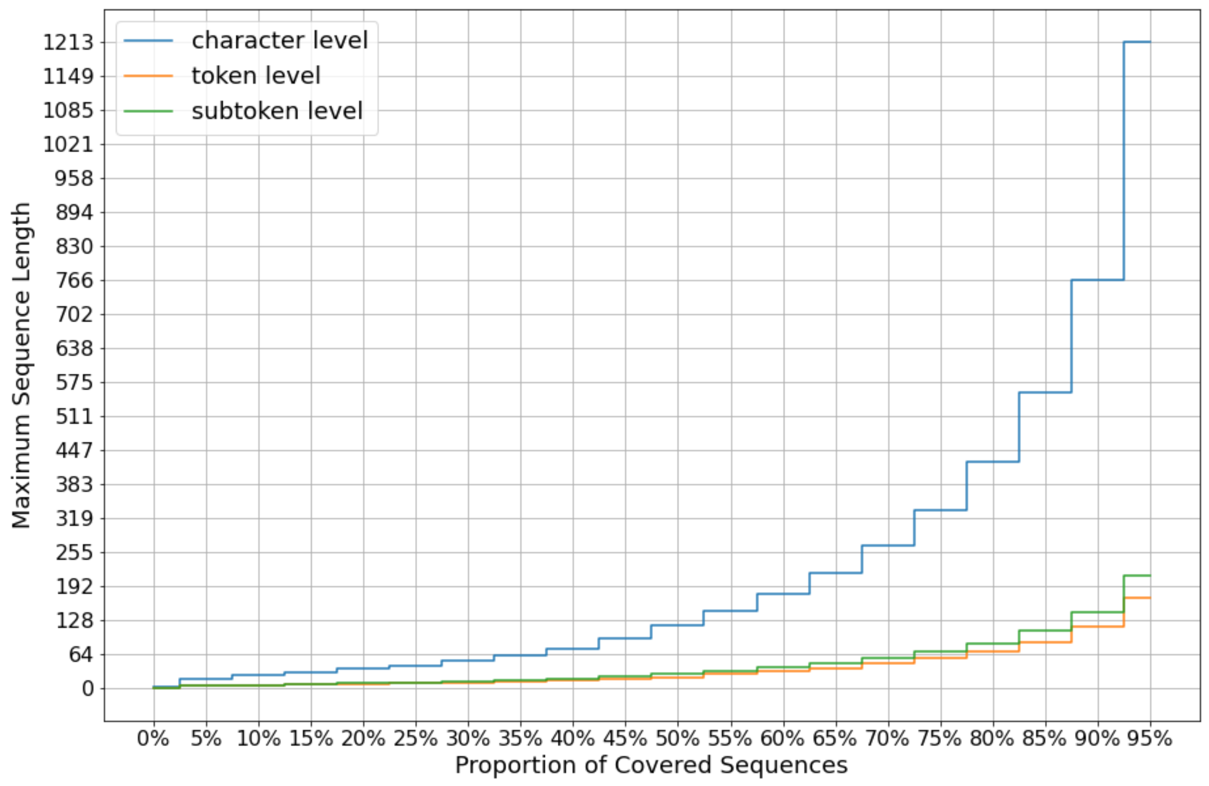


Figure 5.2. Plot of the proportion of sequences which will be used with their complete length for different maximum sequence length thresholds.

Figure 5.3 illustrates the difference between token-level and subtoken-level tokenization of the input sequences even better. In order to cover 95% of the sequences a maximum sequence length of 170 is sufficient at the token level compared to 210 at the subtoken level, which is a small increase when considering the massively reduced vocabulary size at the subtoken level.

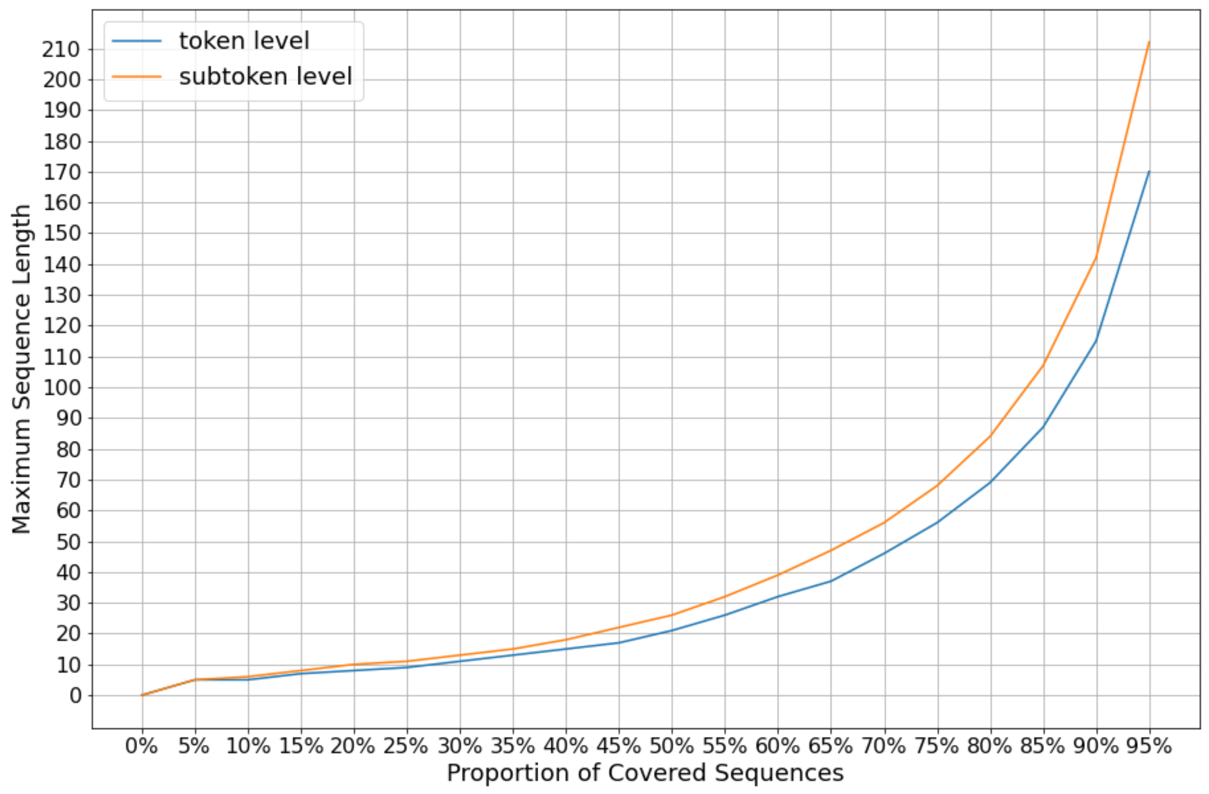


Figure 5.3. Comparison of subtoken vs token level tokenization of the input sequences.

Method Names of Unit Tests are Longer

Methods from testing files, e.g. unit tests, have longer names on average compared to methods from non-testing files. The approximate average number of subtokens in the method name of a testing methods is 4.07 which is much higher compared to an average length of 2.48 subtokens in non-testing methods.

This makes sense because many unit testing methods have very long method names that label a specific test scenario. For example:

- testVerifyVorbisHeaderCapturePatternInvalidHeaderQuite
- testLeastOfIterable_simple_n_withNullElement

If testing methods are excluded from the dataset, then the maximum sequence length for the method names becomes shorter with 2 subtokens. 70% of the sequences can be covered with a maximum sequence length of just 3 subtokens if testing methods were excluded, as Figure 5.4 illustrates.

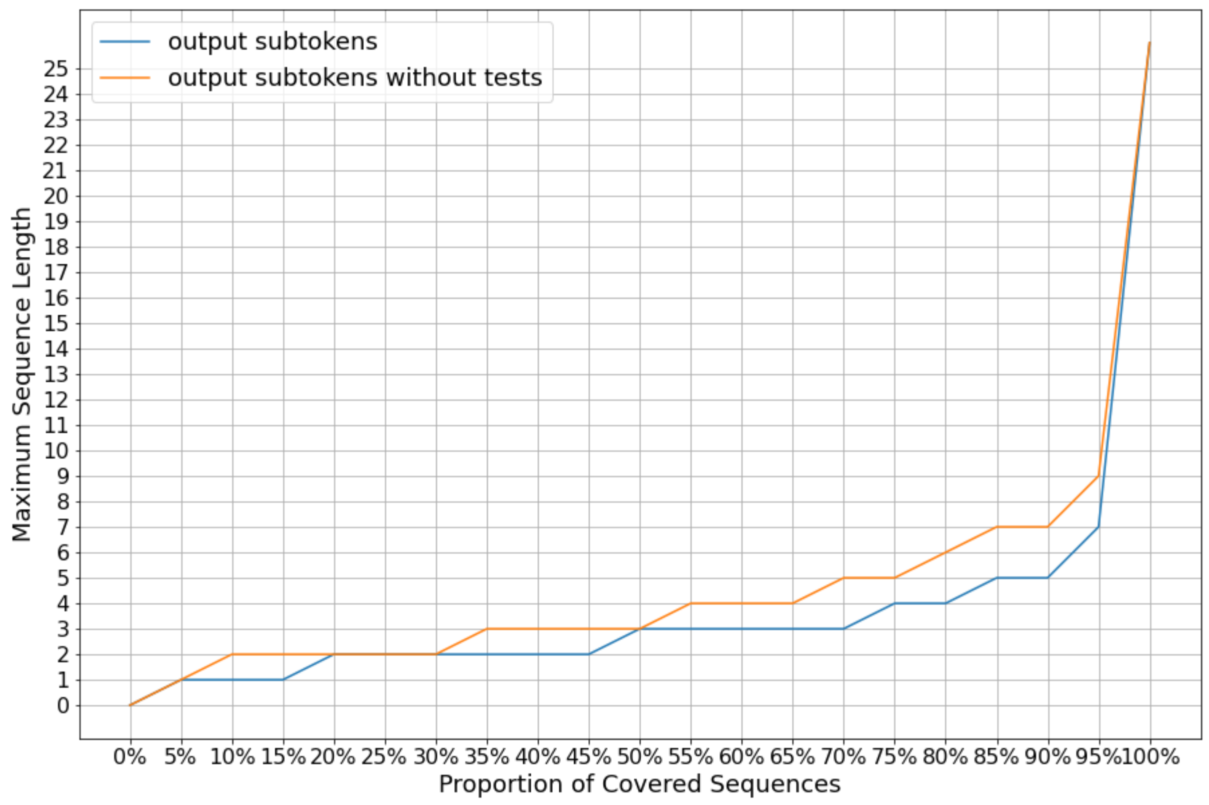


Figure 5.4. Plot of the proportion of covered output sequences when limited to various maximum sequence lengths.

5.3.2. Vocabulary Size

As with the maximum sequence length, the bigger the vocabulary size, the more hardware resources will be required to train the model. There are a few preprocessing methods that can be applied to reduce the vocabulary size.

- Masking Literals

String and number literals comprise a big part of the source code in the Java method bodies. Most of these literals are unique inside the dataset which makes them of no use to the model. By masking string literals with a special "<STR>" literal and masking numbers with a special "<NUM>" literal, we can reduce the size of the vocabulary drastically.

- Tokenization

As it was stated in Section 5.3.1, the tokenization method determines the size of the vocabulary. There is a trade-off between the maximum sequence length and vocabulary size. At the character level, the vocabulary is the smallest and most of the vocabulary size reduction techniques become obsolete because the vocabulary size is already small enough.

- Lower-casing

Programmers use naming conventions to make the code more readable for humans. In Java, camel-casing is used. By definition, a camel case phrase is constructed by concatenating multiple phrases where each of these phrases begins with a capital case and there are no intervening spaces or punctuation in between. Thus, it is very possible for a word like "**client**" to occur both with lower and upper case because there can be identifiers like "**clientRequest**" and "**HTTPClient**".

The vocabulary size can be reduced by lower-casing all tokens. For example, the token "association" would occur 85 times, but it would also occur 185 times with a capital leading character and 8 times with all character capital. If the tokens in the dataset with subtoken tokenization are lower-cased, then the vocabulary size is reduced by more than 40% (from 41 504 to 24 265).

- Stemming

Stemming is the process of reducing words to their stem. For example, the words "fetch", "fetched", "fetching", "fetchable" can be reduced to the stem "fetch". Stemming can reduce vocabulary size drastically, but at the cost of losing expressive power. For example, "fetch" is a verb while "fetchable" is an adjective and can carry a different meaning than the verb, depending on the source code surrounding it.

- Lemmatization

Lemmatization is similar to stemming, but it is more complex. It reduces all words to their lemma, i.e. their canonical form. This reduces the vocabulary size even further compared to stemming and is also more time-consuming. For example, "go", "going" and "went" will be reduced to just "go".

- Stop word removal

Usually, when learning from natural language stop words are removed. However, when learning from source code, special tokens like opening and closing braces and operators (e.g. +, =, -, etc.) carry important meaning (braces signify nesting), and removing them can reduce the ability of the model to learn.

Stemming and lemmatization are very popular in Natural Language Processing, but they aren't applied to the dataset in this thesis due to time constraints.

Figure 5.5 illustrates the effect of the various tokenization methods on the vocabulary sizes. The y-axis of the plot represents vocabulary coverage, which denotes what proportion of all atoms (where an atom can be a character, subtoken or token depending on the tokenization method) in the whole dataset would be included in the limited vocabulary. The character level tokenizer produces the most compact vocabulary which contains only 1266 characters. This is more than the symbols in the ASCII table because the vocabulary contains UTF-8 symbols. Limiting it to the 64 most frequent characters would cover 98.39% of the vocabulary.

The token level tokenizer produces an enormous vocabulary of a total size of 446 727 tokens. There are so many unique symbols in this vocabulary because of the way identifiers are constructed. Gluing together multiple phrases produces identifiers like getDataFormat, setDataFormat, etc. that contain redundant subtokens.

This is where the subtoken level tokenizer makes a big difference. It produces more than 10 times smaller vocabulary compared to the token level tokenizer. A limited vocabulary size of 2048 would include more than 97.16% of the most frequent subtokens which is very efficient compared to the token level vocabulary of the same size which would cover 87.73%.

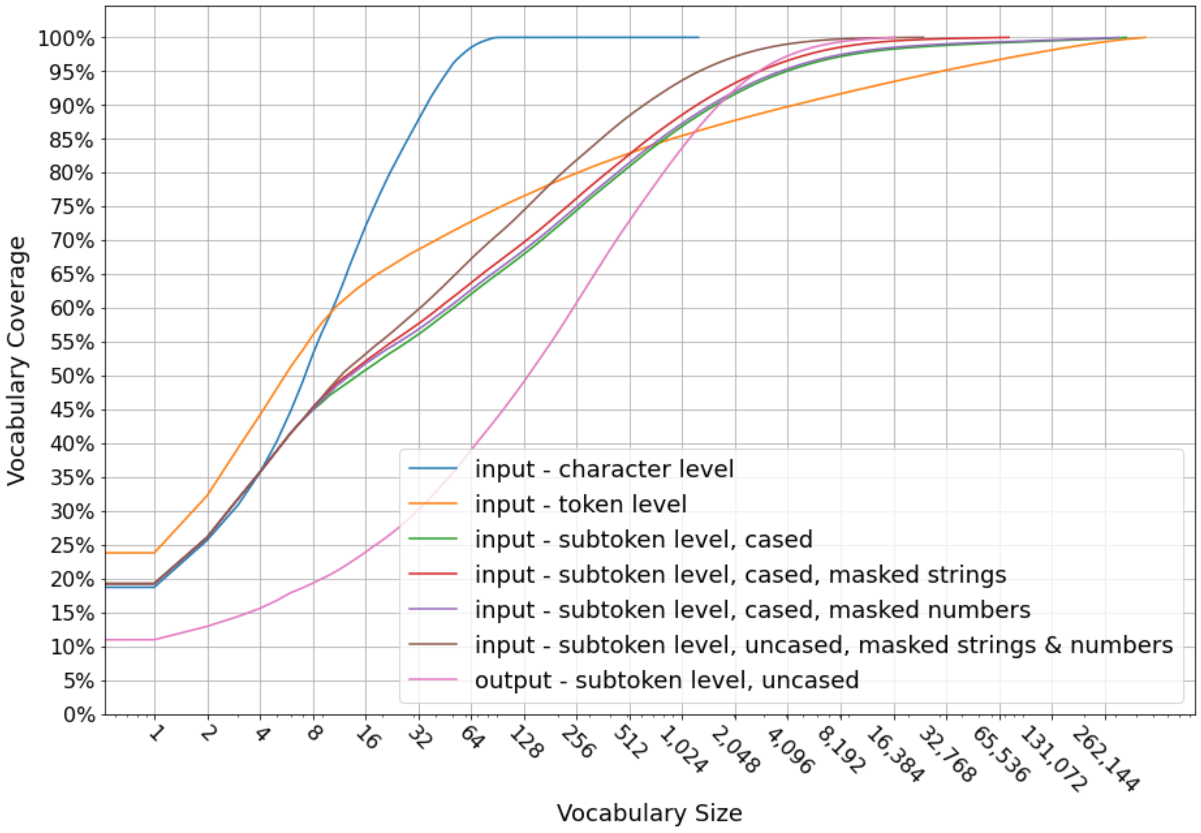


Figure 5.5. A plot illustrating the portion of the vocabulary that would be modeled at different vocabulary sizes for all tokenization methods. The vocabulary coverage on the y-axis represents the total proportion of all symbols in the vocabulary.

The total vocabulary sizes can be observed in Table 5.3. Subtoken level tokenization alone reduces token level vocabulary size with 22.29%. String masking and number masking reduce the subtoken level vocabulary even more by 78.50% and 9.54%, respectively. Lower-casing reduces another 41.56% on top of the previous preprocessing techniques, leading to a compact final vocabulary size of 24265.

The output vocabulary is tokenized at the subtoken level and also lower-cased. Its size is 16742 which is smaller compared to the size of the smallest input vocabulary, but it generally has a smaller coverage at smaller sizes. This is due to the fact that the input sequences contain special symbols like braces and operators that compose most of the source code.

Sequence	**Sequence Preprocessing Method**	**Vocabulary Size**
Input	Character Level	1266
Input	Token Level	446727
Input	Subtoken Level	347141
Input	Subtoken Level + String Masking	74616
Input	Subtoken Level + Number Masking	314029
Input	Subtoken Level + String Masking + Number Masking	41504
Input	Subtoken Level + String Masking + Number Masking + Lower-casing	24265
Output	Subtoken Level + Lower-casing	16742

Table 5.3. Vocabulary sizes produced by the different sequence preprocessing methods.

Input and Output Vocabulary Overlap

The input and output vocabularies share common subtokens. Figure 5.6 illustrates the portion of overlap between the two vocabularies if they were limited to the same size including only the most frequent tokens. When limited from 2000 to 7000 tokens, the vocabularies have a high overlap of more than 75% and up to 80%. The high vocabulary overlap can potentially make it easier for the model to use subtokens from the input sequence when generating a method name.

The reason the vocabulary overlap is not higher than 80% is that the input sequences contain tokens that cannot occur in method names, such as semicolons, operators, braces, etc.

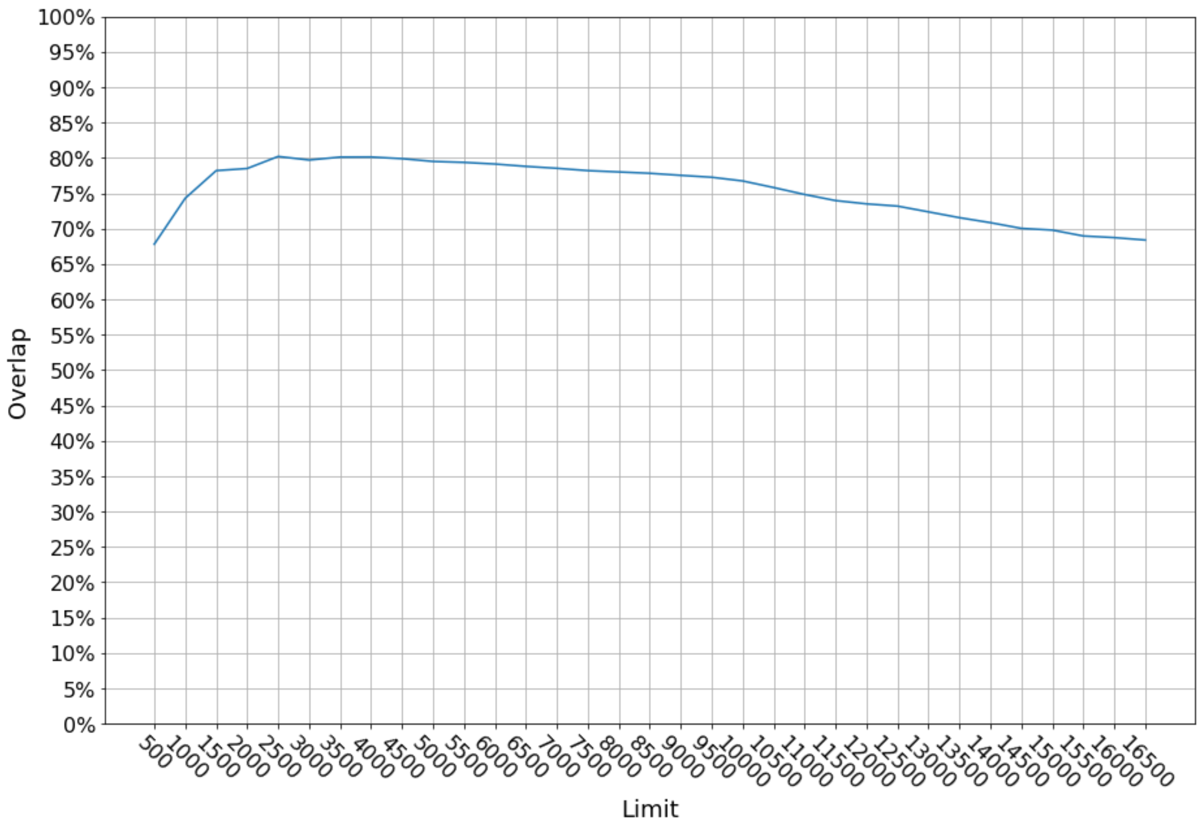


Figure 5.6. Plot of the vocabulary overlap.

5.3.3. Data Noise

The dataset used for training the model can contain noise which can inhibit its prediction performance. For example, there may be samples in the dataset that do not represent what a good method name is. Or there can be too many samples of a certain class, e.g. too many methods from unit tests, which skews the data distribution so that the model learns only this specific class of samples.

- Tests

Unit tests and other types of tests (e.g. integration tests) represent a very large part of the data set. By convention, they are placed in a directory named **test**/. Unit tests have a very repetitive structure composed of assertions and possibly some mocking of objects. In the current data set 36.5% of the 792 034 methods are from test files. However, some of these methods are utility methods that do not follow the repetitive structure of a test and can improve the model's performance.

- Methods annotated with @Override

Java is an object-oriented programming language that supports inheritance. When a class inherits from another class or interface it can override the definition of a method. The new definition of the overridden method is annotated with the @Override annotation, but the name of the method remains the same. This means that there can be multiple implementations of the same method with the same method name.

In general, having more examples of the same method name can be seen as beneficial for model training purposes. However, there are exceptions, for example, the popular Runnable.run method can have multiple and entirely unrelated implementations. 240 650 of the methods are annotated with @Override which makes up for 30.38% of the whole dataset.

Further data analysis can identify more potential outliers and data noise. For example, there are methods composed of method invocations that receive a very large amount of string parameters. However, if the strings are masked as they are in our final dataset, then there can be more <string> mask tokens than any other token and the model will not be able to learn anything useful from such an example.

Also, the distributions of all data subsets (e.g. training, validation, test) must match. This implies that each subset should have a proportionate amount of methods from each of the 29 source code repositories in the current dataset. Alternatively, for example, if the 29 repositories were partitioned into training and test sets without any overlap, then the model would learn to suggest method names in the domains of databases and web development, but would be evaluated on game development and image processing, leading to very bad performance.

5.4. Data Preparation

Finally, the methods in each of the 29 repositories are randomly split into training, validation, and testing subsets with a ratio of 80 / 10 / 10. Then the 29 training sets are combined into a single training set and shuffled, with the same procedure being performed for the validation and the test sets. This approach preserves the distribution of the data between the 29 repositories in all three types of data sets.

The test set is used to evaluate the final performance of the trained model. The validation set is used for the evaluation of a trained model only during hyperparameter optimization. It is important not to use the test set itself during hyperparameter optimization because it can bias the selection of hyperparameters.

Each machine learning model has its own requirements for how the data is represented. Consequently, each model has its own data preprocessing step. For example, all deep learning models work with numeric data, so the tokens have to be encoded into integers before they are fed into the neural networks. This preprocessing step is described in Section 6 below.

6. Model

Method name suggestion is a supervised learning problem. Once the data set is developed and the input features and output labels are defined, then the model training can take place.

The most complex feature of the data set that carries most of the meaning of the method name is undoubtedly the method body. This is due to the fact that the method name should reflect what the method is actually doing. Since the method body is a sequence of tokens, sequence models are a good fit. The method body is a special type of sequence, as it is written in a programming language. Thus, it is reasonable to assume that a language model would be a good fit.

Standard classifiers like SVM, Multi-class Logistic Regression, Feed Forward Neural Network, etc, are not directly applicable, because they don't work with sequential data. Method names are compound phrases that can be comprised of up to 10 sub-phrases. If there are 5000 possible sub-phrases, then there are $5000^{10} \approx 9.8 \times 10^{36}$ possible labels, which makes these classifiers impractical and too expensive computationally.

Method Name Suggestion can be reduced to the Extreme Summarization [17] problem because the method name is a compound phrase, summarizing the method's meaning. It can also be reduced to the Machine Translation problem because the input is a sequence and the output is also a sequence, but the input and output vocabularies differ. In this way the model will be able to generate a sequence of tokens that can be concatenated into a camel-cased method name.

The NLP community has been widely using n-gram probabilistic language models for Statistical Machine Translation [18], but recently Neural Machine Translation with Deep Learning has the best performance. [9] Numerous neural language models have been developed in the last decade. The most prominent such models are based on Recurrent Neural Networks (RNN) and/or the Attention mechanism, like the Seq2Seq (i.e., sequence to sequence) and Transformer architectures [19].

Deep learning models can generalize for complex problems like the problem considered in this thesis. If the model generates a method name for a given method body and that source code is slightly updated, then a different method name will be generated, reflecting the new meaning of the method. Neural networks can automatically learn features of the data, which avoids the feature engineering step. However, deeper models require more data and more training time than classical machine learning models. The number of parameters that have to be learned for a neural language model can be millions and some state-of-the-art language models even have billions of parameters.

In this thesis, the Seq2Seq architecture [9] is used and Section 6.2 is dedicated to it.

6.1. Intermediate Representation of Tokens

The first important aspect of the model is how it is going to represent the input data. Neural Networks work with numbers, but the method bodies are sequences of discrete tokens.

One approach to representing the tokens with numbers is to use the popular one-hot encoding for the set of tokens. Each token from the vocabulary V can be one-hot encoded into a vector of size |V| such that all numbers in the vector are zero, except the number at the index that matches the position of the token in the vocabulary. For example, if V = { ‘a’, ‘b’, ‘c’, ‘d’ }, then ‘c’ would be encoded as [0, 0, 1, 0]. However, this encoding is fixed and very sparse, it does not model any of the meaning the tokens carry. Tokens like ‘get’ and ‘fetch’ that have a similar meaning are not close to each other because all vectors are orthogonal to each other.

There are other less-sparse encoding strategies like TF-IDF (Term Frequency - Inverse Document Frequency) which is very popular in the NLP community. It is counting-based and tokens that are rare within a document, but frequent among documents, have a larger weight. However, this encoding is still fixed and very sparse.

Another problem which both one-hot encoding and TF-IDF share is that the dimensions of the encoded vectors equals the vocabulary size which is in the thousands for the method name suggestion problem. A dimensionality reduction algorithm can be applied to reduce the dimensions of the produced vectors. It can make the vectors denser, but this still isn’t an optimal way to represent the tokens.

6.1.1. Token Embedding

In NLP, distributed vector representations of words, also known as word embeddings, have been used successfully for feature learning in language modeling. [20] Embeddings consist of a function that maps a token from the vocabulary into a fixed-length vector in a continuous vector space of a smaller dimension than the size of the vocabulary. This map is the first layer in the neural network. The embedding layer is essentially a matrix of size $|V| \times$ ***latent dimension*** where the latter is the dimension of the latent vector space within which tokens are embedded.

The embedding layer has several benefits. Firstly, the encoded vectors are of a smaller dimension and denser. Secondly, that layer is learned by the neural network end to end along with all other parameters of the model while optimizing for the downstream task that is being solved. This means that the weights of the embedding layer adapt to the problem the network solves. As a consequence, the learned vector space can model the meaning of the tokens in the vocabulary. Tokens which have similar meaning would be positioned closer to each other in the vector space. Thus, tokens like ‘get’ and ‘fetch’ would be closer to each other. This property of embeddings is discussed again in Section 8.3.

The downside is that the number of parameters that have to be optimized grows linearly with a larger vocabulary. However, the embedding matrix from another Embedding layer can be reused inside a neural network for a different problem. This is a normal practice in the NLP community where pre-trained embeddings like word2vec [20], GloVe [21], and fastText [22] are used for downstream tasks. This way the embedding layer weights are frozen and they don’t need to be optimized.

A reasonable hypothesis is that such pre-trained embeddings can be applied for the method name suggestion problem. However, such an experiment is outside the scope of this thesis, the primary problem being that there is a mismatch between the spoken English vocabulary and the source code vocabulary which is full of programming terminology.

Embeddings are an efficient way to encode discrete data into a continuous representation. Token Embeddings and any other type of Embeddings can be combined (e.g. concatenated) with any other vectors to form features inside a neural network.

6.2. Architecture

The Seq2Seq neural architecture has proven to be very efficient for learning from sequential data as well as for generating sequential data. It had breakthrough success in NLP [9], which is why it was selected for the method name suggestion problem in this thesis.

It is composed of two neural networks – an encoder and a decoder. The encoder processes the input sequence time-step by time-step (or token by token when working with text). The encoder produces a thought vector, i.e., an internal representation that encodes the meaning of the input sequence. This internal representation is then fed into the decoder network which generates the output sequence token by token. This process is illustrated in Figure 6.1.

The Seq2Seq model heavily relies on Recurrent Neural Networks which are covered in greater depth in the next subsections which explain the architecture.

This model architecture does not require manual feature engineering to be performed because, like many other sequential Deep Learning models, it creates its own features through the use of an encoder Recurrent Neural Network (RNN).

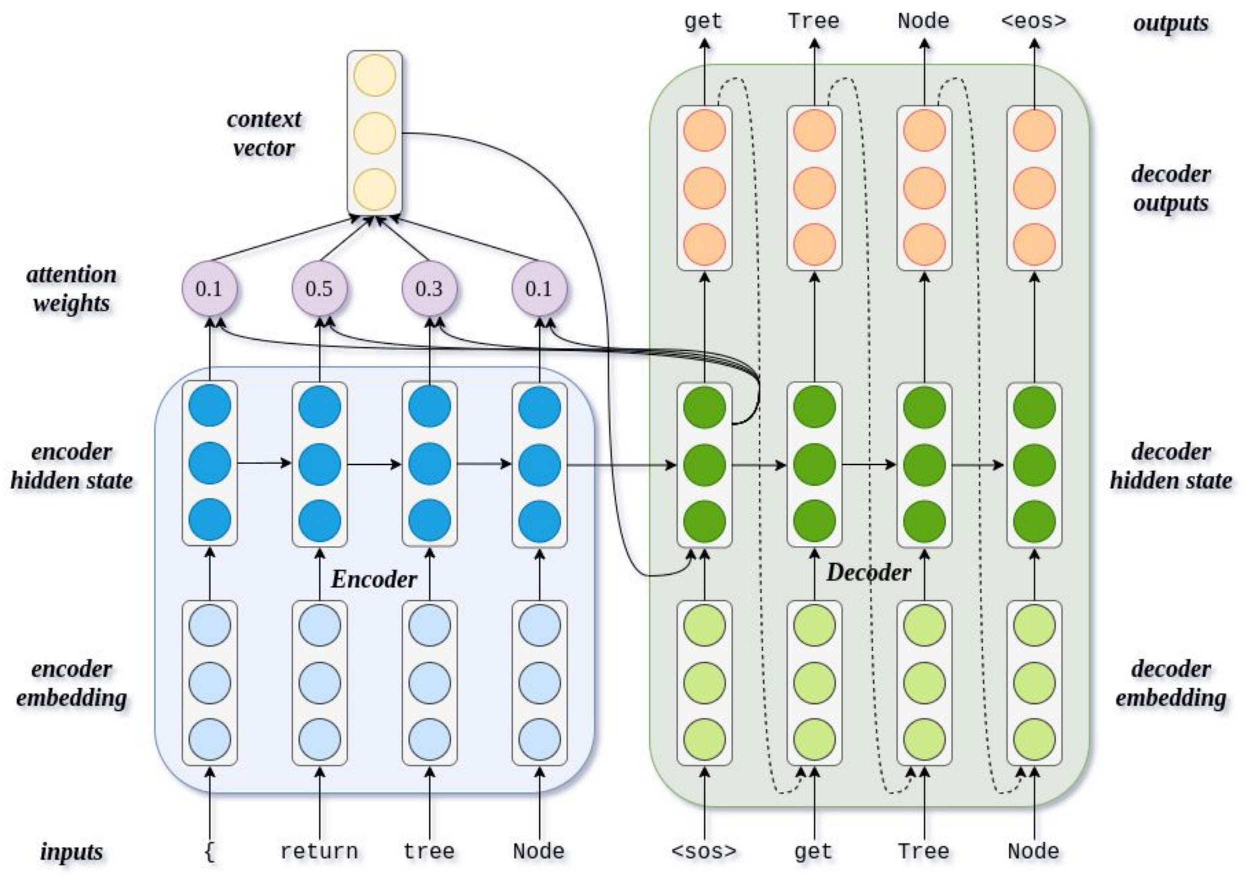


Figure 6.1. Illustration of the Seq2Seq architecture.

6.2.1. Encoder

To understand the Seq2Seq architecture, consider the simple input sequence "{ return treeNode; }" (it is short for simplicity) in Figure 6.1. It gets fed into the the Encoder network subtoken by subtoken. Each subtoken is first encoded into an integer, as explained in Section 6.3, and then it is encoded into an embedding vector by the encoder embedding layer.

After all inputs are embedded, they are fed one by one to the Encoder RNN that maintains a hidden state at step N that is passed as input to the next step N + 1. This way the network can "remember" useful features of the inputs. The output of the Encoder is the last hidden state of its RNN, i.e., the thought vector that represents the input features extracted by the Encoder. The Encoder RNN also produces an output vector for each input, all of which are used by the Attention Mechanism as explained below.

6.2.2. Decoder

The decoding process in the Decoder, illustrated in the right side of Figure 6.1, is started by feeding in as input the special <sos> (start of sequence) marker and initializing the Decoder

RNN hidden state with the thought vector that the Encoder produced in the encoding phase. In the vanilla Seq2Seq architecture this hidden state is directly passed through the output layer to predict the first subtoken in the method name, but in the architecture in Figure 6.1 the Attention Mechanism [9] is used.

The Attention Mechanism, illustrated in the upper left corner of Figure 6.1, helps the decoder to focus on (attend to) the most important subtokens in the input sequence with regard to the next subtoken in the output sequence. This is done by calculating attention weights for each of the Encoder RNN outputs with regard to the current hidden state of the Decoder RNN. These attention weights are then used to produce a context vector that is concatenated with that same Decoder input embedding vector and this new vector is fed into the Decoder RNN.

Finally, the Decoder RNN output is fed to the last output dense layer that, using the Softmax activation function (Formula 6.2), produces a probability distribution over the output vocabulary. The output subtoken is sampled from this distribution and is fed back into the Decoder, as illustrated by the dashed lines in the Decoder in Figure 6.1, so that the next subtoken in the output sequence gets generated and so on. Section 6.5 explains the inference process and the different strategies for sampling the output probability distribution.

This decoding process is repeated until the <eos> (end of sequence) marker is reached. Then, all generated output subtokens are concatenated (in camelCase) to produce the predicted method name. This is the standard decoding process the Decoder follows. However, during training the decoding is different because the generated output at step N is not fed back into the Decoder at step N+1, but instead the reference subtoken is fed. This training process, known as Teacher Forcing, is explained in Section 6.4.

It is important to note that Figure 6.1 illustrates how the attention weights and context vector are calculated only for the first decoder step. This same process is repeated for each decoder step. This means that at step 2, the attention weights will be calculated for all Encoder outputs with regards to the Decoder RNN hidden state at step 2. Then, the context vector is concatenated with the Decoder embedding vector at step 2.

6.2.3. LSTM Recurrent Neural Network

The LSTM (Long Short-Term Memory) [16] is a special kind of RNN (Recurrent Neural Network) because it controls its memory through **input**, **output** and **forget** gates. Each of these 3 gates is represented by a separate feedforward neural network and provide a way to optionally let information through.

As their names suggest, the input gate decides what new input information flows in the LSTM unit, the forget gate chooses what information should be forgotten and the output gate selects what information goes out of the LSTM unit.

Training vanilla RNNs on long sequences can lead to vanishing or exploding gradients, which means that the weights of the network might respectively stop updating or change too much – both of which hindering the learning process. It has been shown that the LSTM deals with these problems and has better performance.

In this thesis, the TensorFlow reference implementation of LSTM is used for the Encoder and Decoder.

6.2.4. Attention Mechanism

The Attention Mechanism can learn which inputs from the input sequence are most important for the prediction of the next output. It has many implementations, but in this thesis Bahdanau Attention [9] is used, shown in Formula 6.1.

$$score = V \cdot tanh(W_1 \cdot H_{decoder} + W_2 \cdot H_{encoder})$$

Formula 6.1. Bahdanau Attention score.

The score is a linear combination of the decoder hidden state $H_{decoder}$ and the encoder outputs $H_{encoder}$ that is passed through the *tanh* function and then multiplied by a third weight matrix V. The weights W_1, W_2 and V are learned by the model throughout the training process.

This score is then fed into the Softmax function from Formula 6.2 that produces a probability distribution over the encoder outputs, i.e., the attention weights that have higher values for the most important parts of the input sequence.

$$a_{ij} = \frac{\exp(s_{ij})}{\sum_{k=1}^{T_x} \exp(s_{ik})}$$

Formula 6.2. Softmax function in the context of Bahdanau Attention scores. T_x is the length of the encoder inputs over which attention is distributed.

Finally, the attention weights are multiplied with $H_{encoder}$ to adjust how important each of the encoder inputs is, resulting in the context vector. This context vector is then concatenated with the decoder embedding and then fed into the Decoder LSTM.

6.3. Data Preprocessing

The token sequences from the dataset have to be encoded into numbers before they are fed into the neural networks. The tokenizer assigns a unique number to each token in the vocabulary. However, the vocabulary size is limited and this maximum size is a hyperparameter of the model. The number for a token is determined based on the number of times it occurs in the dataset. The most frequent tokens have smaller numbers. Thus, when the vocabulary size is limited to N, only the most frequent N tokens are encoded and used. Because the vocabularies of the input and output sequences differ, there are two corresponding hyperparameters as well as two tokenizers.

The sequences are limited in size by the maximum sequence length hyperparameter. All sequences must be of the same size when fed into the model. If we denote the maximum sequence length as M, then there are many sequences with a length larger or smaller than M. In order to align all sequences to be with the same lengths, a zero padding suffix is used for shorter sequences and suffix truncation is used for longer sequences. Because the input sequences are very long (they represent source code) and the output sequences are very short (i.e., method names), there are two different maximum sequence length hyperparameters.

The output sequences must start with a special <sos> "start of sequence" marker and end with an <eos> "end of sequence" marker. This way the decoder knows it has to start predicting the output sequence when fed the <sos> token. When the <eos> token is reached, the decoder stops predicting tokens for the output sequence. The sequence markers must remain after the padding step.

There are two more special tokens. The <oov> token marks that the token is "out of vocabulary". It replaces any tokens which are not present in the limited vocabulary. The padding is represented by the <pad> token which is always encoded as the number zero.

For example, here is how the *getTreeNode* output sequence gets preprocessed in 4 steps:

1. Tokenization is applied:
   get, tree, node
2. The sequence is wrapped with the special start and end markers:
   <sos>, *get, tree, node,* ***<eos>***
3. Padding is added:
   <sos>, get, tree, node, <eos>, ***<pad>, <pad>, <pad>***
4. Text tokens are encoded into numbers:
   1, 3, 78, 15, 2, 0, 0, 0

Ideally, a neural network can be trained on all training samples for a single step. However, the large amounts of data cannot fit in the operational memory. Training on a single training sample at a time is very time-consuming. Mini-batch training helps the model train faster than the latter approach but within reasonable memory limits. Thus, it is important to find the right balance between memory and time consumption. The data must be aligned into batches of equal length. The batch size can be different when training and when evaluating. It is also a good practice to feed the training data in a random order in every epoch so that the model does not overfit because of the fixed order of the training data.

6.4. Optimization

While Stochastic Gradient Descent (SGD) is one of the most popular and widely used optimizers, in this paper Adam [23] is chosen. Adam has shown better performance on attention models than SGD [24]. It has an adaptive learning rate and it has learning rates per parameter instead of a global learning rate as in SGD.

6.4.1. Loss Function

The loss function which the optimizer aims to minimize is the Categorical Cross Entropy Loss (CCE for short). It is often applied when the Softmax activation function (see Formula 6.2) is used on the final dense linear layer of the neural network. The Softmax function produces a probability distribution over the output classes which in our case are the vocabulary tokens. The Categorical Cross Entropy Loss measures the difference between two probability distributions.

At each timestep the decoder network produces a probability distribution over the output vocabulary. This predicted distribution is fed into the Categorical Cross Entropy Loss along with the true distribution which has a probability of 1 for the true token (the token from the golden label in the training data set) and a probability of 0 for the remaining tokens in the vocabulary. The loss function is defined as follows:

$$CCE(T, P) = -\sum_{i=1}^{|V|} t_i \log(p_i)$$

Formula 6.3. The Category Cross Entropy loss function.

The Category Cross Entropy loss function from Formula 6.3 takes two probability distributions: the true probability distribution T and the predicted probability distribution P. Both T and P are distributions over the vocabulary V which contains |V| tokens (labels). The whole sum is negated, because $\log(p_i)$ is negative when $p \in [0, 1]$, so that the produced loss value is non-negative.

At every training step, the loss is averaged along both axes – batch size and sequence length.

6.4.2. Teacher Forcing

During training, the decoding process is more special. Instead of feeding the predicted output on step N as input for step N+1, the ground truth output of step N is fed as input to the Decoder at step N+1. This is the Teacher Forcing [25] algorithm.

For example, if at decoding step 1 on Figure 6.1 the Decoder predicted **test** instead of **get**, then **get** would get fed as input to the Decoder for step 2 because it is the ground truth output for the previous step. This way the model can learn faster compared to feeding in the wrong predictions at every decoding step.

6.5. Inference

At each time step, the decoder outputs a conditional probability distribution over the output vocabulary. That distribution is conditional on the previously predicted tokens and the context of the decoder (i.e., the decoder hidden state of the previous step or the encoder hidden state if on the first step). This distribution is then sampled so that a token for the output sequence is selected. The sampled token is then fed back into the decoder, so that the next token is predicted and so on, until the <eos> (i.e., "end of sequence") token is reached.

In order to obtain the most probable sequence, the conditional probability of the generated sequence, as defined in Formula 6.4 below, must be maximized.

$$\prod_{t=1}^{M} P(y_t \mid y_1, \ldots, y_{t-1}, \mathbf{h})$$

Formula 6.4. The total conditional probability of generating the output sequence $y_1, \ldots, y_M$. M is the maximum output sequence length, *t* denotes each timestep in the sequence and h is the context on which the prediction is conditioned, i.e., the LSTM hidden state.

However, finding this optimal sequence is very computationally expensive if an exhaustive search is done. If the vocabulary size is 10000 and the maximum output sequence length M is 10, then there would be 10^{40} possible sequences.

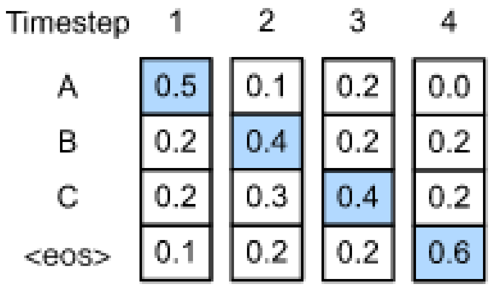
6.5.1. Greedy Search Sampling

The greedy search strategy is an alternative to exhausting all sequences. At each time step, it samples the most probable token until reaching the <eos> token. Formula 6.5 illustrates this process.

$$y_t = \underset{y \in \mathcal{V}}{\operatorname{argmax}} P(y \mid y_1, \ldots, y_{t-1}, \mathbf{h})$$

Formula 6.5. Token y_t is the most probable token from the conditional probability distribution over the vocabulary $\mathcal{V}$. The probability is conditional on the previously predicted tokens and the context h.

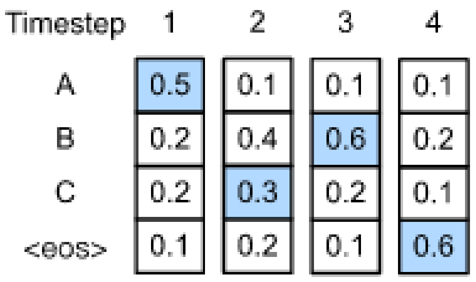
Greedy Search Sampling completes in at most M steps, where M is the maximum sequence length. However, it is not optimal as illustrated by the following example. Assume that the sequence { A, B, C, <eos> } is sampled because A is most probable at step 1, B is most probable at step 2, and so on. The total probability for the sequence is 0.5 * 0.4 * 0.4 * 0.6 = 0.048, as illustrated in Figure 6.2.



Timestep	1	2	3	4
A	**0.5**	0.1	0.2	0.0
B	0.2	**0.4**	0.2	0.2
C	0.2	0.3	**0.4**	0.2
<eos>	0.1	0.2	0.2	**0.6**

Figure 6.2. Greedy Search Sampling samples the most probable token at each timestep. [26]

What could happen if at step 2 C was sampled instead of B? Selecting the second most probable token at step 2, i.e. C, will influence the probabilities for step 3 and step 4. This on its own could lead to a higher total probability for the sequence. As seen in Figure 6.3, the total probability now becomes 0.5 * 0.3 * 0.6 * 0.6 = 0.054, which is higher than the probability of 0.048, obtained greedily.



Timestep	1	2	3	4
A	**0.5**	0.1	0.1	0.1
B	0.2	0.4	**0.6**	0.2
C	0.2	**0.3**	0.2	0.1
<eos>	0.1	0.2	0.1	**0.6**

Figure 6.3. Selecting the second most probable token at timestep 2 changes the probabilities for timesteps 3 and 4 and can actually lead to a higher total probability for the sequence. [26]

6.5.2. Beam Search Sampling

Beam Search is a heuristic graph searching algorithm which explores only the K most promising nodes at each step. K is the beam size hyperparameter, i.e., the size of the set of most promising nodes to keep at each step. This parameter is chosen upfront.

At the first time step, the K tokens with the highest conditional probability are selected to be the first token of the K candidate output sequences. For each subsequent time step, the K output sequences with the highest conditional probability are selected from the total of K × |V| possible output sequences based on the K candidate output sequences from the previous timestep. These will be the candidate output sequences for that time step and this process is repeated until the maximum sequence length M is reached. Every time a candidate sequence reaches the <eos> token, it is discontinued. In other words, all subsequences after the <eos> marker are discarded in order to obtain the set of final candidate output sequences.

Finally, the algorithm selects the K best sequences out of all accumulated candidate sequences throughout the time steps based on the scoring function in Formula 6.6. For each of these sequences, the score is computed, where L is the length of the sequence, and α is a sequence length penalty hyperparameter in the range (0; 1]. When the penalty α is closer to 1, the shorter sequences are preferred. And when α is smaller, then longer sequences are preferred. This value is selected empirically.

$$\frac{1}{L^\alpha}\log P(y_1,\ldots,y_L) =$$
$$= \frac{1}{L^\alpha}\log\prod_{t=1}^{L} P(y_t \mid y_1,\ldots,y_{t-1},\mathbf{c}) =$$
$$= \frac{1}{L^\alpha}\sum_{t=1}^{L}\log P(y_t \mid y_1,\ldots,y_{t-1},\mathbf{c})$$

Formula 6.6. The scoring formula for the final sequence candidates in Beam Search Sampling.

Beam Search is sub-optimal, but it can produce better results than Greedy Search when K > 1. It has a time complexity of O(M × K × |V|) because it makes at most M time steps and on each time step, it selects the K best nodes out of K × |V| candidates. The memory complexity is O(M × |K|) because at each time step the K best sequence continuations are kept. Thus, it is significantly more efficient than Exhaustive Search, which is of exponential complexity

Beam Search Sampling is widely used for inference in sequential models in the Deep Learning community. The most important aspect of its application is to ensure that the prediction of the next item in the sequence is conditional on the predicted items before it.

Beam Search is an ideal fit for the method name suggestion IDE tool because it can provide a number of different suggestions ranked by relevance. In this thesis, it is implemented from scratch, in order to meet the needs of the IDE tool.

6.6. Regularization

One of the problems with machine learning models is overfitting. This happens when the model has good predictive performance on the training data, but poor performance on the test data. Overfitting can be prevented through the use of regularization methods.

6.6.1. Dropout

Dropout [27] is a widely used computationally efficient regularization technique that disables the activation of a randomly selected subset of non-output units of the neural network on each training step. This method is configured by one hyperparameter – the dropout rate or the probability for a unit to be disabled during a training step. A dropout rate below 0.2 is considered in this thesis.

Dropout practically explores different network architectures, particularly subnetworks. This is similar to ensemble learning methods like bagging [28] where k different models are trained independently with sampled data from the same dataset. While there are k independent models in bagging, in Dropout the randomly activated subnetworks share parameters with other subnetworks allowing it to explore a fraction of an exponential amount of models in tractable amount of time and memory. Both bagging and dropout feed subsets of the original dataset to the models in the ensemble.

6.6.2. Early Stopping

When training machine learning models it is often the case that both the training error and validation error decrease simultaneously until reaching a point after which only the training error decreases, but the validation error starts increasing. The strategy of early stopping [29] is to stop the training process when the model has not improved its validation performance for a predetermined number of epochs. Figure 6.4 illustrates this. If the last epoch with improvement is epoch E, then the parameters of the model at epoch E are kept or it is trained for E epochs.

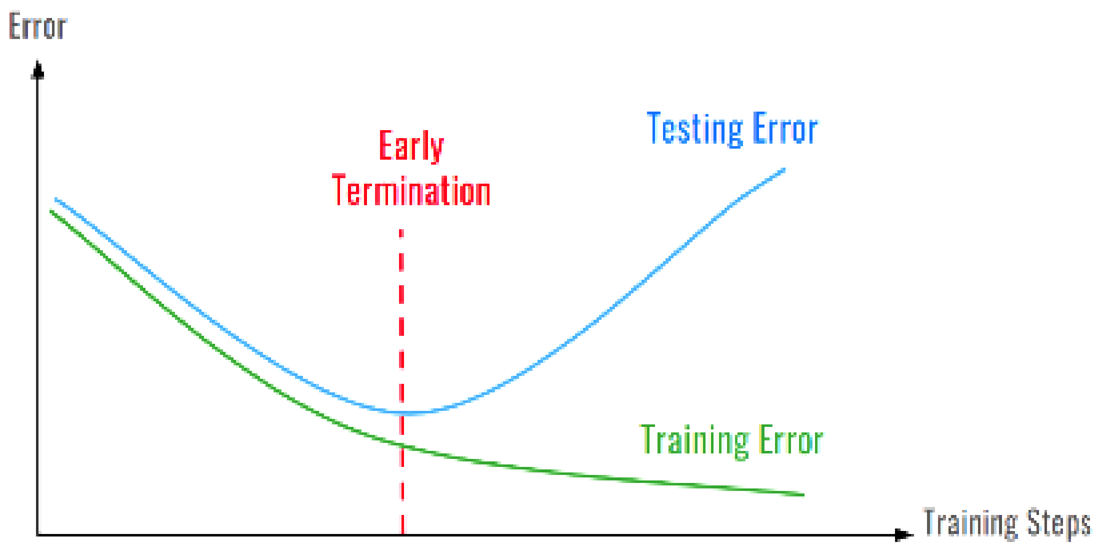


Figure 6.4. Illustration of early stopping. The training procedure is terminated before the testing error starts increasing.

Early stopping is easy to apply because it does not require changes to the loss function of the model. The early stopping procedure can wrap and control the underlying training procedure without modifying it. It can also be applied alongside other regularization methods.

Also, early stopping can be seen as an easy to implement alternative to parameter norm regularization methods, like L^2 regularization, which is a form of weight decay [30]. Weight decay penalizes complexity by keeping parameters closer to their norm, forcing the weights to be small, but non-zero.

6.7. Hyperparameters

The hyperparameters of the model control its behavior and have to be carefully chosen in order to achieve the best predictive performance. Changing some of the hyperparameters affects many aspects of the model, some of which are its learning capacity, training speed, and computational complexity. Hyperparameter optimization can be done manually or automatically through algorithms like Random Search. Such experiments are presented in Chapter 8.

Below is a list of the most notable hyperparameters of the model and their effects:

1. **Learning Rate**: This is perhaps the most important parameter. It is very often set to 0.001 by default. Increasing the value would result in larger weight updates, making the model more unstable. The optimizer can quickly diverge away from a local minimum which can be beneficial if it finds a better minimum, or have an adverse effect if it never finds a minimum – resulting in very poor performance. A smaller learning rate leads to a more stable optimizer, but also requires more training time.

2. **Dropout Rate**: A higher dropout rate would disable more units in the model, leading to a smaller learning capacity, but potentially better generalization. A lower dropout rate increases the model capacity, but can lead to overfitting.

3. **Batch Size**: A higher batch size increases the computational performance of the training procedure because less time is spent for memory access, but it requires more operational memory. However, decreasing the batch size can actually improve generalization performance of the model because of the greater gradient estimation noise of using smaller batches [31], but at the cost of slower training times due to increased memory access.

4. **Embedding Dimensions**: The larger the number of embedding dimensions, the more parameters the model has to optimize which leads to increased learning capacity, but greater computational complexity. Unfortunately, there is no golden rule of thumb for selecting this hyperparameter, so it will be selected after empirical testing with various values that can be as small as 30 and as large as 300.

5. **Latent Dimensions**: This is the RNN unit's hidden size. Again, it is best to select it after empirical testing. In most cases it is larger than the embeddings dimensions

because the latent space of the RNN models sequences that are more complex than the singular tokens modeled by the embedding layer. Increasing this hyperparameter increases the model's learning capacity and the training time.

6. **Early Stopping Patience**: This is the required number of consecutive epochs without improvement before the training procedure is stopped. There is also a minimum delta hyperparameter – the minimum amount of improvement in score required to restart the non-improved epoch counter to zero.

The model behavior is heavily affected by the data it is working with. Below is a list of the hyperparameters related to data preprocessing:

1. **Vocabulary Sizes**: Both input and output sequences have vocabularies and the larger they are, the more parameters the model will have, resulting in greater learning capacity and slower training cycles. Increasing the output vocabulary size too much makes it too hard for the decoder to generate a probability distribution over the vocabulary. But if the vocabulary is too small, then some of the tokens will not be recognized and will be marked as "out of vocabulary" (the special <oov> token).

2. **Maximum Sequence Lengths**: The longer the input or output sequences, the more time it takes for the model to train. Having too long sequences can make it hard for the model to model long-term dependencies in the data, but if the sequences are too short, then useful information would not be utilized.

7. Evaluation

In order to find the best model that solves the method name suggestion problem, it is necessary to have a way of comparing models based on their suggestion performance. This section explains what is the evaluation process and how it is done in this thesis.

7.1. Training, Validation and Test Sets

In order to obtain an unbiased estimate of the model's performance, the dataset is split into three non-intersecting subsets:

1. A **training** dataset that is used for training the models.
2. A **testing** dataset that is used to evaluate the model's performance. This dataset is held-out during model training and contains data unseen by the model.
3. A **validation** dataset is used for model evaluation while searching for the optimal hyperparameters of the model. The testing subset is not used for hyperparameter optimization because otherwise the final evaluation scores would be biased.

The proportions of the three subsets are empirically chosen. For the experiments in this thesis, a split of 80/10/10 has been used.

It is often possible for the model to perform better on the training data than on the testing data. This behavior is called overfitting and it can happen for various reasons, e.g. training for too long or the model having too many parameters. It can also happen when the training and testing subsets' distribution of samples is different. For example, the training subset may have method samples only from database projects and the testing subset may have samples from game development projects. This issue can be avoided by doing the split for each project so that each of the three subsets represents all application domains, as it is done in Section 5.4.

7.2. Evaluation Metrics

Ideally, the method name suggestions of the model can be evaluated by expert programmers through code review. However, this is very labor-intensive and expensive.

Instead, automated evaluation is done through the use of evaluation metrics that measure the model's prediction performance. The choice of appropriate evaluation metrics strongly depends on the problem definition and the data. Since the output of the model is sequential, evaluation metrics that work on sequences seem as a good fit. A deeper analysis of the problem definition should be done in order to choose the most appropriate evaluation metrics.

7.2.1. Unigram Precision, Recall, F1

One simple way of evaluating the predictions is to treat the output sequences as sets and calculating what proportion of the predicted sequence overlaps with the reference sequence.
For example, if the reference sequence is **transformSearchResponse** and the predicted sequence is **removeSearchDataResponse**, then the number of overlapping subtokens is 2 (**search** and **response**). Then the following measures can be defined:

- **Precision** measures the number of predicted subtokens that are relevant. It is calculated by dividing the number of overlapping tokens by the length of the predicted sequence, i.e., 2 / 4 = 0.5.
- **Recall** measures the number of relevant subtokens that are predicted. It is calculated by dividing the number of overlapping subtokens by the length of the reference sequence, i.e., 2 / 3 = 0.6666.
- **F1** is the harmonic mean of precision and recall, i.e., F1 = 2 * P * R / (P + R) = 0.5714

In this thesis, 2 ways of averaging these scores have been implemented:

- Micro-averaging – averaging the precision and recall scores across all samples.
- Macro-averaging – averaging the precision and recall scores per class, i.e., per token.

This way of evaluation does not restrict the position of the predicted tokens. Alternatively, the precision and recall metrics can take into account the order of the tokens, however that would be too restrictive. Instead, evaluation metrics that work with n-grams are considered in the following Section 7.2.2.

7.2.2. N-gram Evaluation

According to the problem statement in Section 3, the model has to suggest the method name based on the method's definition. This problem can be interpreted as machine translation where the input source code sequence is translated into an output sequence of method name subtokens. BLEU is a very popular metric used for the evaluation of machine translation models [32].

Another point of view is that the method name essentially represents the meaning of the method itself. Thus, the problem can also be reduced to extreme text summarization [17] where the source code of the method has to be reduced to a short sequence that best represents the meaning of the method. ROUGE is widely used for evaluating text summarization [33] and is very similar to BLEU in the sense that it evaluates n-grams. ROUGE is a generalization of the unigram precision and recall that were discussed in Chapter 7.2.1 and it is chosen as the evaluation metric in this thesis.

7.2.3. ROUGE

ROUGE, or Recall-Oriented Understudy for Gisting Evaluation, [33] is a set of metrics used for evaluating text summarization in natural language processing. The metrics compare the predicted sequence against the reference sequence. In this thesis, it compares the predicted and reference method names at the subtoken level.

ROUGE has a few modifications:

- **ROUGE-N**: Evaluates the overlapping n-grams between the predicted and reference sequences. ROUGE-1 provides unigram precision, recall and F1 scores that are calculated the same way as in Section 7.2.1. ROUGE-2 measures at the bigram level, i.e., any two adjacent tokens, ROUGE-3 is at the trigram level and so on.

- **ROUGE-L**: Finds the longest common subsequence of tokens in the predicted sequence that overlaps with tokens in the reference sequence and divides that number by the length of the prediction for the precision measure and by the length of the reference sequence for the recall measure.

The Python package **rouge-score** [34] provides a reference implementation of the metrics and is used in this thesis. This package also provides stemming of tokens, so that the evaluation is less restrictive. The scores in Section 8 are calculated with stemming enabled. For example, the predicted method name **startDownloadingProcess** will have a full match with the reference **startDownloadProcess** because the stem of the second token is **download**.

8. Experiments

This section discusses the experiments that were executed[2] in this thesis. An experiment consists of training the model on a dataset and evaluating its performance on a test set or validation set. Throughout the development of the model multiple experiments were executed in order to debug the model and resolve issues with its implementation or to find better hyperparameters. A few of these experiments were very helpful in finding out issues with the model concerning its slow computational performance (low GPU usage). These issues were resolved afterwards.

While executing the initial experiments, the hyperparameters were adjusted so that the scores could improve and before any hyperparameter optimization was done, the best ROUGE-L F1 score on the test set was achieved by a model with:

- Embedding Dimensions: 50
- Vocabulary Size: 5000
- Latent Dimensions: 320
- Learning Rate: 0.0001
- Max Input Length: 128
- Max Output Length: 8
- Dropout Rate: 0.0

The model had a total of 3 670 121 parameters and achieved a test ROUGE-L F1 score of 0.5226 and a training score of 0.6374 (potentially overfitting)[3].

8.1. Hyperparameter Optimization

The hyperparameters of the model can be optimized through manual experimentation of using automated searching strategies. Two such strategies are grid search and random search. In this thesis, random search was preferred because there are too many hyperparameters and too little hardware resources available to carry out a meaningful grid search. Also, it has been shown that random search selects (samples) more efficient trials for hyperparameter optimization than grid search [35].

[2] All experiments are recorded and can be found at https://app.wandb.ai/antonpetkoff/identifier-suggestion
[3] https://app.wandb.ai/antonpetkoff/identifier-suggestion/runs/2h37hin6?workspace=user-antonpetkoff – link to the experiment in the Weights & Biases platform with charts, examples, and logs.

Two random searches were carried out. The first was not as successful, because the learning rate was too high (drawn from a random uniform distribution in the range [10^{-5}; 10^{-1}], which produced values closer to 10^{-1}) and the loss did not converge on most experiments.

The second random search was more successful. The parameters below were sampled uniformly from the following sets:

- Batch Size: {32, 64, 128, 256, 512, 1024}
- Latent Dimensions: {256, 384, 512, 640, 768, 896, 1024}
- Embedding Dimensions: {32, 48, 64, 80, 96, 112, 128}
- Vocabulary Size: {2000, 3000, 4000, 5000, 6000, 7000}
- Max Input Sequence Length: {50, 75, 100, 125, 150, 175}
- Max Output Sequence Length: {5, 6, 7, 8}

The continuous parameters were sampled as follows:

- Learning Rate: 10^{-N} where N is uniformly distributed in the range [-4; -2]
- Dropout Rate: uniformly distributed in the range [0.0; 0.2]

The integral values were drawn intentionally from sets, instead of discrete distributions, so that there is a higher variance in the values that are used for the experiments. Also, early stopping patience was fixated to 3 epochs and early stopping minimum delta was 0. 20 experiments were sampled from the distributions above and are saved in the file **experiments/random_search_1.json** in the GitHub project of the thesis[4].

8.1.1. Results

Table 8.1 contains the top results from the second wave of Random Search experiments. They are sorted from left to right, where the best experiments are on the left. Experiment #18 had the highest validation score, however it has the smallest vocabulary size, which means that most of the predictions contained the special <oov> (out of vocabulary) token.

Experiment #10 has a 3.5 times larger vocabulary compared to #18, allowing it to model much more method names. It reached its best ROUGE-L F1 on the validation set after just 10 epochs with a relatively large batch size of 512, resulting in one of the fastest training times. The hyperparameter values of experiment #10[5] were used to train the best model in this thesis and evaluate it on the test set, achieving an impressive score of **0.5955** ROUGE-L F1 on the test set (and a score of 0.6549 on the training set). This model is the one used for the demonstrations and examples in Section 9 where it shows good generalization performance.

[4] https://github.com/antonpetkoff/identifier-suggestion/blob/master/experiments/random_search_1.json

[5] https://app.wandb.ai/antonpetkoff/identifier-suggestion/runs/p13km0ym?workspace=user-antonpetkoff - Link to the experiment in Weights & Biases with all its metrics tracked throughout the training and evaluation process.

Experiment	#18	#10	#8	#5	#11	#15	#13	#17	#3	#14
Batch Size	256	512	256	512	32	512	32	128	1024	128
Dropout Rate	0.1407	0.1164	0.0421	0.0947	0.0276	0.0531	0.121	0.1106	0.1556	0.1377
Embedding Dim	80	128	80	128	48	80	48	96	32	128
Latent Dim	896	768	256	640	640	1024	256	256	512	640
Learning Rate	0.0014	0.0032	0.0023	0.0003	0.0018	0.0065	0.0013	0.0018	0.0011	0.0001
Max Input Length	50	175	175	150	175	75	125	50	75	125
Max Output Length	8	6	5	7	7	5	7	6	8	7
Vocabulary Size	2000	7000	3000	2000	2000	4000	5000	4000	7000	5000
Epoch	39	10	32	40	11	11	21	36	39	40
Val. ROUGE-L F1	0.5899	**0.5829**	0.5684	0.5654	0.5607	0.56	0.5558	0.554	0.5311	0.5296
Val. ROUGE-1 F1	0.5917	**0.584**	0.5694	0.5672	0.5625	0.561	0.5575	0.5552	0.5324	0.5313
Val. ROUGE-2 F1	0.3503	**0.3369**	0.3241	0.3196	0.3107	0.3136	0.3099	0.3113	0.2842	0.2769
Val. ROUGE-3 F1	0.1711	**0.1642**	0.1547	0.1461	0.1435	0.1488	0.1461	0.1472	0.1254	0.1167

Table 8.1. The top 10 from the 20 Random Search experiments with the highest Validation ROUGE-L F1 score, sorted from left to right.

Table 8.1 alone does not give much insight about the impact of the hyperparameters values because their relationship is not that simple and there are too few experiments in the Random Search. However, this Random Search achieved our goal by improving the best scores before it from 0.5226 ROUGE-L F1 on the test set to 0.5955 (13.9% relative gain) without increasing the training score too much (just by 2.7%), which is a sign that the model does not overfit the training data. This final model has a total number of 16 227 417 parameters, making it one of the most complex models that was built in this thesis.

Learning Rate Impact

The learning rate is the most sensitive hyperparameter and its impact on the model's performance is illustrated on Figure 8.1. It is observed that learning rates below 0.0005 (too small) and above 0.006 (too large) result in bad scores. Also, training time increases with smaller learning rates.

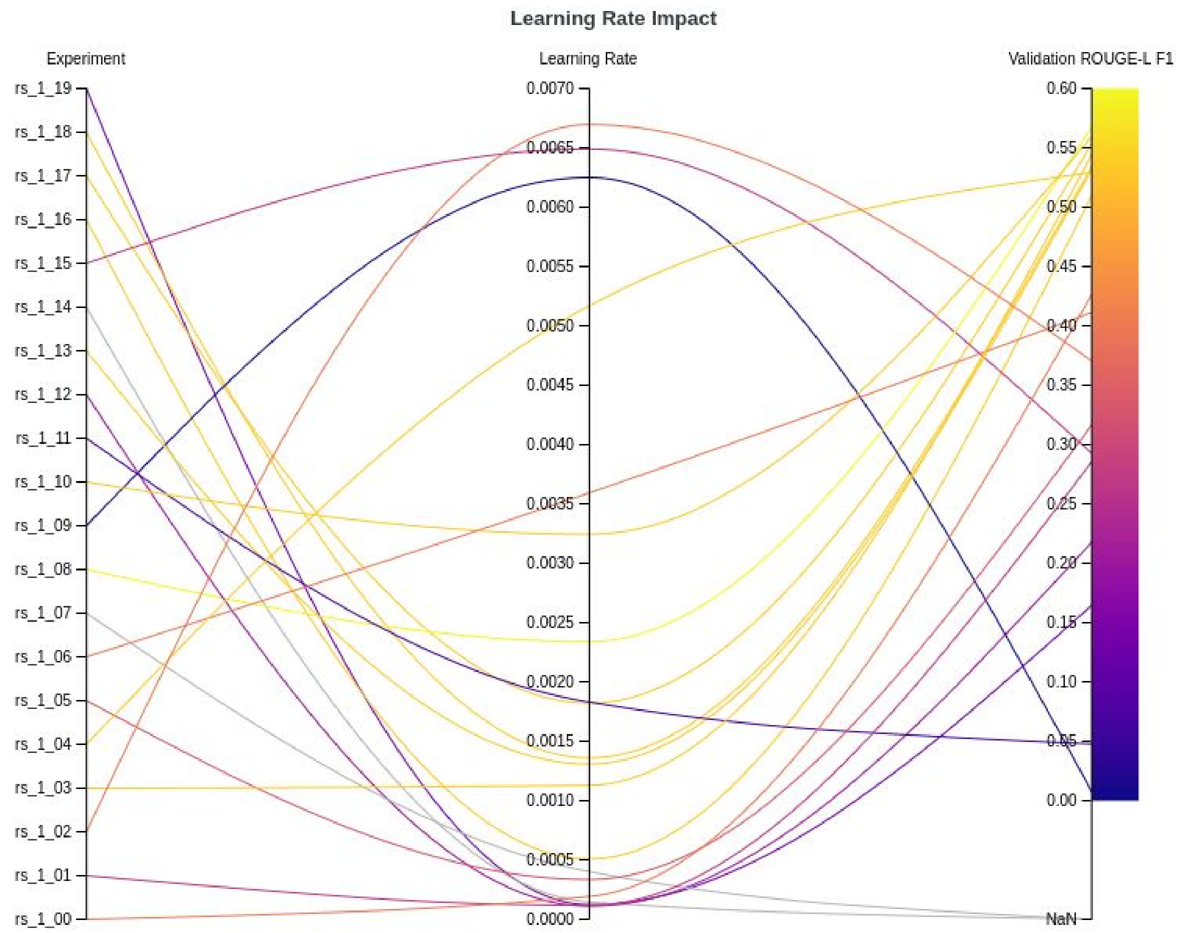


Figure 8.1. Plot of the 20 experiments and their corresponding learning rates and validation set ROUGE-L F1 scores.

Overfitting

It is often the case that the model overfits the training data, when the model has a much higher evaluation score on the training set than on the validation set. Dropout and Early Stopping alleviate this problem.

The Dropout Rate correlates inversely with a lower difference between the training and validation set ROUGE-L F1 scores, as shown on Figure 8.3. The correlation coefficient is -0.3585. This does not necessarily mean that there is a strong inverse relationship between the two variables. Instead, a separate experiment where only the dropout rate varies should be carried out because the random search experiment generated only 20 combinations of 8 different parameters which creates noise.

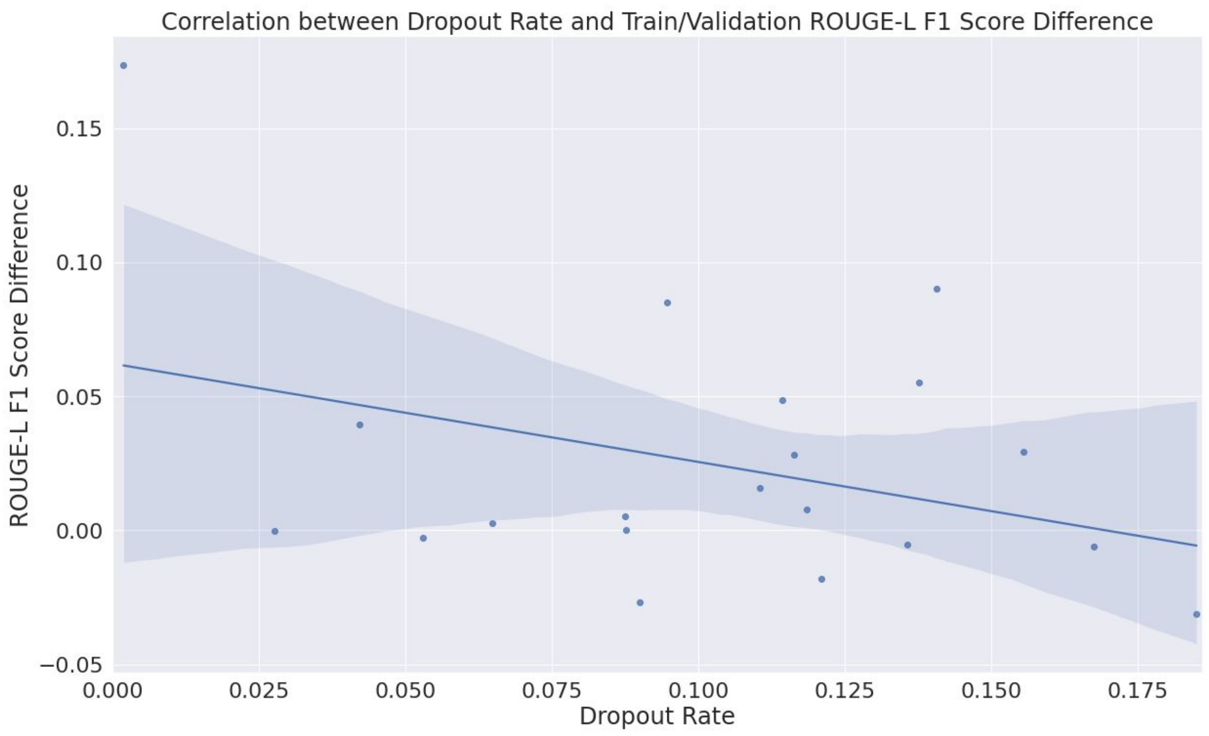


Figure 8.2. Linear regression (Correlation) plot between Dropout Rate and Train/Validation Score Difference.

The 5 experiments that result in the largest difference between training and validation scores are illustrated on Figure 8.3. It can be seen even though experiment #16 has early stopping with patience 3, it still managed to overfit the training data so much that it has the largest train-validation score difference of 0.1738. Experiment #16 has the lowest dropout rate of 0.0019.

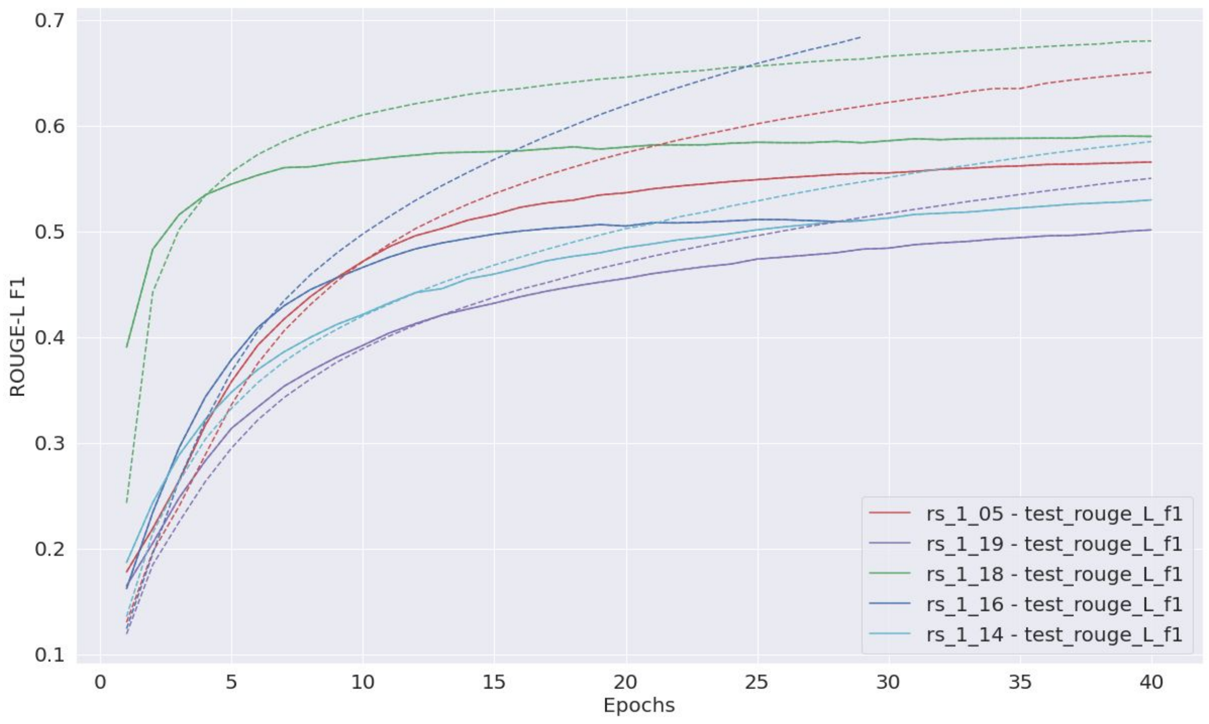


Figure 8.3. Top 5 most overfitting models.

It can also be hypothesized that the batch size also impacts training data overfitting. A correlation coefficient of 0.2377 was measured between the batch size and the difference between training and validation scores, i.e., it is more probable to overfit the training set with a larger batch size.

8.2. Attention Weights Visualization

Deep Learning models are very hard to interpret because they have a lot of parameters, millions in this thesis. The Attention Mechanism of our model can be used to plot a heatmap of the attention weights. This way it can be seen which subtokens in the method body are most important to the model when predicting the method name.

Figure 8.4 illustrates such a heatmap for the popular Object.equals method. The method body is on the x-axis and the predicted method name is on the y-axis. For the first predicted output subtoken "equals", the model focuses mostly on the comparison operators **instanceof** and ==.

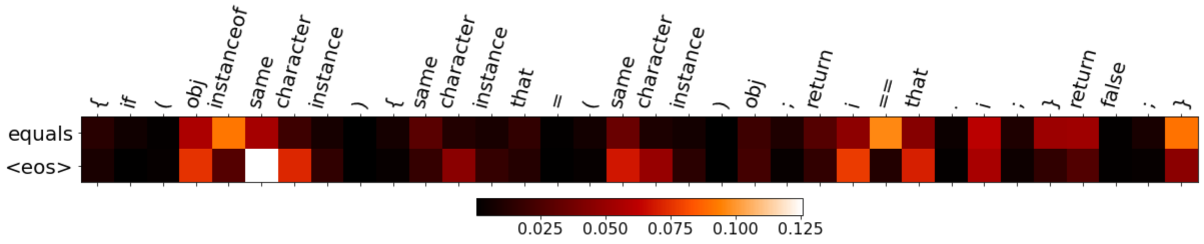


Figure 8.4. Heatmap of the attention weights for the equals method (inherited from Java Object).

Figure 8.5 illustrates an even easier to interpret heatmap of a simple getter method. The class member variable "remoteRootLogDir" has been tokenized into subtokens and the attention mechanism has correctly focused on each of the subtokens to re-create a getter method name for that value.

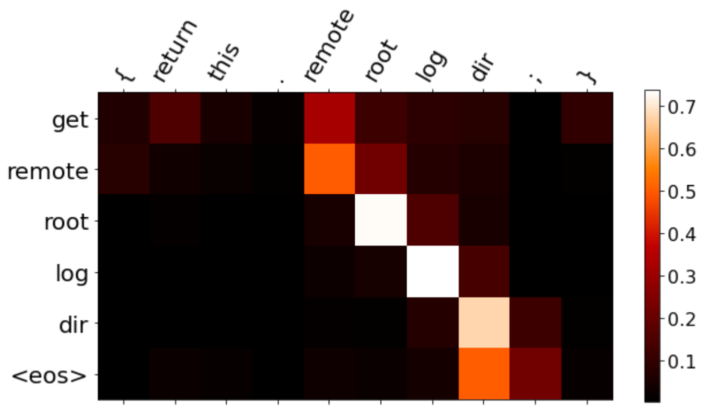


Figure 8.5. Heatmap of the attention weights for a getter method.

8.3. Embeddings Projection

Figure 8.6 illustrates what the encoder embedding layer of the best performing model has learned. The figure shows the 22 word vectors that are closest to the token **list**. Some of these words are very closely related to lists:

- **stream** – an infinite list
- **nums** – an identifier that is often used to identify a list of numbers
- **suffixes** – also an identifier appropriate for a list of list suffixes
- **concatenate** – operation on lists
- **count** – refers to how many items a list has
- **of** – referring to the Java Stream.of constructor

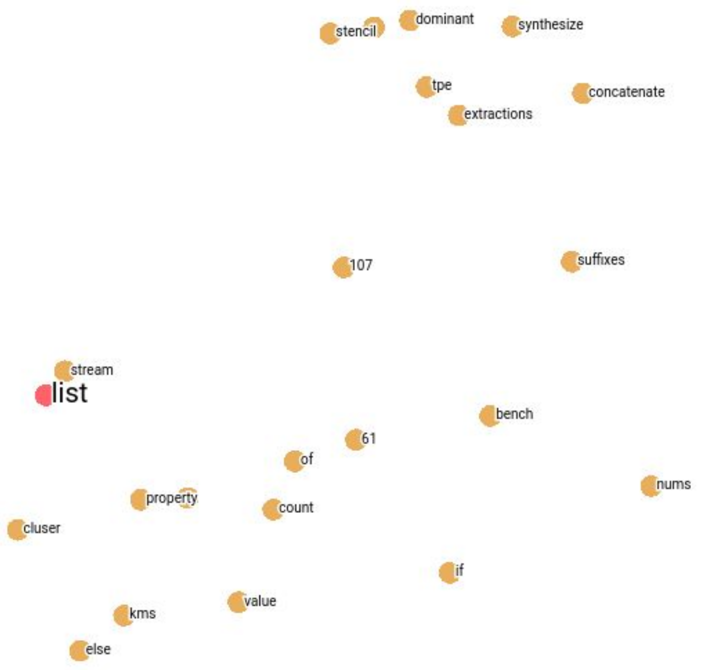
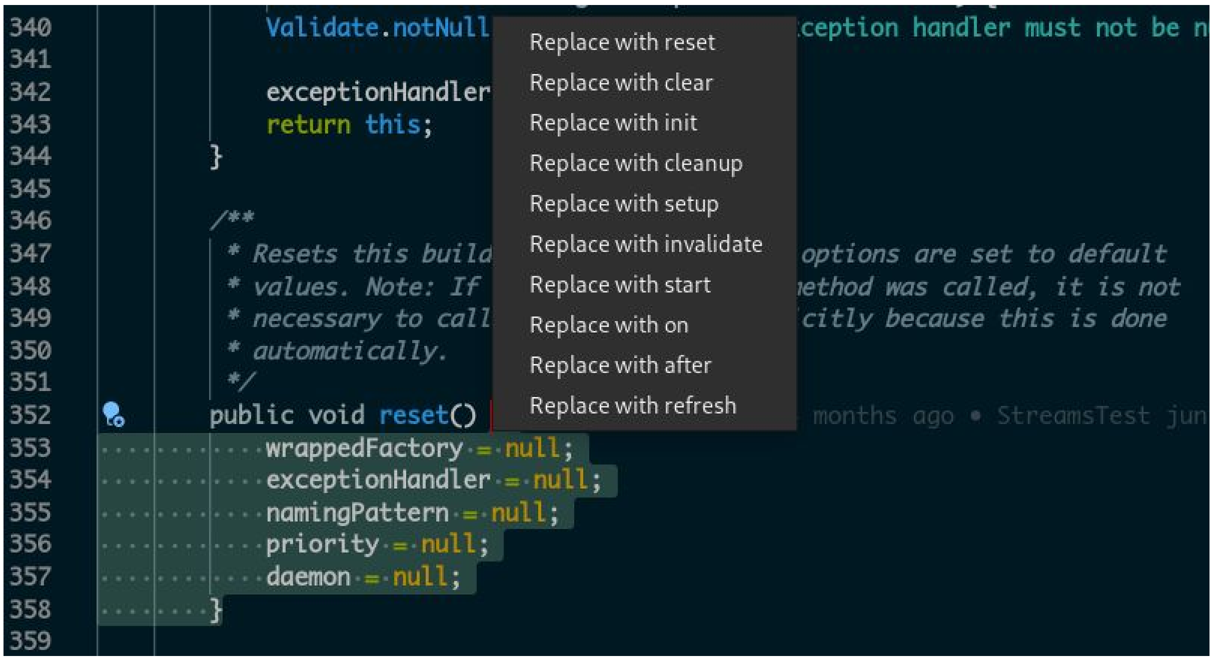


Figure 8.6. t-SNE plot [36] of the encoder embedding vector space of the best performing model.

9. Demonstration and Examples

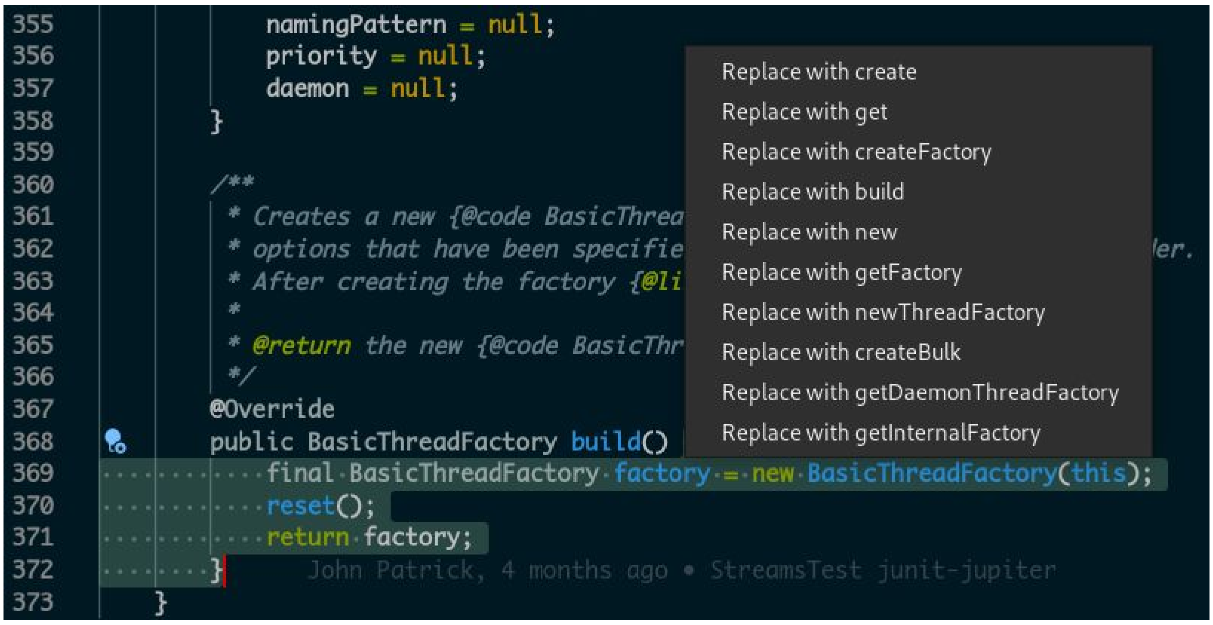
In order to demonstrate the capabilities of the model, an IDE extension for the popular source-code editor Visual Studio Code is created so that it can be applied to real-world Java code. The extension is intended to assist programmers in finding the most suitable method names for the methods they write. This section provides examples that demonstrate how this extension works with the best model that was developed in the previous sections.



```
340            Validate.notNull                              ception handler must not be n
341
342            exceptionHandler
343            return this;
344        }
345
346        /**
347         * Resets this build                                  options are set to default
348         * values. Note: If                                 ethod was called, it is not
349         * necessary to call                                 citly because this is done
350         * automatically.
351         */
352        public void reset()                                   months ago • StreamsTest jun
353            wrappedFactory = null;
354            exceptionHandler = null;
355            namingPattern = null;
356            priority = null;
357            daemon = null;
358        }
359
```

Figure 9.1. Demonstration of the IDE extension suggesting method names for the method body between lines 352 and 358.

The first example in Figure 9.1 demonstrates that the model can provide not only the correct method name, but also similar names that are also very appropriate. In this case, it can be seen how well the model has learned words similar to **reset**. The programmer does not even need to search for synonyms because the model directly provides them.

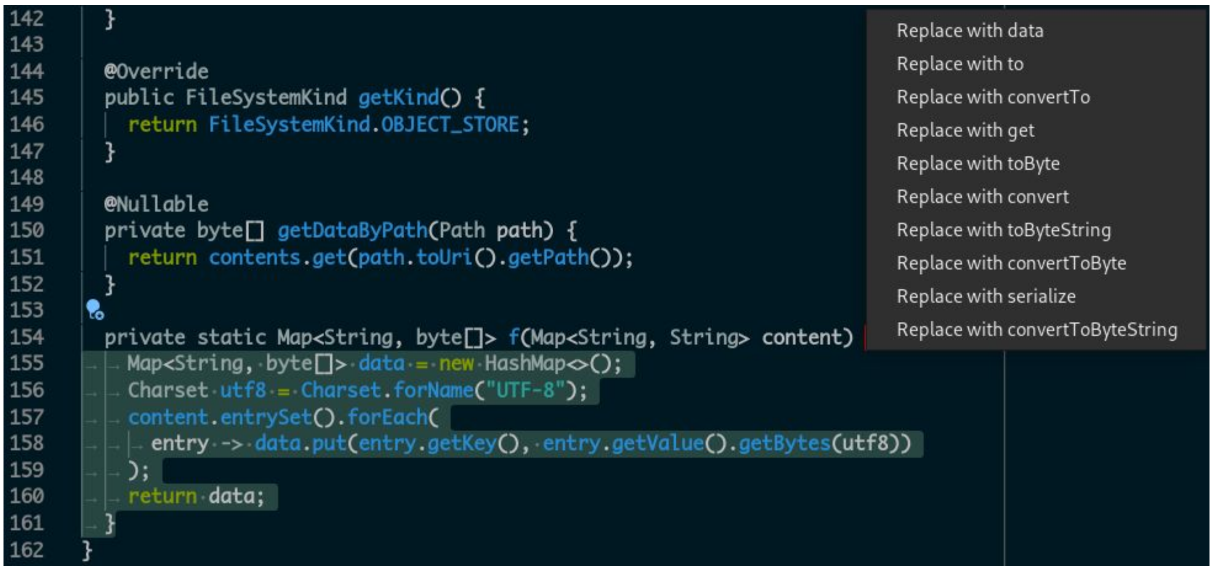

Replace with create
Replace with get
Replace with createFactory
Replace with build
Replace with new
Replace with getFactory
Replace with newThreadFactory
Replace with createBulk
Replace with getDaemonThreadFactory
Replace with getInternalFactory

```
            namingPattern = null;
            priority = null;
            daemon = null;
        }

        /**
         * Creates a new {@code BasicThrea
         * options that have been specifie                                    ler.
         * After creating the factory {@li
         *
         * @return the new {@code BasicThr
         */
        @Override
        public BasicThreadFactory build()
            final BasicThreadFactory factory = new BasicThreadFactory(this);
            reset();
            return factory;
        }      John Patrick, 4 months ago • StreamsTest junit-jupiter
    }
```

Figure 9.2. The IDE extension in action.

As it can be seen in Figure 9.2, which demonstrates the extension in Visual Studio Code, the model generalizes pretty well and suggests multiple appropriate method names (summaries), containing two or more subtokens, for the selected code snippet. Despite the possibility of this method being in the training set or the fact that it is annotated with @Override (it has a single possible method name), the model managed to suggest new ideas for naming this method. This makes the tool not just an assistant, but more akin to a peer who suggests new naming ideas as opposed to simply performing tedious tasks automatically.

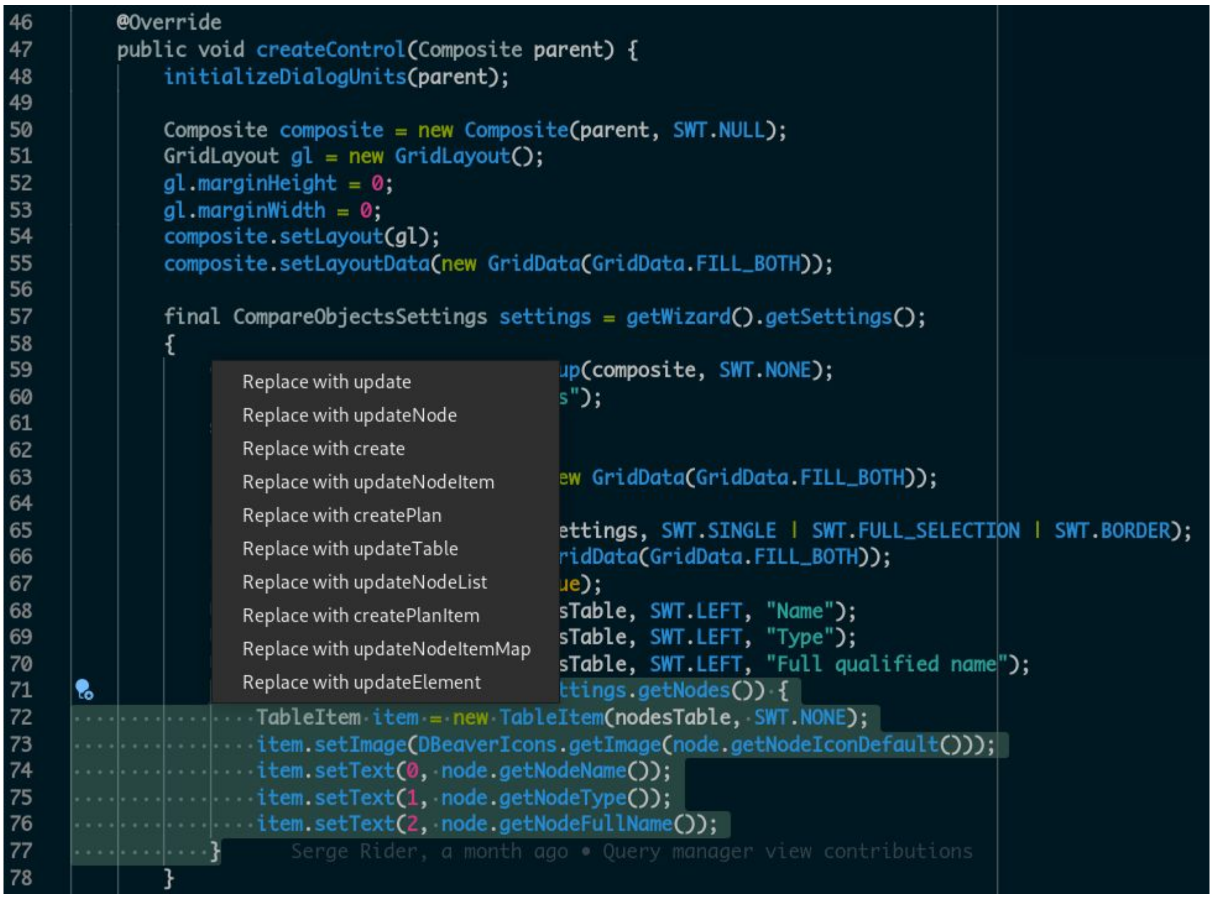


```
142     }
143
144     @Override
145     public FileSystemKind getKind() {
146       return FileSystemKind.OBJECT_STORE;
147     }
148
149     @Nullable
150     private byte[] getDataByPath(Path path) {
151       return contents.get(path.toUri().getPath());
152     }
153
154     private static Map<String, byte[]> f(Map<String, String> content)
155       Map<String, byte[]> data = new HashMap<>();
156       Charset utf8 = Charset.forName("UTF-8");
157       content.entrySet().forEach(
158         entry -> data.put(entry.getKey(), entry.getValue().getBytes(utf8))
159       );
160       return data;
161     }
162   }
```

Figure 9.3. Another example where a method name is suggested for the method between lines 154 and 161.

Figure 9.3 illustrates a more complex example where the model still manages to suggest appropriate method names, but they are lower in the list. This is because shorter method names have a higher score in the Beam Search Decoder. This behavior can be modified by adjusting the alpha parameter from Section 6.5.2. By lowering the alpha hyperparameter, longer method names would come up higher in the list of suggestions.

Because the model works with source code snippets as input, the tool can be also used to summarize any source code snippet within a method, as shown in Figure 9.4. This can be very useful to the programmer when refactoring code, e.g. extracting a chunk of code into a new method. In Figure 9.4, the selected code snippet is looping through nodes and creates new table items from the nodes that are then updated.


Replace with update
Replace with updateNode
Replace with create
Replace with updateNodeItem
Replace with createPlan
Replace with updateTable
Replace with updateNodeList
Replace with createPlanItem
Replace with updateNodeItemMap
Replace with updateElement

```
46      @Override
47      public void createControl(Composite parent) {
48          initializeDialogUnits(parent);
49
50          Composite composite = new Composite(parent, SWT.NULL);
51          GridLayout gl = new GridLayout();
52          gl.marginHeight = 0;
53          gl.marginWidth = 0;
54          composite.setLayout(gl);
55          composite.setLayoutData(new GridData(GridData.FILL_BOTH));
56
57          final CompareObjectsSettings settings = getWizard().getSettings();
58          {
59                                          up(composite, SWT.NONE);
60                                          s");
61
62
63                                          ew GridData(GridData.FILL_BOTH));
64
65                                          ettings, SWT.SINGLE | SWT.FULL_SELECTION | SWT.BORDER);
66                                          ridData(GridData.FILL_BOTH));
67                                          ue);
68                                          sTable, SWT.LEFT, "Name");
69                                          sTable, SWT.LEFT, "Type");
70                                          sTable, SWT.LEFT, "Full qualified name");
71                                          ttings.getNodes()) {
72                  TableItem item = new TableItem(nodesTable, SWT.NONE);
73                  item.setImage(DBeaverIcons.getImage(node.getNodeIconDefault()));
74                  item.setText(0, node.getNodeName());
75                  item.setText(1, node.getNodeType());
76                  item.setText(2, node.getNodeFullName());
77              }       Serge Rider, a month ago • Query manager view contributions
78          }
```

Figure 9.3. Summarization suggestions for the source code between lines 72 and 77.

10. Implementation

The implementation of the method name suggestion tool is freely accessible on GitHub on https://github.com/antonpetkoff/identifier-suggestion/. Its file system organization follows the Cookiecutter Data Science project structure [37]. To run the model training and serving code, Python 3.7 or above is required along with a basic Unix-like environment with a standard coreutils library. The code repository includes scripts and descriptions with steps for reproducing the datasets, models and results described in the thesis.

The following subsections cover the most important parts of the system – where are they located, how they are built and how to use them.

10.1. Data Processing Pipeline

The dataset is built in several steps which are automated by Shell and Python scripts. The file **`data/README.md`** describes how to reproduce the dataset. First, the specified Java source code repositories are downloaded in **`data/external/`**. Then, for each of these repositories the Java methods are extracted in CSV files under **`data/raw/`**. Then, the final preprocessing step cleans and tokenizes the text data and stores it under **`data/interim/`**. The `data/processed/` directory is reserved for any data processing which the model does, e.g. encoding the tokens into numbers and storing binary data in the HDF5 file format.

NumPy [38], Pandas [39], scikit-learn [40] and some other Python libraries are used for data processing. Matplotlib [41] is used for plots directly or indirectly through Pandas. Jupyter notebooks [42] are used for data analysis and visualizations. Such notebooks are **notebooks/08-exploratory-data-analysis.ipynb**, **notebooks/11-attention-plots.ipynb** and **notebooks/12-experiment-results.ipynb**.

10.2. Model Training Pipeline

Once the data set has been pre-processed and ready under data/interim/, it can be used to train the model. The script **src/scripts/seq2seq/train.sh** runs the training loop. It also includes the model-specific data preprocessing, e.g. encoding tokens into numbers, and caches the processed data under **data/processed/**. The script exposes a number of hyperparameters, like the dimensions of the embeddings, the learning rate, etc. The random number generators are seeded so that the results can be reproduced.

The training pipeline code is located in **src/pipelines/baseline.py**. It loads and preprocesses the data, trains the model on the training data and reports evaluation results on the test data. Model checkpoints are saved under **models/checkpoints/**, so that later it can be restored and used, e.g. to serve suggestions or just continue training. The custom model, training and evaluation loops are built using TensorFlow 2.2 [43].

10.2.1. Google Colaboratory

For the execution of the training pipeline, the Jupyter notebook **notebooks/00-colab.ipynb** is executed inside Google Colaboratory which provides free access to high-quality GPUs like the NVIDIA Tesla P100 and Tesla K80, depending on their availability. Google Drive is used as a persistent cloud storage to access the training data and source code, and save checkpoints and other artifacts.

The paid version, Google Colab Pro, allows 4 parallel active sessions with the Tesla P100 GPU, but they are still limited to no more than 24 hours of usage. There were a few experiments in this thesis that were not completed because the system forcibly stopped the execution.

10.2.2. Experiment Tracking

Each execution of the training pipeline is a separate experiment and they can quickly pile up and become unmanageable. This is why the pipeline is integrated with the external experiment tracking system Weights and Biases [44].

Below is a list of the most notable features of Weights and Biases and how they are applied to solve most of the experiment tracking problems in this project:

- Message Logging – log any messages, such as how much time a single training epoch takes or a summary of how many trainable parameters the model has.
- Numeric Data Tracking – track various measurements like the loss and the evaluation metrics throughout the training process. Weights and Biases then makes interactive plots which can give insight on the model performance in real time while it is training.
- System Monitoring – track in real time the utilization of hardware resource like CPU, GPU, RAM, I/O, Network, etc. This feature is very useful for monitoring the performance of the training pipeline. There were several occasions when the code written in TensorFlow was not optimal and had to be updated to work with tensor operations, so that it could be executed on the GPU.
- Rich Media Logging – images, plots and tables are some of the few objects which this project logged in Weights and Biases. A table with 10 sample predictions from the data set can be very insightful to track if and how the model predictions improve over time. Plots of the attention weights give transparency for what the model attends to the most.
- Data storage – Various artifacts like model checkpoints, vocabulary files, images, tables, etc can be saved to this storage and then restored.
- Comparison of several experiments in the main dashboard of the system can help in finding the best experiments.
- Hyperparameter Sweeps help in finding optimal hyperparameters of the experiment.

Tracking with Weights and Biases is configured through the three environment variables **WANDB_USERNAME**, **WANDB_PROJECT** and **WANDB_API_KEY** that are stored in the **.env** file. These settings can be taken from the user profile in the system. The project for this thesis is publicly available at https://app.wandb.ai/antonpetkoff/identifier-suggestion.

10.3. Model Server

Once a the model is trained, it can be restored from one of its checkpoints. The script **`src/scripts/baseline/serve.sh`** starts up an HTTP server, developed in Flask, a lightweight Python HTTP server framework [45]. The server restores the given model checkpoint and suggestions can be made using the **`http://localhost:5000/predict`** endpoint which accepts the method body as a query parameter. As a response, a JSON with a list of the suggested method declaration names is returned.

10.4. IDE Tool

The IDE tool is an extension for Visual Studio Code. Its source code is located in **vscode-extension/** and the **README.md** file explains how to build it, install it and use it in the IDE. The extension is developed in TypeScript and the npm module **axios** is used for HTTP requests to the model server, covered in the previous section.

Visual Studio Code was chosen because it is a free and open-source IDE with a large and active community supporting it. It provides very well documented APIs for extension development and comes with example projects for different use cases. The developed extension can afterwards be easily published for all users to use.

11. Conclusion and Future Work

11.1. Research Contributions

The research contributions in this thesis include:

- A new dataset for the method name suggestion problem in Java along with the framework for creating the dataset. The dataset can easily be increased by providing more source code and the framework can be adapted to other languages by swapping the programming language parsing logic.

- A Seq2Seq language model achieving 0.5955 ROUGE-L F1 score on the test set. The model is accompanied with visualizations in this thesis that help make it more interpretable.

- A VSCode extension that can be used for summarizing a Java code snippet into a method name. This tool can be used to improve method names, refactor logic and write more maintainable code.

The source code of this thesis is available online[6]. All experiments with measurements, logs and plots are also available online[7].

11.2. Future Work

The performance and application of the current model can be improved in a number of ways discussed in the subsections below.

11.2.1. Data

As with every other machine learning model, the performance of this model can be improved by crafting a larger data set. Experiments with different preprocessing methods can be done

[6] https://github.com/antonpetkoff/identifier-suggestion/
[7] https://app.wandb.ai/antonpetkoff/identifier-suggestion

to improve the quality of the data set. Some of these methods are stemming and lemmatization.

The current model can easily be extended to other programming languages. It can also be made multilingual, i.e., to work with multiple languages like Java, Python, Ruby, JavaScript, etc. However, the vocabulary would grow in size and so the dimensions of the embeddings and the recurrent layers would have to be increased.

11.2.2. Feature Engineering

Source code is written in a programming language which, unlike natural language, has a much less ambiguous structure, defined by a grammar. Parsers can extract abstract syntax trees (AST) which contain valuable information which can be used by neural networks to learn better continuous representations of source code. One such way is to create AST path embeddings for tokens that represent the path inside the AST of the token [8].

The current model takes into account only the method definition. Another approach is to consider all usages of the method. A single method can be used in multiple places throughout the source code repository. For each method call (or reference), a window of K tokens to the left and to the right of the method call can be extracted to form a context of that usage [10]. Each such context can be encoded into a continuous representation and all of them can be combined/reduced into a single feature vector.

11.2.3. Models

The NLP field constantly advances forward with new deep learning architectures that raise the best scores on benchmarks. It can be experimented with various other deep learning architectures like Convolutional Neural Networks, Transformers and even domain-specific architectures like Abstract Syntax Networks [46].

However, in order to improve the predictive performance of the Seq2Seq model, it would be easier to upgrade the current architecture by replacing the unidirectional encoder RNN with a bidirectional one. Bidirectional RNNs read the input sequence from left to right and right to left, which could improve the model’s spatial learning abilities.

12. References

[1] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, “GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding,” *arXiv [cs.CL]*, 20-Apr-2018 [Online]. Available: http://arxiv.org/abs/1804.07461

[2] “GLUE Benchmark Leaderboard.” 2020 [Online]. Available: https://gluebenchmark.com/leaderboard

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *arXiv [cs.CL]*, 11-Oct-2018 [Online]. Available: http://arxiv.org/abs/1810.04805

[4] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019 [Online]. Available: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

[5] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, “XLNet: Generalized autoregressive pretraining for language understanding, 2019,” *URL https://www. arxiv. org/abs*, 1906.

[6] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A Survey of Machine Learning for Big Code and Naturalness,” *arXiv [cs.SE]*, 18-Sep-2017 [Online]. Available: http://arxiv.org/abs/1709.06182

[7] F. Deissenboeck and M. Pizka, “Concise and consistent naming,” *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, Sep. 2006, doi: 10.1007/s11219-006-9219-1. [Online]. Available: https://doi.org/10.1007/s11219-006-9219-1

[8] U. Alon, S. Brody, O. Levy, and E. Yahav, “code2seq: Generating Sequences from Structured Representations of Code,” *arXiv [cs.LG]*, 04-Aug-2018 [Online]. Available: http://arxiv.org/abs/1808.01400

[9] D. Bahdanau, K. Cho, and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” *arXiv [cs.CL]*, 01-Sep-2014 [Online]. Available: http://arxiv.org/abs/1409.0473

[10] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Suggesting accurate method and class names,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo, Italy, 2015, pp. 38–49, doi: 10.1145/2786805.2786849 [Online]. Available: https://doi.org/10.1145/2786805.2786849. [Accessed: 07-Jul-2020]

[11] M. Allamanis, H. Peng, and C. Sutton, “A Convolutional Attention Network for Extreme Summarization of Source Code,” *arXiv [cs.LG]*, 09-Feb-2016 [Online]. Available: http://arxiv.org/abs/1602.03001

[12] K. O’Shea and R. Nash, “An Introduction to Convolutional Neural Networks,” *arXiv [cs.NE]*, 26-Nov-2015 [Online]. Available: http://arxiv.org/abs/1511.08458

[13] openhub.net, “The Apache Hadoop Open Source Project on Open Hub : Estimated Cost Page.” [Online]. Available: https://www.openhub.net/p/hadoop/estimated_cost. [Accessed: 07-Jul-2020]

[14] C. Tunes, “javalang: Java SE8 Parser In Pure Python.” 2020 [Online]. Available: https://github.com/c2nes/javalang

[15] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to Sequence Learning with Neural Networks,” *arXiv [cs.CL]*, 10-Sep-2014 [Online]. Available: http://arxiv.org/abs/1409.3215

[16] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco.1997.9.8.1735. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[17] S. Narayan, “What is this Article about? Extreme Summarization with Topic-aware Convolutional Neural Networks,” *J. Artif. Intell. Res.*, vol. 1, pp. 1–15, 1993 [Online]. Available: https://github.com/shashiongithub/XSum

[18] A. R. Babhulgaonkar and S. V. Bharad, “Statistical machine translation,” in *2017 1st International Conference on Intelligent Systems and Information Management (ICISIM)*, 2017, pp. 62–67, doi: 10.1109/ICISIM.2017.8122149 [Online]. Available: http://dx.doi.org/10.1109/ICISIM.2017.8122149

[19] F. Stahlberg, “Neural Machine Translation: A Review,” *arXiv [cs.CL]*, 04-Dec-2019 [Online]. Available: http://arxiv.org/abs/1912.02047

[20] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed Representations of Words and Phrases and their Compositionality,” in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119 [Online]. Available: http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

[21] J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation,” in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543 [Online]. Available: https://www.aclweb.org/anthology/D14-1162.pdf

[22] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching Word Vectors with Subword Information,” *arXiv [cs.CL]*, 15-Jul-2016 [Online]. Available: http://arxiv.org/abs/1607.04606

[23] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *arXiv [cs.LG]*, 22-Dec-2014 [Online]. Available: http://arxiv.org/abs/1412.6980

[24] J. Zhang *et al.*, “Why ADAM Beats SGD for Attention Models,” *arXiv [math.OC]*, 06-Dec-2019 [Online]. Available: http://arxiv.org/abs/1912.03194

[25] R. J. Williams and D. Zipser, “A Learning Algorithm for Continually Running Fully Recurrent Neural Networks,” *Neural Comput.*, vol. 1, no. 2, pp. 270–280, Jun. 1989, doi: 10.1162/neco.1989.1.2.270. [Online]. Available: https://doi.org/10.1162/neco.1989.1.2.270

[26] “9.8. Beam Search — Dive into Deep Learning 0.14.0 documentation.” [Online]. Available: https://d2l.ai/chapter_recurrent-modern/beam-search.html. [Accessed: 08-Jul-2020]

[27] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *J. Mach. Learn. Res.*, vol. 15, no. 56, pp. 1929–1958, 2014 [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html. [Accessed: 07-Jul-2020]

[28] L. Breiman, “Bagging predictors,” *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996, doi: 10.1007/BF00058655. [Online]. Available: https://doi.org/10.1007/BF00058655

[29] R. Caruana, S. Lawrence, and C. L. Giles, “Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping,” in *Advances in Neural Information Processing Systems 13*, T. K. Leen, T. G. Dietterich, and V. Tresp, Eds. MIT Press, 2001, pp. 402–408 [Online]. Available:

http://papers.nips.cc/paper/1895-overfitting-in-neural-nets-backpropagation-conjugate-gradient-and-early-stopping.pdf
[30] I. Goodfellow, Y. Bengio, and A. Courville, “Regularization for deep learning,” *Deep learning*, pp. 216–261, 2016 [Online]. Available: https://mnassar.github.io/deeplearninghandbook/slides/07_regularization.pdf
[31] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima,” *arXiv [cs.LG]*, 15-Sep-2016 [Online]. Available: http://arxiv.org/abs/1609.04836
[32] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL ’02*, Philadelphia, Pennsylvania, 2001, p. 311, doi: 10.3115/1073083.1073135 [Online]. Available: http://portal.acm.org/citation.cfm?doid=1073083.1073135
[33] C.-Y. Lin, “Rouge: A package for automatic evaluation of summaries,” in *Text summarization branches out*, 2004, pp. 74–81 [Online]. Available: https://www.aclweb.org/anthology/W04-1013.pdf
[34] “rouge-score,” *PyPI*. [Online]. Available: https://pypi.org/project/rouge-score/. [Accessed: 07-Jul-2020]
[35] J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization,” *J. Mach. Learn. Res.*, vol. 13, no. null, pp. 281–305, Feb. 2012 [Online]. Available: https://dl.acm.org/doi/abs/10.5555/2188385.2188395
[36] L. van der Maaten and G. Hinton, “Visualizing Data using t-SNE,” *J. Mach. Learn. Res.*, vol. 9, no. Nov, pp. 2579–2605, 2008 [Online]. Available: http://www.jmlr.org/papers/v9/vandermaaten08a.html. [Accessed: 09-Jul-2020]
[37] drivendata.org, “Cookiecutter Data Science Project Structure.” 2020 [Online]. Available: https://drivendata.github.io/cookiecutter-data-science/
[38] S. Van Der Walt, S. Chris Colbert, and G. Varoquaux, “The NumPy array: a structure for efficient numerical computation,” *arXiv [cs.MS]*, 08-Feb-2011 [Online]. Available: http://arxiv.org/abs/1102.1523
[39] W. Mckinney, “pandas: a Foundational Python Library for Data Analysis and Statistics,” Jan. 2011 [Online]. Available: http://dx.doi.org/. [Accessed: 07-Jul-2020]
[40] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in Python,” *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011 [Online]. Available: http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf
[41] J. D. Hunter, “Matplotlib: A 2D Graphics Environment,” *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, May 2007, doi: 10.1109/MCSE.2007.55. [Online]. Available: https://aip.scitation.org/doi/abs/10.1109/MCSE.2007.55
[42] T. Kluyver *et al.*, “Jupyter Notebooks -- a publishing format for reproducible computational workflows,” in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 2016, pp. 87–90.
[43] M. Abadi *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th ${USENIX} symposium on operating systems design and implementation ({OSDI}$ 16)*, 2016, pp. 265–283 [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi
[44] L. Biewald, “Experiment Tracking with Weights and Biases.” 2020 [Online]. Available: https://www.wandb.com/
[45] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*, 1st ed. O’Reilly Media, Inc., 2014 [Online]. Available:

https://dl.acm.org/citation.cfm?id=2621997
[46] M. Rabinovich, M. Stern, and D. Klein, “Abstract Syntax Networks for Code Generation and Semantic Parsing,” *arXiv [cs.CL]*, 25-Apr-2017 [Online]. Available: http://arxiv.org/abs/1704.07535