Bachelor Thesis

August 4, 2019

Suggesting Meaningful Method Names

Analysing Source Code using Deep Learning Techniques

Yves Rutishauser

of Zürich, Schweiz (16-701-658)

supervised by

Prof. Dr. Harald C. Gall

Adelina Ciurumelea

University of Zurich^UZH

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

Bachelor Thesis

Suggesting Meaningful Method Names

Analysing Source Code using Deep Learning Techniques

Yves Rutishauser

University of Zurich^UZH

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

Bachelor Thesis

Author: Yves Rutishauser, yves.rutishauser@uzh.ch

Project period: 21.3.2019 - 21.9.2019

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

Abstract

Good identifier names provide a high-level summary of source code and are therefore beneficial during software maintenance. Hence, automatically suggesting descriptive and accurate method names reduces time spent maintaining code and improves understandability and readability of software corpora. Since source code has many similar properties to natural language many models originally developed for Natural Language Processing (NLP) are successfully applied to code. In this thesis, I propose 2 different approaches and experiment with a total of 6 models that are specifically adjusted to solve the method naming problem. These models learn to assign tokens to locations (embeddings) such that tokens with similar meanings have similar embeddings. Based on the combination of these embeddings, I can suggest accurate method names. I demonstrate that models are more effective if partially trained on the current project than models that predict on projects completely unobserved during training. Furthermore, I show the effectiveness of splitting a method name into sub-tokens. These models can predict neologisms (names that are not in the vocabulary). In a quantitative analysis, I compare the different models and approaches with different metrics. I furthermore adapt a metric, which is specifically designed for this task and has been used in the past. Additionally, I evaluate the models with different input parameters and show the effectiveness of using the type, parameters, and the method body to suggest its name. In a qualitative analysis, I discuss 8 different use cases, demonstrate visualizations and show the limits of the proposed models.
The code and data are available on Github [12] and Zenondo [56].

Zusammenfassung

Gute Identifier-Namen bieten eine kurze Zusammenfassung des Quellcodes und sind daher bei der Softwarewartung von Vorteil. Das automatische Vorschlagen von deskriptiven und akkuraten Methodennamen reduziert den Zeitaufwand der Codepflege und verbessert die Verständlichkeit und Lesbarkeit von Software. Da der Quellcode viele ähnliche Eigenschaften wie die natürliche Sprache aufweist, werden viele ursprünglich für Natural Language Processing (NLP) entwickelten Modelle erfolgreich auf Software angewendet. In dieser Arbeit schlage ich 2 verschiedene Ansätze vor und experimentiere mit insgesamt 6 Modellen, die speziell angepasst sind, um Methodennamen vorzuschlagen. Diese Modelle lernen, sogenannte Tokens an Locations (Embeddings) so zuzuordnen, dass Tokens mit ähnlicher Bedeutung ähnliche Embeddings haben. Basierend auf der Kombination dieser Embeddings kann ich akkurate Methodennamen vorschlagen. Ich zeige, dass Modelle effektiver sind, wenn sie teilweise auf dem aktuellen Projekt trainiert worden sind, als Modelle, die bei Projekten vorhersagen, die während des Trainings unbeobachtet bleiben. Darüber hinaus zeige ich die Effektivität der Aufteilung eines Methodennamens in sogenannte Sub-Tokens auf. Diese Modelle können Neologismen vorhersagen (Namen, die nicht im Vokabular enthalten sind). In einer quantitativen Analyse vergleiche ich die verschiedenen Modelle und Ansätze mit unterschiedlichen Messgrößen. Außerdem adaptiere ich eine Metrik, die speziell für diese Aufgabe entwickelt wurde und in der Vergangenheit verwendet wurde. Zusätzlich bewerte ich die Modelle mit verschiedenen Inputparametern und zeige die Effektivität der Verwendung des Typs, der Parameter und des Methodeninhalts, um den entsprechenden Namen vorzuschlagen. In einer qualitativen Analyse liste ich 8 verschiedene Anwendungsfälle auf, demonstriere Visualisierungen und zeige die Grenzen der vorgeschlagenen Modelle auf.
Der Code und die Daten sind auf Github [12] und Zenondo [56] verfügbar.

Contents

1 Introduction **1**

2 Related Work **5**
- 2.1 Identifier Names 5
- 2.2 Language Models for Source Code 5

3 Background **7**
- 3.1 Language Models 7
  - 3.1.1 Representation of Words 7
  - 3.1.2 Feed-forward Neural Networks 9
  - 3.1.3 Recurrent Neural Networks 10
- 3.2 Sequence-to-Sequence Model 12
  - 3.2.1 Encoder 13
  - 3.2.2 Decoder 13
  - 3.2.3 Sequence-to-Sequence with Attention 14
  - 3.2.4 Training 15
  - 3.2.5 Inference 15

4 Data **17**
- 4.1 Dataset 17
  - 4.1.1 Allamanis Dataset 17
  - 4.1.2 Java-small 17
- 4.2 Parsing 17
- 4.3 Data Preprocessing 18
  - 4.3.1 Rare Tokens 19
- 4.4 Statistics on the Data 20

5 Approach **23**
- 5.1 Models 23
  - 5.1.1 Token Based Models 23
  - 5.1.2 Sub-token Based Models 26

6 Evaluation **29**
- 6.1 Predicting Identifier Names using Neural Networks 29
  - 6.1.1 Evaluation of Token based Models 29
  - 6.1.2 Evaluation of Sub-token Based Models 32
- 6.2 Token Models vs. Sub-token Models 33
  - 6.2.1 Qualitative Analysis 34

6.3 Different Input Parameters 43

7 Threat to Validity **45**
7.1 Out-of-Vocabulary Tokens 45
7.2 Quality of Training Data 46
7.3 Splitting of Training and Test Set 46

8 Implementation Details **47**
8.1 Implementation of the Token Approach 47
8.2 Implementation of the Sub-token Approach 48

9 Conclusion **51**

Chapter 1

Introduction

Maintainability of source code is important. Developers spend the majority (80%) of their time maintaining and understanding code. As stated by Lawrie et al. [41], readers of programs have two main sources of comprehending a piece of code: either by its comments or by its identifier names. Because a lot of code is uncommented the importance of consistent and understandable naming is high. Furthermore, Allamanis et al. [21] state that one-third of code reviews contain feedback about coding conventions, indicating the programmers might not always follow them and that team members deeply care about adherence.
Method names are especially important because "methods are the smallest named units of aggregated behavior in most conventional programming languages and hence the cornerstone of abstraction" [52]. Furthermore, if a method is used in an API, its name heavily matters, making the library irrelevant when the functions are poorly named [22]. A method name must be capable of giving a short summary of its functionality. But this is not an easy task: "A suitable name must not only describe what a method is, but also what it does" [22].
Programming Language Processing [45] is a relatively new research field that aims to model, understand and process source code. Allamanis et. al [19] formulated the "naturalness" hypothesis which states that software is not only an instruction for the computer but also a form of communication between developers. Thus, the aim is to provide a fluent communication between different developers. Additionally, if developers can understand source code snippets written by someone else more easily, it also reduces maintainability costs, which is an important factor of source code projects. Similarly to Natural Language Processing (NLP), Programming Language Processing aims to find a model that accurately represents the meaning of a sequence of words or tokens.
The problem in this work can be framed as a translation problem, that given an input (method body, parameters, return type) the goal is to predict its method name. I propose two approaches to tackle this matter. For the token approach, a code snippet is split in a set of tokens. The models then assign each identifier to a low-dimensional vector space. I process these embeddings with a recurrent unit (GRU, LSTM) to produce a final vector where I derive the final prediction. The sub-token approach allows the model to suggest neologisms, i.e. method names it has not seen during training. With this approach, I further split identifiers into sub-tokens (e.g. "setCancelled" is transformed to "set", "Cancelled") [22]. This allows me to apply state-of-the-art neural translation models (Sequence-to-Sequence and Sequence-to-Sequence with Attention) to the task. In contrast to the token models, the sub-token models predict a sequence of sub-tokens.

Motivating Task: Table 1.1 presents a code snippet along with the prediction of the models. A human reading the code can easily label the code snippet as "setCancelled". The goal of this thesis is to predict method names automatically. In this case, all the trained models in this work are able to infer the correct method name. I furthermore present a visualization of the attention weights by the most sophisticated model used in this work. The heatmap describes how the model derives

	Motivating Task

Code Snippet	`public void f(boolean cancel) {`
	`    this.cancelled = cancel;`
	`}`

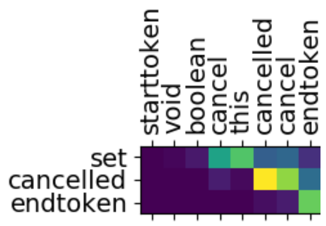
Ground truth	setCancelled
Prediction	setCancelled
Attention Weights	

Table 1.1: Motivating Task: a code snippet, the ground truth, the models' prediction and the distribution of attention weights

the method name. For example, it uses the argument "cancel" and the token "this" to infer "set". Additional and more complex examples are presented in the qualitative analysis section.

Goal: The main contributions of the thesis are as follows:

- I frame the method naming problem as a text generation task and evaluate the effectiveness of the token and sub-token approach on 2 different datasets. I show that a deep learning model is much more powerful if partially trained on the source code project it later suggests method names.
- I compare the token and sub-token models and illustrate that the latter are especially more useful if evaluated on source code projects it has not been trained on.
- I demonstrate what kind of context information is needed to successfully predict identifier names. I confirm that models are more effective if the input is composed of both the parameters and the method body.
- I present a qualitative analysis to illustrate what the models can and cannot predict. Furthermore, I show interesting predictions made by the sub-token models that are only partially correct. I also present a visual interpretation of the attention weights for the Sequence-to-Sequence with Attention Sub-token Model.

Use Cases: The models in this work can be embedded in multiple ways. Consider the example of an IT company which maintains a huge software with millions of lines of code (e.g. a banking software). The model in this work could be extended such that it highlights method names that do not capture the semantics of its functionality. Furthermore, the model could be partially trained on the project to comprehend its coding conventions and later be used to predict method names of new functions. Such a tool could reduce time for maintainability and would increase understandability and readability of the project's source code.

Organization: The remainder of the thesis is organized as follows. Chapter 2 presents relevant related work. Chapter 3 provides the background materials for the deep learning models. Chapter 4 discusses the data used in this work. Chapter 5 elaborates the details of the models and the proposed approach. In Chapter 6, I evaluate the models with a quantitative and qualitative analysis. Chapter 7 explains the implementation details. Finally, Chapter 8 concludes the thesis.

Chapter 2

Related Work

Coding conventions are an important aspect of software engineering. They boost readability and therefore maintainability of source code. Hence, coding conventions are standard practice [29]. Source code consists of 70% identifier names, whereas good identifier names improve readability and comprehension of source code [21]. In a survey, Arnaoudova et. al [27] found that 68% of 94 developers consider suggestions for identifier renaming useful. Such a tool could help in reviewing code automatically to suggest better method names when it doesn't match with its functionality [26]. First, I will review related work in naming identifiers, then I will discuss related applications.

2.1 Identifier Names

In prior work, *n*-gram language models have been tested with source code. *n*-grams count the occurrence of consecutive words and the frequency in which they appear. But these models do not have the capacity to learn representations that generalize beyond the training data and are quickly overwhelmed by the curse of dimensionality [55]. Allamanis et. al [20] suggested the naturalness hypothesis where software is a form of human communication and not simply a command for the computer. If this hypothesis holds, software corpora can be treated similar to language and in consequence natural language machine learning models can be applied to analyze source code [20]. Allamanis et. al [22] propose a language model which suggests method and class names by embedding them in a low dimensional continuous space. Furthermore, Allamanis et. al [22] were the first to come up with a sub-token model for the method naming task. With this approach, a method name is further split into sub-tokens (e.g. toLowerCase -> to Lower Case). But their work omits the syntactical structure of source code. Hence Alon et. al [26] suggest decomposing a code snippet into a collection of paths in its abstract syntax tree to capture their semantic meaning. Allamanis et al. [23] represent software corpora with gated graph neural networks. The current state-of-the-art is proposed by Alon et. al [25]. This work suggests a novel neural attention model that leverages the syntactical structure of source code and decomposes tokens into sub-tokens.

2.2 Language Models for Source Code

Identifier (class, method and variable) names can be seen as an extreme summary of its functionality [22]. Similarly, code comments provide a natural language summary of a specific source code

snippet. Therefore, a similar task is to predict code comments given a source code snippet. Iyer et. al [40] used a neural attention model to generate high-level summaries of software corpora. Wan et. al [53] improved the current state-of-the-art by encoding source code as a tree structure and used deep reinforcement learning during decoding.
Oda et al. [46] generate pseudo-code from source code using machine translation techniques. The goal here is to produce a more readable version of source code.
Furthermore, machine learning-based approaches have been used to find code defects. These models assign probabilities to code snippets. Ray et. al [47] found that buggy code tends to be less natural and therefore less probable. A difficulty here is to distinguish between code that is just rare behavior and code that is actually buggy. Campbell et. al [30] used *n*-grams to localize errors and suggest a fix.
Deep learning approaches have also been applied to code translation [20]. Recently Chen et. al [31] tried to translate source code to another language by using a tree2tree neural translation model. This model uses an encoder-decoder architecture to translate a sub-tree (e.g. source code snippet in Java) to another sub-tree (e.g. source code snippet in Python).
A big problem during developing is that developers frequently copy code. This behavior results in code clones where the same code snippet appears at different locations [20]. White et al. [54] use recurrent neural networks to detect clones. Similar to this work, they use distributed vector representations which is able to generalize well [20].

Chapter 3

Background

3.1 Language Models

The main goal of this thesis is to predict the method name given the corresponding method type, parameters, and body while obeying the style conventions of the current project. We can frame this as a text generation problem. Similarly, I want to predict token n given some context (e.g. tokens $n-2$, $n-1$, $n+1$, $n+2$). Thus, the objective is similar to problems in natural language processing (NLP). Therefore, I will use models in this work that have seen a wide range of success in NLP, e.g. in speech recognition, machine language translation, and chatbots. An important aspect of NLP is a meaningful representation of words that serves as the first layer of each model. Hence, I will first review popular representations of words, then I will move on to the models used in this work.

3.1.1 Representation of Words

In this section, I will discuss how words are represented in neural language models. The aim is to find an efficient representation of words and a representation such that words with similar semantic meaning have similar representations.

One-hot Encoded Vectors

Neural networks work with matrices of numbers because they cannot handle words. Therefore we need to encode each unique word in a set of inputs as a one-hot encoded vector. Consider the following 2 input sequences (adapted from [5]):

- "Have a good day"
- "Have a great day"

Therefore, vocabulary V consists of:

- V = {Have, a, good, great, day}

First, each word in the vocabulary is assigned to an index. In this example, this leads to:

- have = 1
- a = 2
- good = 3

- great = 4
- day = 5

The next step is to map the words in the vocabulary to one-hot encoded vectors. The size for each word vector is the size of vocabulary $|V|$, in this case the size is 5. These vectors consist of all zeros and one 1. The position of the 1 is defined according to its index. In the provided example that would lead to the following vectors (where T represents the transpose):

- have = $[1, 0, 0, 0, 0]^T$
- a = $[0, 1, 0, 0, 0]^T$
- good = $[0, 0, 1, 0, 0]^T$
- great = $[0, 0, 0, 1, 0]^T$
- day = $[0, 0, 0, 0, 1]^T$

This approach suffers from the following 2 drawbacks:

1. The length of the one-hot encoded vectors grows with the vocabulary size $|V|$. That means if we add another word like "night" to the vocabulary, this would lead to words vectors of size 6. In a very large vocabulary, this leads to very high-dimensional, sparse vectors.
2. It is not possible to show similarities between words. In this example "great" and "good" are as different as "day" and "good".

Distributed Word Vectors

Distributed word vectors were first introduced by Bengio et. al [28] over a decade ago. Similarly to one-hot encoded vectors, words are assigned to real-valued vectors. But in contrast to one-hot encoded vectors, where each word in vocabulary V occupies 1 dimension, words in word embedding methods are mapped to a low-dimensional space (the dimensions are in the size of hundreds as opposed to the one-hot encoded vectors). The function $V : \text{Vocabulary} \rightarrow \mathbb{R}^n$ maps the words to a fixed high-dimensional space n which is independent of vocabulary size $|V|$. The size of the embedding can be freely determined, common values are a power of 2, such as 64, 128, 256. Just for illustration purposes, I will use $n = 3$ here to demonstrate that this method usually results in fewer dimensions than the corresponding one-hot encoded vectors. In the example introduced in the last section and for $n = 3$ that leads to the following vectors (where ^{T} represents the transpose):

- have = $[0.2, 0.3, -0.4]^T$
- a = $[0.1, 0.2, 0.9]^T$
- good = $[0.8, -0.3, 0.5]^T$
- great = $[0.7, 0.5, 0.8]^T$
- day = $[0.1, 0.3, -0.4]^T$

The vectors are initialized with small random values. These values are then learned by the model to have meaningful vectors according to its task jointly with the other parameters of the model using backpropagation [28]. A multiplication of vocabulary size $|V|$ and dimension n of each feature vector determines the number of trainable parameters for the first layer of the model.

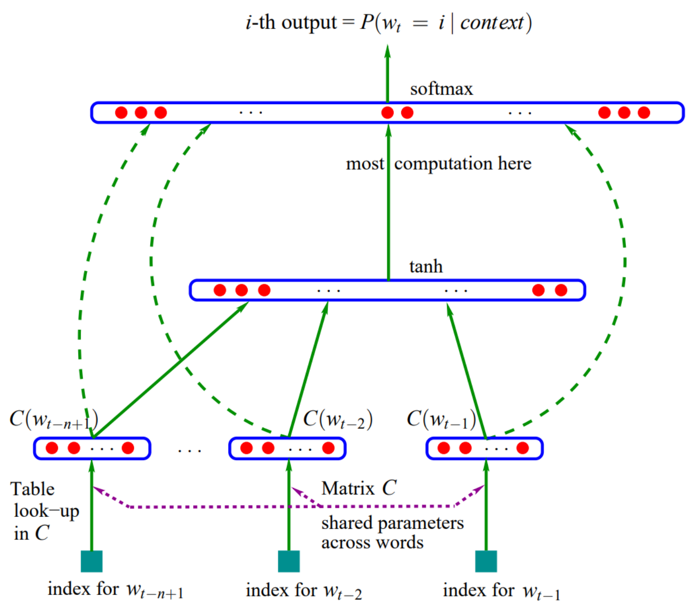


Figure 3.1: feed-forward neural network architecture (adapted from [28])

In contrast to one-hot encoded vectors, where each word is independent of any other word in the vocabulary, distributed word vectors learn to group similar words. Word vectors have the ability to generalize because similar words tend to have similar word vectors. A famous example shows the power of distributed word vectors: In this vector space, "King - Man + Woman" results in a vector very close to "Queen". In this example, "good" and "great" would end up with a similar vector representation. Word embeddings can either be obtained from pre-trained embeddings (e.g. word2vec [44]) or can be jointly learned together with the other parameters by the model.

3.1.2 Feed-forward Neural Networks

A simple feed-forward neural network architecture can be seen in figure 3.1. First, the input words are assigned to distributed word vectors, described in the last section. After, the embedded word vectors $C(i)$ are concatenated:

$$x = (C(w_{t-1}), C(w_{t-2}), ..., C(w_{t-n+1})) \tag{3.1}$$

Based on the concatenation, the hidden layer of the neural network computes:

$$z = tanh * (d + Hx) \tag{3.2}$$

where hyperbolic tangent tanh is applied element by element and pushes all values between -1 and 1. The last layer is computed as follows:

$$y = b + Uz \tag{3.3}$$

The neural network computes a softmax function as an output, which guarantees the summation to 1:

$$\widetilde{P}(w_t | w_{t-1}, w_{t-2}, ..., w_{t-n+1}) = \frac{e^{y_{wt}}}{\sum_i e^{y_i}} \tag{3.4}$$

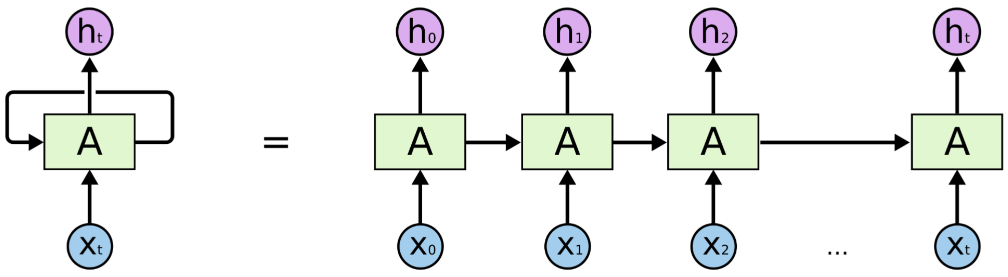


Figure 3.2: An unrolled recurrent neural network (adapted from [49])

Therefore, for each word in the target vocabulary the softmax function computes a probability given its input. During training, the model adjusts the parameters of the model (b, d, W, U, H, C) through backpropagation [22].

3.1.3 Recurrent Neural Networks

In feedforward neural networks feature vectors are processed in one go [34]. That means you show the whole sequence to the network and turn it into a single data point. This refers to equation 3.1. The goal here is to find a better representation of an input sequence other than just a concatenation of the input words.
Recurrent neural networks (RNNs) process the input sequence by looping through the input elements while keeping a state of what came before [34]. This way, the model is able to not only process each word independently but also compute a hidden state that is a combination of its internal meaning and its history. More formally, for a timestep t, the hidden state is computed as follows:

$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t) \tag{3.5}$$

Figure 3.2 shows an illustration of a recurrent neural network architecture. The disadvantages of RNNs are exploding or vanishing gradients during backpropagation [39]. Long Short-Term Memories (LSTMs) or Gated Recurrent Units (GRUs) are prominent solutions to these problems.

Long Short-Term Memory

The Long Short-Term Memory cell was developed by Schmidhuber and Hochreiter and is especially designed to learn long-term dependencies [39]. Similar to RNNs, LSTMs also have a chain like structure but in contrast to the simple single interactions between the modules, an LSTM is connected through 4 different ways with the previous module.
At each time step t, there is an input gate (3.6), a forget gate (3.7) an output gate (3.8), a new

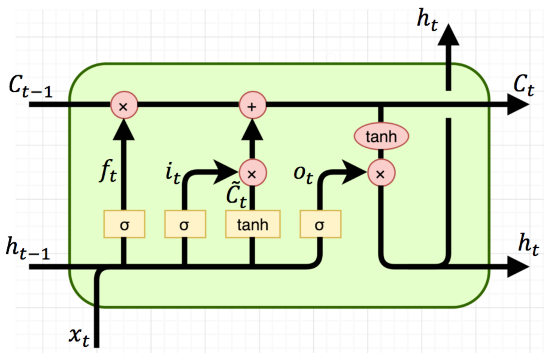


Figure 3.3: An LSTM (adapted from [8])

memory cell (3.9), a final memory cell (3.10) and a final hidden state (3.11).

$$i_t = \sigma\left(x_t U^i + h_{t-1} W^i\right) \tag{3.6}$$
$$f_t = \sigma\left(x_t U^f + h_{t-1} W^f\right) \tag{3.7}$$
$$o_t = \sigma\left(x_t U^o + h_{t-1} W^o\right) \tag{3.8}$$
$$\tilde{C}_t = \tanh\left(x_t U^g + h_{t-1} W^g\right) \tag{3.9}$$
$$C_t = \sigma\left(f_t * C_{t-1} + i_t * \tilde{C}_t\right) \tag{3.10}$$
$$h_t = \tanh(C_t) * o_t \tag{3.11}$$

An illustration of the LSTM can be seen in figure 3.3. Equations 3.6-3.9 can be seen as single layer neural networks. The input gate looks at the previous hidden state and the current input and outputs a number between 0 and 1, using a sigmoid nonlinearity function. The input gate describes how much we care about the current vector, this refers to equation 3.6 and i_t in figure 3.3.
The forget gate looks at the previous hidden state and the current input and outputs a number between 0 and 1 again using a sigmoid nonlinearity function. Counterintuitively, the output 0 refers to completely get rid of the last hidden state (forget the past) while the output 1 refers to completely keep the last hidden state, this refers to equation 3.7 and f_t in figure 3.3.
The output gate describes how much the cell is exposed. It serves as a filter, describing what parts of the cell state is important for the current hidden timestep. This enables the mechanism that the current cell may not be important for the current hidden timestep, but may become important in the future, this refers to equation 3.8 and o_t in figure 3.3. Equation 3.9 creates a new candidate vector which could be added to the cell state ($\tilde{C}_t$ in figure 3.3). After, equation 3.10 creates the new cell states based on how much to forget of the previous cell state and how much to keep around from the current candidate vector (C_t in figure 3.3). Finally, the current hidden state is computed, which is a multiplication of the normalized current cell state and the output gate, this refers to equation 3.11 and h_t in figure 3.3.

To summarize, the LSTM introduces a memory cell which can selectively forget information from past hidden states. This memory cell is able to preserve longer dependencies than by using simple recurrent neural networks [14].

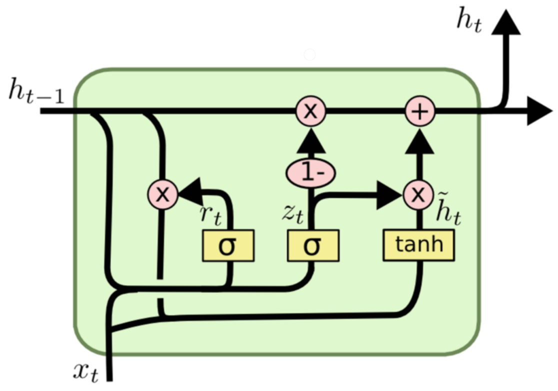


Figure 3.4: A Gated Recurrent Unit (adapted from [8])

Gated Recurrent Unit

GRUs are very similar to LSTMs but proposed more recently [32]. In contrast to LSTMs GRUs don't use a cell state but use the hidden state to transfer information [32]. Also unlike the LSTM which uses 3 gates, the GRU uses 2 gates: a reset gate and an update gate. At each time step, there is a reset gate (3.12), an update gate (3.13), a new memory content (3.14) and a final memory (3.15).

$$r_t = \sigma\left(x_t U^r + h_{t-1} W^r\right) \tag{3.12}$$
$$z_t = \sigma\left(x_t U^z + h_{t-1} W^z\right) \tag{3.13}$$
$$\tilde{h}_t = \tanh\left(x_t U^h + (r_t * h_{t-1}) W^h\right) \tag{3.14}$$
$$h_t = z_t * h_{t-1} + (1 - z_t) * \tilde{h}_t \tag{3.15}$$

An illustration of the GRU can be seen in figure 3.4. The reset gate describes how to combine the new input with the previous memory. If the reset gate is close to 0, it ignores the previous memory and only keeps information about the new word. This refers to equation 3.12 and r_t in figure 3.4. The update gate controls how much of the past should matter now. If the update gate is close to 1 then we copy information of the previous memory to the current memory. This refers to equation 3.13 and z_t in figure 3.4. The new memory is computed similarly to the one of traditional RNNs (equation 3.14 and $\tilde{h}_t$ in figure 3.4). Lastly, the final memory h_t is a linear interpolation between the previous final memory h_{t-1} and the new memory content $\tilde{h}_t$ controlled by the update gate z_t (equation 3.15 and h_t in figure 3.4). GRUs, therefore, have fewer parameters to adjust during backpropagation while its performance is often comparable to LSTMs [35].

3.2 Sequence-to-Sequence Model

The Sequence-to-Sequence model has seen a lot of success in the past few years, for example in speech recognition, machine language translation, and chatbots [42]. The architecture of the model was first introduced by Cho et. al [33] as a statistical machine translation model.
The goal of the Sequence-to-Sequence model is to estimate the conditional probability of the target sequence y given an input sequence x. So given the source sentence x = *I am a student* the model should assign a high probability to the target sentence y = *Je suis étudiant* [42]. The model

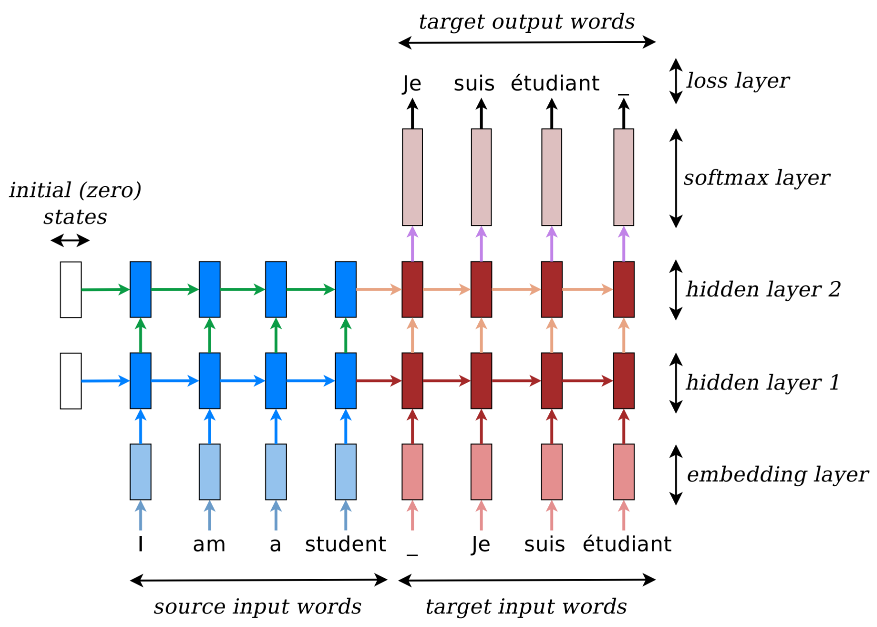


Figure 3.5: An example of a Sequence-to-Sequence Model (adapted from [42])

achieves this by using an encoder and a decoder, where the encoder and decoder are essentially both made up of recurrent neural networks, more precisely mostly the LSTM architecture introduced in section 3.1.3. An example of the model can be seen in figure 3.5.

3.2.1 Encoder

The encoder works the same as the LSTM described in section 3.1.3. Each input (a token) is embedded using a feature vector and processed with an LSTM. The last hidden state and the last cell state are important, they serve as a representation for the entire input sequence.

3.2.2 Decoder

The decoder is used to generate a translation, one target word at a time. This is done by conditioning every hidden state h_t in the decoder based on the previous hidden state h_{t-1}, the last hidden vector of the encoder c and the previous predicted output word y_{t-1}. Hence, for a time step t, the decoder of the hidden state is computed by,

$$h_t = f(h_{t-1}, c, y_{t-1}) \tag{3.16}$$

After the hidden state for time step t is found, similarly, the appropriate word y_t is computed by,

$$y_t = \text{softmax}(h_t, c, y_{t-1}) \tag{3.17}$$

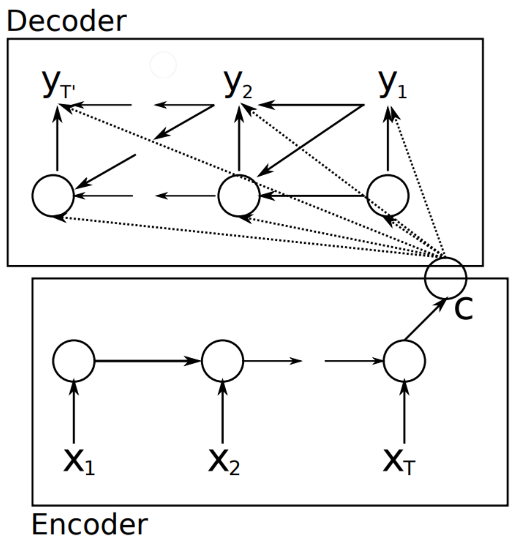


Figure 3.6: An illustration of the Sequence-to-Sequence Model (adapted from [33])

The softmax function computes a valid probability distribution over a target vocabulary y conditioned on the hidden state h_t, the last hidden state of the encoder c and the previous target word y_{t-1}. This explanation can be followed by looking at the arrows in the decoder in figure 3.6.

3.2.3 Sequence-to-Sequence with Attention

In a regular encoder-decoder architecture, the model processes the input words one by one and computes a final vector representation of the sentence. Based on this representation the decoder extracts each word one by one using another RNN (usually an LSTM or a GRU) [1]. While this works well for relatively short sequences, it is hard for the model to get a good input representation if the sequence is longer. A more sophisticated version of the Sequence-to-Sequence model uses an attention mechanism during decoding. Using attention, the model is able to selectively focus on certain parts of the input [1]. Figure 3.7 demonstrates the attention mechanism during decoding. During decoding the word "Je", the model mostly concentrates on the source word "I" . In a Sequence-to-Sequence Attention model, we first hand each hidden state computed during encoding and the final hidden state to the decoder. During decoding and at each time step t, a score function is computed that compares the last target hidden state h_{t-1} with each source hidden state $e_{t'}$ [9]. The score function is a scalar where a high number means that for a current time step t, this source hidden state is important.

$$score(h_{t-1}, e_{t'}) = \begin{cases} h_{t-1}^T e_{t'} & \text{dot product} \\ h_{t-1}^T W e_{t'} & \text{Luong's multiplicative style} \\ v^T \tanh(W[h_{t-1}, e_{t'}]) & \text{Bahdanau's additive style} \end{cases} \tag{3.18}$$

After each score is obtained, a softmax function is applied to compute weighted values for the attention distribution:

$$\alpha_{ts} = \text{softmax}(score(h_{t-1}, e_{t'})) \tag{3.19}$$

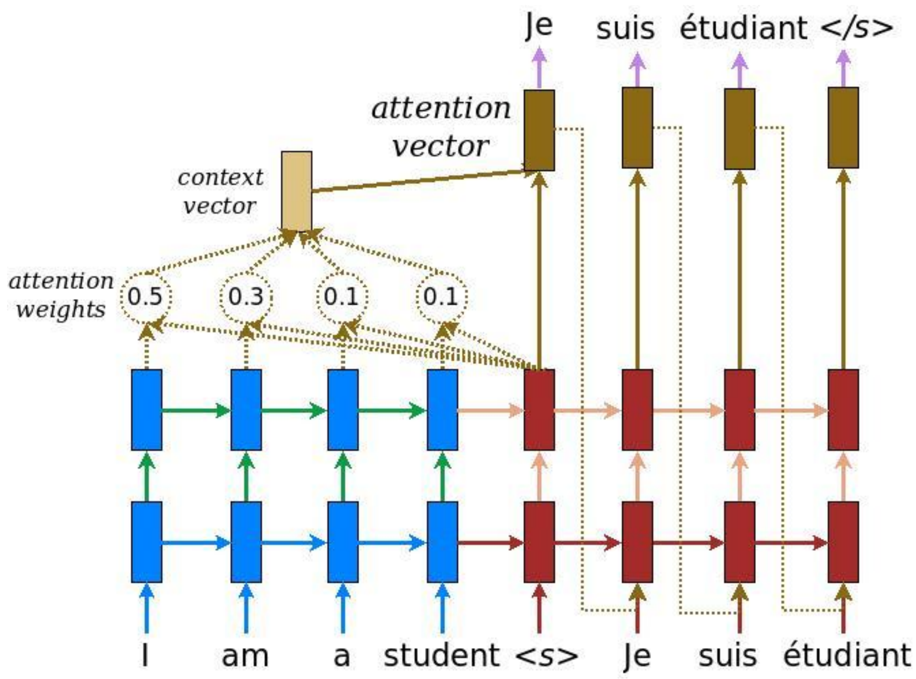


Figure 3.7: An example of a Sequence-to-Sequence Model with Attention (adapted from [9])

The context vector c_t is then computed by summing up all weighted input hidden states:

$$c_t = \sum_{t'=0}^{n} \alpha_{ts} e_{t'} \tag{3.20}$$

After, the context vector c_t is concatenated with the previous target embedding y_{t-1} and processed with an LSTM or a GRU [11]. Therefore, for a time step t, the GRU is computed as follows:

$$h_t = \text{GRU}\left(h_{t-1}, [y_{i_{t-1}}, c_t]\right) \tag{3.21}$$

Next, the hidden state h_t is passed to an additional layer s_t and finally a softmax function is applied to generate a valid probability distribution over the target vocabulary:

$$s_t = g(h_t) \tag{3.22}$$
$$p_t = \text{softmax}(s_t) \tag{3.23}$$

3.2.4 Training

During decoding of a sequence, the current word to be predicted y_t heavily depends on the previous predicted word y_{t-1}. But we cannot always rely that the decoder's guess is the ground truth. So during training, an important concept used is "teacher forcing" [16]. This concept uses the real target output as the next input instead of the decoder's guess.

3.2.5 Inference

During testing/ inference, we cannot use "teacher forcing" because the model doesn't know the ground truth. The simplest approach to solve this problem is to use greedy decoding by feeding to the next step the most likely word predicted in the previous step. But if the model then makes a small error by predicting a wrong word in the sequence, this would mess up the whole decoding. Beam search is a better way of decoding: Instead of just keeping the best sequence generated so far, beam search keeps track of the current top k sequences generated so far. More formally,

$$\mathcal{H}_t := \{(w_1^1, \ldots, w_t^1), \ldots, (w_1^k, \ldots, w_t^k)\} \tag{3.24}$$

Consider the example from figure 3.5. If $k = 3$ and time step $t = 2$, one possible $\mathcal{H}_2$ could be:

$$\mathcal{H}_2 := \{(\text{Je suis}),\ (\text{Mois suis}),\ (\text{Mois est})\} \tag{3.25}$$

For $k = 3$, beam search instantiates 3 copies of the network and searches the top 3 next words based on each sequence ending up with a set of 9 sequences (candidate C_3).

$$\begin{aligned} C_3 =&\{(\text{Je suis étudiant}),\ (\text{Je suis étudier}),\ (\text{Je suis suis})\} \\ &\cup\{(\text{Mois suis étudiant}),\ (\text{Mois suis est}),\ (\text{Mois suis étuidié}) \\ &\cup\{(\text{Mois est étudiant}),\ (\text{Mois est étuidié}),\ (\text{Mois est suis})\} \end{aligned}$$

From the set of candidates C_3 the most probable 3 sequences are kept. In this example a possible $\mathcal{H}_3$ could be:

$$\mathcal{H}_3 := \{(\text{Je suis étudiant}),\ (\text{Je suis étudier}),\ (\text{Mois est étudiant})\} \tag{3.26}$$

This search continues until the end of the translation is reached. Then, the translation with the highest probability is returned [11].

Chapter 4

Data

In this chapter, I present the different datasets used in this work. First, I will explain how the projects were parsed and processed. I will also include statistics on methods.

4.1 Dataset

In this work, I use 2 different datasets to train, validate and test the models. They are deliberately chosen to be of various sizes and composed differently.

4.1.1 Allamanis Dataset

The dataset consists of 20 active open-source software projects [22]. These projects have been chosen because they have been used in prior work by Allamanis et. al [22] and considered to follow good naming practices. For each project, 60% of the methods are used for training, 10% for validation and 30% for testing [22].

4.1.2 Java-small

Java-small contains about $700K$ examples and was first used by Allamanis et. al [24]. It is composed of 11 relatively large projects. 9 of these projects are used for training, 1 for validation and 1 for testing. Therefore, I train across multiple source code projects but predict on projects the model has not seen before similar to Alon et. al [25].

4.2 Parsing

In a first attempt, the models have been built using a preprocessed version of the Allamanis Dataset provided by Allamanis et. al [22]. Each Java file was processed and in a separate column, all the method names were indexed. It was hard to extract the return type, parameters and method body of this preprocessed version precisely, therefore the original files have been parsed again using a Java parser [6]. It was difficult to build a Java parser that is able to collect method names of all the Java projects without resulting in stack-over-flow or out-of-memory errors. Because of the size of the dataset, it was necessary to use multiprocessing. Also, to avoid out-of-memory errors, the methods are saved in chunks of a maximum of 100k methods per file. A sample of a Java method can be seen in listing 4.1 and its processed version in table 4.1. During parsing, comments by developers inside method bodies have been removed.

```java
@Override
  public void channelRead(ChannelHandlerContext ctx, Object msg) {
    if (acceptMessage(msg)) {
      log((SpdyFrame) msg, Direction.INBOUND);
    }
    ctx.fireChannelRead(msg);
  }
```

Listing 4.1: An example of a Method Declaration in Java

Version	Method Name	Type	Parameters	Method Body
parsed	channelRead	void	ChannelHandlerContext ctx, Object msg	if (acceptMessage(msg)) { log((SpdyFrame) msg, Direction.INBOUND); } ctx.fireChannelRead(msg);
token-level	channelRead	void	channelhandlercontext ctx object msg	if acceptmessage msg log spdyframe msg direction inbound ctx firechannel-read msg
sub-token-level	channel read	void	channel handler context ctx object msg	if accept message msg log spdy frame msg direction inbound ctx fire channel read msg

Table 4.1: Example of a parsed Java Method

4.3 Data Preprocessing

In my thesis, I consider 2 types of models. In a first approach, the model operates on a token-level. This means that the method name and all identifiers used in source code are considered to be tokens. In a second approach, the model operates on a sub-token level which implies that tokens in source code can be further split into sub-tokens.

List of Operations The following operations have been performed for all models in order to clean the data:

1. String values have been replaced with a separate token: For example: errormessage = "there is an error message" is transformed to errormessage = STRINGVALUE
2. All characters are transformed to lowercase

3. Special characters such as newline and {}()',.;" have been removed
4. Abstract methods are removed from the dataset. For example:

```java
public abstract int myMethod(int n1, int n2);
```

5. Empty methods are removed from the dataset. For example:

```java
public void voidCastleRight() { }
```

Additionally, for the sub-token models, the tokens are split according to CamelCase or snake case. Table 4.1 provides a sample of a preprocessed method on a token-level and a sample of a preprocessed method according to the sub-token level respectively. For simplicity, the same example has been used as in the parsing section (listing: 4.1).
Additionally, I have recognized a big number of project-specific method names in the Allamanis dataset. Specifically, I have observed 1000 different "getXXX" method names. An example of such a method name is the following:

```java
public int get10() {
      return 10;
   }
```

Listing 4.2: An example of a project-specific or bad Method Name

These methods seem useless to the model and have therefore been removed from the dataset. Table 4.2 shows the amount of abstract, empty and getXXX methods. Finally, the total amount of training samples is shown. The amount of abstract methods are 8-10% independent of the dataset and 1-3% of the methods are empty. More interesting is that for the Allamanis dataset nearly 1/3 of all the methods have been composed of getXXX methods. Including these methods during training significantly impacted the training and evaluation of the model. The numbers of final training methods in table 4.2 approximately align with the number of methods Alon et al. [25] count in their Java-small dataset.

Type of Method	Allamanis Dataset	Java-small
All Methods	186694	750485
Abstract Methods	15573	90373
Empty Methods	2409	27554
getXXX Methods	51121	2
Final Training Methods	117591	632556

Table 4.2: Preprocessing of Methods

4.3.1 Rare Tokens

Rare tokens are words that occur less than a certain amount of times in the training set. This is due to the fact that the model is not able to learn an appropriate embedding for the word if it occurs rarely. In this work, tokens that appear less than 3 times in the training set get replaced by an UNK token. The percentage of unknown method names can be seen in table 4.3. The Allamanis training dataset is composed of 44% unknown method names which is significantly higher than the Java-small dataset. This is probably due to fewer samples in the dataset. The high percentage of unknown method names, in general, provides a strong motivation to split the method names into sub-tokens.

Dataset	Allamanis Dataset	Java-small
Total UNK	51372	170050
Total Method Names	117591	632556
Percentage UNK	44%	27%

Table 4.3: Amount of Unknown Method Names during Training

4.4 Statistics on the Data

	Token Model		Sub-token Model	
Dataset	50%	75%	50%	75%
Allamanis Dataset	11	35	17	48
Java-small	8	30	13	45

Table 4.4: Length of method bodies

	Token Model		Sub-token Model	
Dataset	50%	75%	50%	75%
Allamanis Dataset	2	2	2	4
java-small	2	4	2	6

Table 4.5: Length of method parameters

	Sub-Token Model	
Dataset	50%	75%
Allamanis Dataset	2	3
Java-small	3	4

Table 4.6: Length of method names

Table 4.4 and Table 4.5 illustrate the length of method bodies and the length of parameters respectively. The input of the models is composed of the return type, the body and the parameters of a method. In deep learning models, the input has to be of a fixed size: shorter inputs get

padded with a special PAD token and longer inputs get truncated. I use the distribution of the length of method bodies and parameters to determine an optimal input size. I use values between the 50% and the 75% quantile.
Similarly for the sub-token model, I need to decided on the maximum length of the target sequence, i.e. the maximum length of sub-tokens a method name can be composed of. Likewise, I use the distribution of the length of method names to determine a reasonable target size. Table 4.6 suggests to use values between 2 – 4.

Chapter 5

Approach

5.1 Models

In this chapter, I propose 2 different approaches to suggest method names given its type, parameters, and body. For each approach, I describe different models that are being used during training and evaluation respectively. Furthermore, for each model, the architecture and hyperparameters are provided.

5.1.1 Token Based Models

In this section, I will introduce the models that operate on a token-level. This approach assumes that a method name and all identifiers used in source code are tokens. Therefore, on a token level, the goal is to predict one token (the method name) based on several input tokens (the method body, parameters, and return type). In deep learning models, the size of the input has to be determined beforehand and is a fixed size. For the method body, I include the first 22 tokens. For the parameters, I include the first 4 tokens. I found that including more tokens did not improve the results of the models. These values also align with the distribution of the length of the method bodies and parameters, analyzed in the data section 4.4. Furthermore, I also include the return type of the function. Therefore the size of the input is set to 27. Models with longer method bodies or parameters are being truncated and methods with shorter parameters and method bodies are padded with a special PAD token.
For the example introduced in the Data chapter, the input for a token based model can be seen in table 5.1.

Feed-forward Token Model

Figure 5.1 illustrates the feed-forward Token Model architecture. It consists of an input layer, an embedding, a flatten, a dropout layer and 2 dense layers.
Model Hyperparameters: The hyperparameters of the model have been optimized with hyperas. Hyperas is a convenient tool to test different hyperparameter ranges for different layers [15]. The size of the word embeddings is set to 128. After, dropout at a rate of 0.488 is added. Dropout randomly sets some inputs to 0, which prevents overfitting [50]. Then, word embeddings are flattened to turn the 3-dimensional tensor into a 2-dimensional tensor. A tensor is generalized matrix which can have 0 dimensions (a scalar), 1 dimension (a vector), 2 dimensions (a matrix) or more than 2 dimensions [18]. Flattening the tensor does not affect the batch size. Furthermore, dropout at a rate of 0.085 is added. Dropout is a form of regularization and addresses overfitting neural networks on the training data [51]. The main idea is to randomly "drop" (around $5-50\%$

Version	Method Name	Type	Parameters	Method Body
token-level	channelRead	void	channelhandlercontext ctx object msg	if acceptmessage msg log spdyframe msg direction inbound ctx firechannel-read msg
	Output	Input		
token-level input	channelRead	if acceptmessage msg log spdyframe msg direction in-bound ctx firechannelread msg void channelhandlercon-text ctx object msg PAD PAD PAD PAD PAD PAD		

Table 5.1: Example of an Input for a token based Model

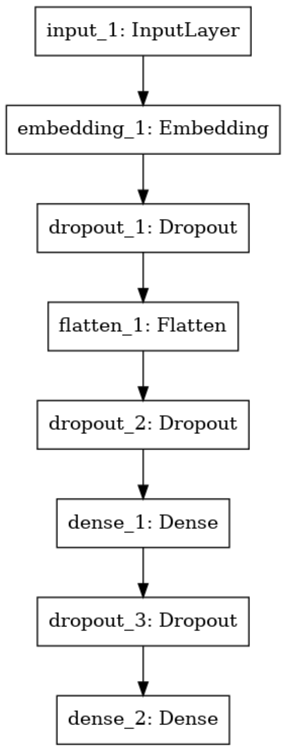


Figure 5.1: Feed-forward Token Model

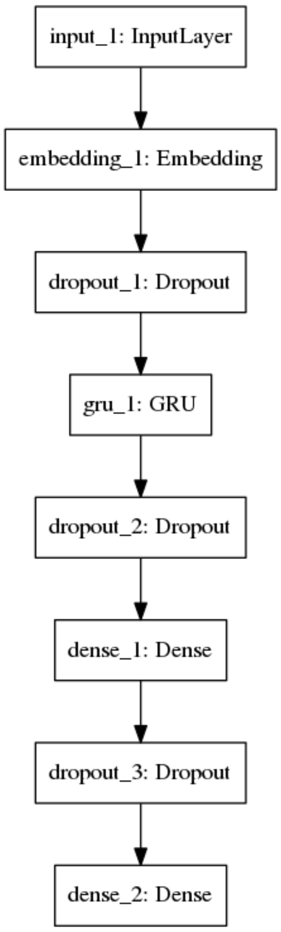


Figure 5.2: GRU Token Model

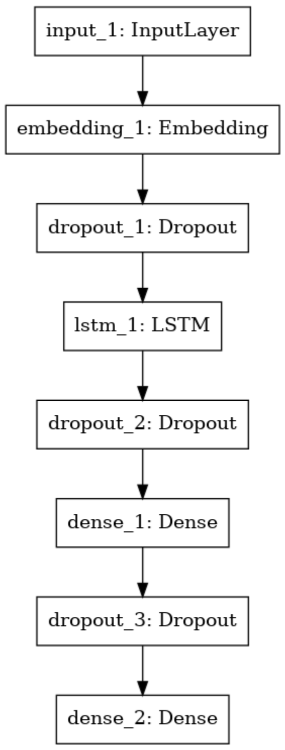


Figure 5.3: LSTM Token Model

of) neurons.
The size of the dense layer is set to 70 and the Exponential Linear Unit (ELU) non-linearity function has been applied. The ELU is an activation function that speeds up neural networks and leads to higher classification accuracies [36]. It is a generalization of the rectified linear unit (ReLU) which also avoids the vanishing gradient problem. Unlike the ReLU, ELU uses negative values to push the activations closer to 0.
After, dropout at a rate of 0.383 is added. The second dense layer computes a softmax non-linearity function which produces a probability distribution over the vocabulary. The model has been trained with a batch size of 128 over 15 epochs. The batch size determines the number of inputs (i.e. rows in the data) the model computes before it updates the model's parameters through backpropagation [17]. The epoch determines the number of times the model goes through the entire data set [17].
Before training, random unknown method names have been removed until a rate of about 30% unknown tokens. I did this to ensure the model is learning something beyond always or nearly always predicting the unknown token.

GRU Token Model

The architecture for the GRU token model is similar to the feed-forward token model but instead of showing all the input embeddings to the model at once, the word embeddings are processed by iterating through the word embeddings and maintaining a state containing information of what the model has seen so far [34]. The GRU token architecture can be seen in Figure 5.2.
Model Hyperparameters: The hyperparameters of the model have been optimized with hyperas [15]. The size of the word embeddings is set to 128. Then, dropout at a rate of 0.235 is added. After, word embeddings are processed with a GRU. The size of the GRU is set to 200 and recurrent dropout of 0.325 is applied. Recurrent dropout randomly drops the connections between the recurrent units [37].
Furthermore, dropout at a rate of 0.36 is added. The size of the dense layer is set to 70 and the elu non-linearity function has been applied. Furthermore, dropout at a rate of 0.25 is appended. The last dense layer computes a softmax non-linearity function which produces a probability distribution over the vocabulary. The model has been trained with a batch size of 64 over 17 epochs. Before training, random unknown method names have been removed until a rate of about 30% unknown tokens.

LSTM Token Model

The architecture for the LSTM token model is similar to the GRU token model. Figure 5.3 exemplifies the LSTM architecture.
Model Hyperparameters: The hyperparameters of the model have been optimized with hyperas [15]. The size of the word embeddings is set to 128. Then, dropout at a rate of 0.24 is added. After, word embeddings are processed with an LSTM. The size of the LSTM is set to 200 and recurrent dropout of 0.32 is applied. Furthermore, dropout at a rate of 0.36 is added. The size of the dense layer is set to 200 and the elu non-linearity function has been applied. Furthermore, dropout at a rate of 0.28 is appended. The last dense layer computes a softmax non-linearity function which produces a probability distribution over the vocabulary. The model has been trained with a batch size of 64 over 17 epochs. Before training, random unknown method names have been removed until a rate of about 30% unknown tokens.

Bidirectional LSTM Token Model

Unidirectional LSTMs compute a memory cell which is a summary of the past sequences. Bidirectional LSTMs are trained in forward and backward directions [48]. They have initially been very successful in speech recognition tasks [38]. The same study showed that bidirectional LSTMs are sometimes significantly more effective than unidirectional ones.
In this work, I use the concatenation mode to combine the forward and backward output of the 2 LSTMs before they are passed to the next layer. This model is, therefore, an extension of the LSTM Token Model.
Model Hyperparameters: The hyperparameters of the model have been optimized with hyperas [15]. The size of the word embeddings is set to 128. Then, dropout at a rate of 0.45 is added. After, word embeddings are processed with a bidirectional LSTM. The size of the forward and backward LSTMs are both set to 100, similar to the size of the word embeddings and recurrent dropout of 0.1843 is applied. Furthermore, dropout at a rate of 0.2 is added. The size of the dense layer is set to 70 and the elu non-linearity function has been applied. Furthermore, dropout at a rate of 0.2 is appended. The last dense layer computes a softmax non-linearity function which produces a probability distribution over the vocabulary. The model has been trained with a batch size of 128 over 30 epochs. Before training, random unknown method names have been removed until a rate of about 30% unknown tokens.

5.1.2 Sub-token Based Models

Token based models assume that all identifiers used in source code are composed of one token. In past work Allamanis et. al [22] proposed to split tokens further into sub-tokens according to camelCase or snake case. Therefore, on a sub-token level, the goal is to predict a sequence of sub-tokens that compose a method name based on input tokens. The input tokens are also split according to camel case and snake case. Equivalent to the token based models, the input size has to be determined beforehand. For the method body, I include the first 37 sub-tokens. For the parameters, I include the first 6 sub-tokens. I found that including more sub-tokens did not improve the results of the models. These values also align with the distribution of the length of sub-tokens in method bodies and parameters respectively, analyzed in the data section 4.4. Furthermore, I also include the return type of the function. Additionally, the size of the target sequence has to be determined beforehand. For the method name, I decided to include the first 4 sub-tokens. Finally, special start- and end-tokens are added to the input and output sequence. Therefore the size of the input is set to 46 and the size of the output is set to 6. Models with longer method bodies or parameters are being truncated and methods with shorter parameters and method bodies are padded with a special PAD token.
Table 5.2 exemplifies the input for a sub-token based model with the same sample introduced in the dataset chapter.

Sequence-to-Sequence Sub-token Model

The Sequence-to-Sequence model consists of an encoder and a decoder. Both are composed of 2 separate LSTMs. Figure 5.4 clarifies the architecture of the proposed model.
Model Hyperparameters: The hyperparameters of the model have been optimized with talos [4]. Talos is another tool for hyperparameter optimization. It has the advantage over hyperas to be more flexible but has less functionality. The size of the word embeddings is set to 256. After, word embeddings are processed with an LSTM. The size of both the encoder and decoder LSTMs is set to 250, similar to the size of the word embeddings. For both the encoder and the decoder recurrent dropout of 0.2 and dropout of 0.2 is added.

Version	Method Name	Type	Parameters	Method Body
sub-token-level	channel read	void	channel handler context ctx object msg	if accept message msg log spdy frame msg direction inbound ctx fire channel read msg
sub-token-level input	starttoken channel read endtoken PAD PAD	starttoken if accept message msg log spdy frame msg direction inbound ctx fire channel read msg void channel handler context ctx object msg endtoken PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD		

Table 5.2: Example of an input for a sub-token based model

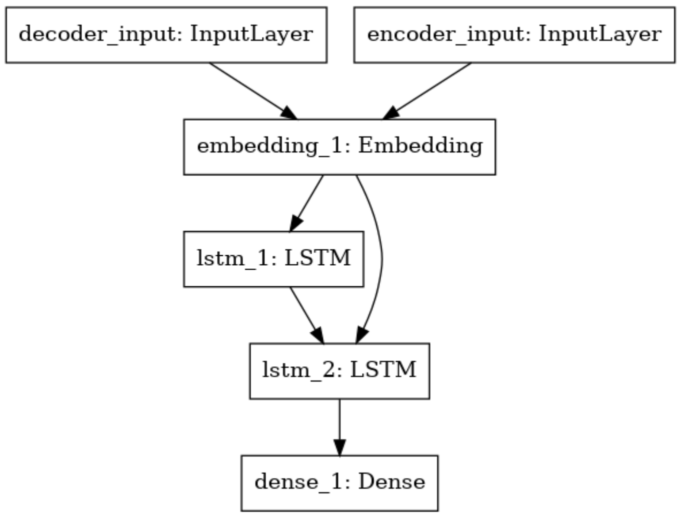


Figure 5.4: Sequence-to-Sequence Sub-token Model

Sequence-to-Sequence with Attention Sub-token Model

Similar to the Sequence-to-Sequence model, this model also consists of an encoder and a decoder. These are composed of 2 separate GRUs. GRU's have been chosen because they are composed of fewer parameters than the LSTMs. This has a significant impact on the Seq2Seq Attention model because it tends to be much slower than the regular Seq2Seq model during training.
Model Hyperparameters: Since the model's training is very time consuming, the hyperparameters were not optimized. Instead, the same values from the Sequence-to-Sequence Token Model have been used. The size of the word embeddings is set to 256. After, word embeddings are processed with a GRU. The size of both the encoder and decoder GRUs is set to 200, similar to the size of the word embeddings. For both the encoder and the decoder recurrent dropout of 0.2 and dropout of 0.2 is added.

Chapter 6

Evaluation

In this chapter, I evaluate the different approaches. Furthermore, I want to know if the token approach or the sub-token approach is better to predict method names. Finally, I want to comprehend what the models' input need to be composed of in order to predict meaningful method names.
Therefore, I specifically focus on the following research questions:

- RQ1: How well can we predict identifier names using neural networks?
- RQ2: Are token or sub-token models better to predict identifier names?
- RQ3: What kind of context information is relevant for suggesting identifier names?

6.1 Predicting Identifier Names using Neural Networks

To answer RQ1, the token based models and the sub-token based models are evaluated and compared using different metrics. First the top-1 and top-5 accuracies are computed to understand if the predictions of the models would be potentially useful in a realistic setting. Furthermore the F1-Score is computed, this represents the harmonic average of precision and recall, where F1=1 is the best score and F1=0 is the worst score. Its value is most influenced by the lowest of precision and recall [43]. Precision is the number of true positives divided by the number of relevant results according to the model and recall is the number of true positives divided by the number of relevant results (all samples that should have been identified) [3]. The formula of the F1-Score is computed as follows:

$$F_1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6.1}$$

For the sub-token models, the accuracy and an F1-Score on a sub-token level are computed.

6.1.1 Evaluation of Token based Models

This section contains the evaluation of the token based models, which includes the Feed-forward Token Model, the GRU Token Model, the LSTM Token Model, and the bidirectional LSTM Token Model. Table 6.3 shows the accuracy of the different token models. Since the percentage of unknown method names is relatively high, it is is also included in the table. Surprisingly, the feed-forward Token Model scores the best results for both the Allamanis and the Java-small dataset. For the Allamanis dataset, the model achieves an overall accuracy of 62.89% and an additional 2% over the GRU and the LSTM Token Model and an additional 4% over the bidirectional LSTM Token Model. For the Java-small dataset, the overall accuracy is a bit lower: 60.78% for the feed-forward Token Model which similarly scores 2 – 5% better than the other token models. Interestingly, the naive baseline to always predict UNK is more accurate overall than the individual token models for the Java-small dataset.

Token Model	Allamanis Dataset	Java-small
Always predict UNK	51.53%	61.14%
Feed-forward Token Model	62.89%	60.78%
GRU Token Model	60.69%	57.16%
LSTM Token Model	60.07%	55.7%
Bidirectional LSTM Token Model	58.57%	57.4%

Table 6.1: Accuracy of Token Models

Table 6.2 illustrates the top-5-accuracy of token models which has an improvement of 18 – 25% over the top-1-accuracy of the different token models for the Allamanis dataset. Similarly, the token models score a 10 – 15% higher top-5-accuracy for the Java-small dataset.

The numbers reported so far contain a bias, because the UNK token is treated like any other

Token Model	Allamanis Dataset	Java-small
Feed-forward Token Model	74.92%	71.18%
GRU Token Model	74.70%	71.3%
LSTM Token Model	73.27%	71.31%
Bidirectional LSTM Token Model	72.08%	71.03%

Table 6.2: Top-5-Accuracy of Token Models

method name and if the model correctly predicts UNK, accuracy improves. I therefore adapt an F1-Score that has been used by Alon et. al [25] for a similar task before. The F1-Score is composed of the amount of true positives, false positives and false negatives. I adjust it, such that the UNK token is treated specially, neither completely right nor completely wrong [22]. For the F1-Score and the Allamanis dataset, all the models achieve a similar score of 0.44 – 0.47. I report much lower results for the Java-small dataset: The models all achieve a score between 0.19 – 0.22.

So far one could interpret these results as follows: For the inputs I pass to the model, an RNN architecture (LSTM, GRU, bidirectional LSTM) does not have any advantage over a feed-forward neural network architecture.

However, figure 6.1 draws a different image. It illustrates the connection between the suggestion frequency and the accuracy of the different token models. For example, the feed-forward token model scores an accuracy of 93.85% if only the top 0.1 predictions of the feed-forward token model are being shown to the developer but scores an accuracy of only 37.49% if every prediction would be shown to the end-user. This plot excludes all UNK method names in order to illustrate that the models can also predict something useful beyond the UNK token.

Token Model	Allamanis Dataset	Java-small
Feed-forward Token Model	0.44	0.19
GRU Token Model	0.47	0.22
LSTM Token Model	0.46	0.22
Bidirectional LSTM Token Model	0.45	0.22

Table 6.3: F1 Score of Token Models

Interestingly, the RNN models achieve a higher accuracy if unknown method names are removed from the dataset: If every prediction is shown to the developer, the RNN models achieve an accuracy of 46 – 48%. The plot furthermore illustrates that in a realistic setting, it makes sense to include a threshold to only show predictions above a certain probability to a potential user.

Figure 6.2 reveals a similar picture. Like for the Allamanis dataset, accuracy improves as pre-

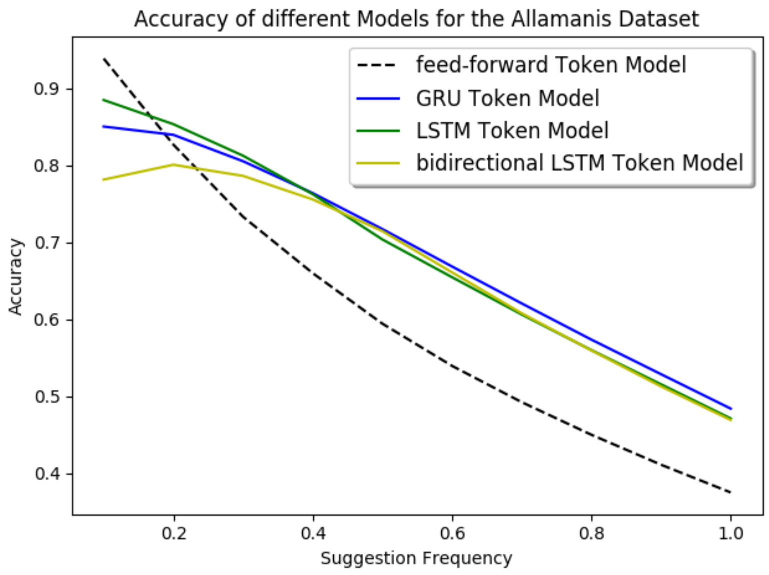


Figure 6.1: Accuracy vs. Suggestion Frequency for the Allamanis dataset

diction confidence increases. The RNN token models score a better result than the Feed-forward Token model independent of the chosen prediction confidence.

Unknown Method Names

We have seen that unknown method names pose a general difficulty when evaluating the token models. It's furthermore interesting to compare the number of unknown method names of the Allamanis test set and the Java-small test set. Because although the Java-small dataset consists of a bigger dictionary of method names, the test set contains a higher percentage of unknown method names. To my understanding, this goes back to how the data is split into the training and test set. For the Allamanis dataset, method names that appear in the test set are from the same projects that also appear in the training set. Differently, for the Java-small dataset, method names

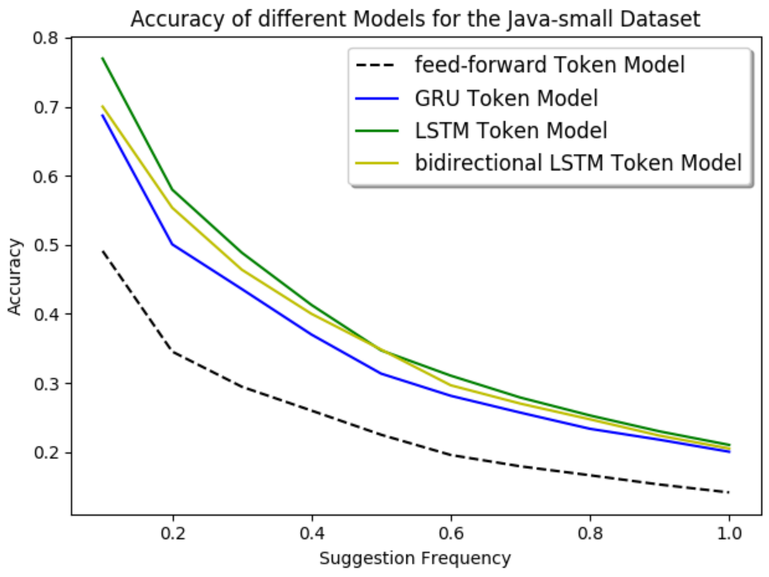


Figure 6.2: Accuracy vs. Suggestion Frequency for the Java-small dataset

that appear in the test set are from projects the model has not seen during training.
Consider the following example: A project has an abstract method that is implemented 4 times. According to the way the Allamanis dataset is split it is likely that 1 out of the 4 implemented methods are contained in the test set, while the other 3 implemented methods are in the training set. This implemented method would then be known by the model, independent of its name. Therefore, even project-specific method names are known by the models for the Allamanis dataset. This is in contrast to the Java-small test set. The model only knows method names that are not specific to any project. This small detail results in the significant higher F1-Score for token models trained and evaluated on the Allamanis dataset over token models trained and evaluated on the Java-small dataset.

6.1.2 Evaluation of Sub-token Based Models

In this section, I discuss the results for the sub-token based models which includes the Sequence-to-Sequence Sub-token Model and the Sequence-to-Sequence with Attention Sub-token Model. First, accuracy is computed. Although, a more interesting evaluation of the sub-token models is the F1-score on a sub-token level adapted from Alon et. al [25]. This metric is able to also credit the model if it partially predicted a method name correct.
For example, accuracy rewards the model if it correctly predicts "get UNK", but punishes the model for "get UNK cache" if the correct prediction is "get UNK cache directory". The model should be given more credits to the second example for being able to predict several sub-tokens of the method name. Table 6.4 illustrates the different results for the sub-token based models for the Allamanis dataset. The first column shows the accuracy of the different sub-token models. The second column shows the accuracy of the different sub-token models where k equals the size of beam search. I chose k to be rather large to find potential differences between greedy decoding ($k = 1$) and beam search decoding ($k > 1$). Column 3 illustrates the F1-Score for the Sequence-to-Sequence Sub-token Model (beam size $k = 1$ and $k = 100$) and for the Sequence-to-Sequence with Attention Sub-token Model. Table 6.5 illustrates the different results for the sub-token based

Sub-token Model	Accuracy	F1 Score
Sequence-to-Sequence Sub-token Model (k=1)	22.1%	0.44
Sequence-to-Sequence Sub-token Model (k=100)	23.25%	0.43
Sequence-to-Sequence with Attention Sub-token Model	28.85%	0.47

Table 6.4: Metrics of Sub-token Models for the Allamanis Dataset

models for the Java-small dataset. For both datasets, beam search did neither improve nor reduce results. Similar to the evaluation of the token based models, the results I report for the Java-small dataset are much lower than the ones from the Allamanis dataset: Accuracy for the Sequence-to-Sequence Sub-token Model is between 12 – 14% for the Java-small dataset and between 22 – 29% for the Allamanis dataset. Also, the F1-Score for the Java-small dataset is lower (0.32 – 0.36) than for the Allamanis dataset (0.43 – 0.47). The Sequence-to-Sequence Sub-token Model with Attention scores the highest results for the Allamanis dataset: The overall accuracy is 28.85% and the F1-Score is 0.47.

Sub-token Model	Accuracy	F1 Score
Sequence-to-Sequence Sub-token Model (k=1)	13.26%	0.34
Sequence-to-Sequence Sub-token Model (k=100)	12.33%	0.36
Sequence-to-Sequence with Attention Sub-token Model	12.6%	0.33

Table 6.5: Metrics of Sub-token Models for the Java-small Dataset

To answer RQ1, I conclude that both token models and sub-token models are able to accurately suggest method names. If unknown method names are excluded from the test set, the accuracy of RNN Token Models provides a relative increase of 21% over the feed-forward Token Model. Furthermore, the token models' accuracy improves as their predictions' confidence increases. For the sub-token approach, the attention mechanism provide a relative increase of 27% in accuracy for the Allamanis dataset and resulted in a slightly higher F1-Score. I did not observe any increase in accuracy or the F1-Score for the Java-small dataset though. Beam search did not significantly increase or reduce my result for both datasets.
Finally, the token and sub-token models are significantly more effective if partially trained on the source code project they later predict on.

6.2 Token Models vs. Sub-token Models

In this section, I will compare the F1-Score of token models with the F1-Score of sub-token models in order to answer RQ2. This metric best evaluates the performance of the individual models because it credits the sub-token models for partial correctness and doesn't punish the prediction of the UNK token. The F1-Score has a slight bias towards the token models because more difficult and rare method names are replaced by the UNK token and are therefore ignored. For the Allamanis dataset, Both the best token model (GRU Token Model) and the best sub-token model (Sequence-to-Sequence with Attention Sub-token Model) score the same F1-Score of 0.47. For the Java-small dataset, the best sub-token model's F1-Score (Sequence-to-Sequence Sub-token Model with beam size $k = 100$) is significantly higher than the F1-Score obtained by the best token model

Token Model	Allamanis Dataset	Java-small
Best Token Model	0.47	0.22
Best Sub-token Model	0.47	0.36

Table 6.6: F1-Score of the best Token and Sub-token Model

(GRU Token Model).

Based on this metric, we can conclude that the token models' performance is similar to the performance of sub-token models if partially trained on the source code project it later predicts. If the model's goal is to predict on a previously unobserved source code project, the sub-token models provide a better suggestion of method names. Nevertheless, these values can only be compared with caution, because the F1-Score is slightly different constructed for the token and the sub-token approach.

6.2.1 Qualitative Analysis

In this section, I will show different use-cases and illustrate what the models are able to predict and what the models cannot predict. The goal is to better understand the strengths and weaknesses of predictions made by the token and sub-token based models to answer RQ2.
First, I will illustrate what the models are able to predict with 4 different examples. For each case, the first row provides a code snippet: a method in Java. For this particular method, the goal is to find the function name denoted as "f" (where the function name is usually placed). Row 2 presents the ground truth. Each row (3 – 8) then shows the prediction for this specific code snippet made by the individual models. Row 9 visualizes the attention weights by the Sequence-to-Sequence with Attention Sub-token Model.
Table 6.7 illustrates the prediction of the different models for a short method body and method name. The token models (feed-forward, GRU, LSTM, bidirectional LSTM Token Model) and the sub-token models (Sequence-to-Sequence, Sequence-to-Sequence with Attention Sub-token Model) all correctly predict "clone" as the method name. In the last row, a visualization of the attention weights for the Sequence-to-Sequence with Attention Sub-token Model can be seen. It illustrates the importance of the individual input sub-tokens for each generated output sub-token: E.g. to predict the sub-token "clone", the model emphasizes most on the input sub-token "clone". A slightly more complicated case exemplifies Table 6.8: In this example, the method name is not already present in the method body. All models except the feed-forward model are able to predict "getItem" correctly. The attention visualization shows that to predict the sub-token "item", the model focuses mostly on the input sub-token "position". Table 6.9 demonstrates a more difficult problem. The function name "getNodesStats" is not known by the token models. Hence, their prediction result in the UNK token. The Sequence-to-Sequence Sub-token model predicts the sub-token "stats", which is only partially correct. Only the Sequence-to-Sequence with Attention Sub-token Model is able to predict the complete method name correctly. Again, the last row illustrates the attention distribution. The last example I present for which the models make a true prediction is the most complicated one. It consists of a long method body and the method name is composed of 4 sub-tokens (test, single, valued, field). Interesting here are the attention weights: For the predicted sub-token "single", the model most focuses on the input sub-tokens around "query" and "match". Next, I present 3 cases for which the method name cannot be predicted with token-based models. The method name is unknown to the model and hence out-of-

	Case 1

Code Snippet	
	```
	public ExtendedRails f(){
	  return (ExtendedRails) super.clone();
	  }
	```

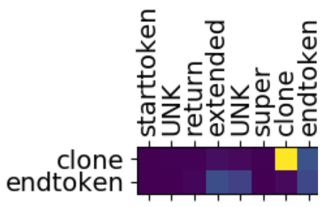
Ground truth	clone
feed-forward Token Model	clone
GRU Token Model	clone
LSTM Token Model	clone
bidirectional LSTM Token Model	clone
Sequence-to-Sequence Sub-token Model	clone
Sequence-to-Sequence with Attention Sub-token Model	clone
Attention Weights	

Table 6.7: Case 1: A short method name that consists of 1 sub-token

	Case 2

Code Snippet	`public Object f(int position){`
	`  return position;`
	`  }`

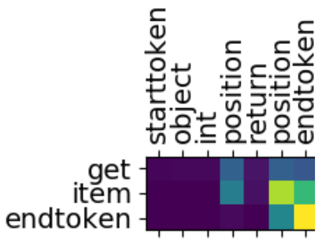
Ground truth	getItem
feed-forward Token Model	getItemId
GRU Token Model	getItem
LSTM Token Model	getItem
bidirectional LSTM Token Model	getItem
Sequence-to-Sequence Sub-token Model	getItem
Sequence-to-Sequence with Attention Sub-token Model	getItem
Attention Weights	

Table 6.8: Case 2: A slightly more complicated case: The method name consists of 2 sub-tokens

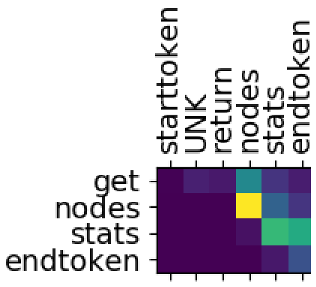
	Case 3
Code Snippet	`public ClusterStatsNodes f(){` `return nodesStats;` `}`
Ground truth	getNodesStats
feed-forward Token Model	UNK
GRU Token Model	UNK
LSTM Token Model	UNK
bidirectional LSTM Token Model	UNK
Sequence-to-Sequence Sub-token Model	stats
Sequence-to-Sequence with Attention Sub-token Model	getNodesStats
Attention Weights	

Table 6.9: Case 3: The ground truth is not in the vocabulary

	Case 4

Code Snippet

```java
@Test
  public void f()
   throws Exception {
     SearchResponse searchResponse = client()
         .prepareSearch("idx")
         .setQuery(matchAllQuery())
             .addAggregation
               (extendedStats("stats")
             .field("value"))
             .execute().actionGet();
     assertThat(searchResponse.getHits()
         .getTotalHits(), equalTo(10l));

     ...
   }
```

Ground truth	testSingleValuedField
feed-forward Token Model	testUnmapped
GRU Token Model	testSingleValuedField
LSTM Token Model	testSingleValuedFieldPartiallyUnmapped
bidirectional LSTM Token Model	testMultiValuedField
Sequence-to-Sequence Sub-token Model	testSingleValuedField
Sequence-to-Sequence with Attention Sub-token Model	testSingleValuedField

Attention Weights

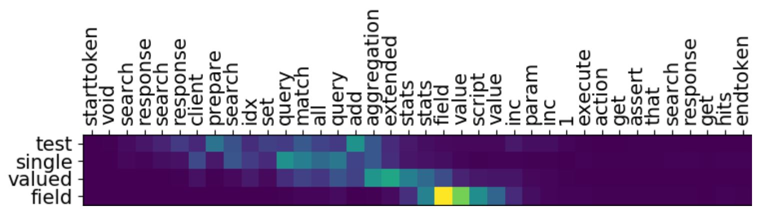


Table 6.10: Case 4: A complicated use-case where the method name is composed of 4 sub-tokens

vocabulary. Table 6.11 demonstrates this: The goal is to predict the method name "decodeInteger" which it has never seen before. All the token models predicted UNK and are therefore not helpful for a potential end-user. Instead, the Sequence-to-Sequence Sub-token Model with beam size $k = 100$ predicts the sub-token decode, which is at least partially correct. Another interesting

	Case 5

Code Snippet	`public BigInteger f(byte[] pArray){`
	`   return new BigInteger(1,`
	`    decodeBase64(pArray));`
	`}`

Ground truth	decodeInteger
Token Models	UNK
Sequence-to-Sequence Sub-token Model ($k = 100$)	decode

Table 6.11: Case 5: Partially correct prediction by the Sequence-to-Sequence Sub-token Model with $k = 100$

prediction provided by the Sequence-to-Sequence Sub-token Model is demonstrated in table 6.12. For another method name the models have never seen before (and hence the prediction from the Token Models are UNK) the Sequence-to-Sequence Sub-token Model makes a prediction which captures the semantical meaning of the function. For beam size $k = 1$ (greedy decoding), the model predicts "getPagesCount" and for beam size $k = 100$, the model predicts "count" while the ground truth is "countPages". An example for a longer function body (and hence a more difficult task for the models) is provided in table 6.13. The ground truth in this example is "ejectUserFromMeeting". This method name is not in the model's vocabulary and the token models therefore correctly predict the UNK token. The Sequence-to-Sequence Sub-token Model predicts "removeUser". "Remove" and "Eject" have a similar meaning and therefore this prediction is more useful than just UNK provided by the token models. The last example I will show is a situation where my models are not useful to a potential end-user. Case 8 in table 6.14 is a complicated function that is originally close to 90 lines long. It is difficult for the models to infer the method's functionality. The token models all predict UNK. The sub-token models also don't predict anything useful. But in this situation, the token models' prediction is better: at least nothing is shown to the end-user.

Case 6

Code Snippet

```java
private int f
  (UploadedPresentation pres) {
    int numPages = 0;
    if (pageCounter == null) {
      log.warn("No page counter!");
      return 0;
    }
    numPages = pageCounter
      .countNumberOfPages
      (pres.getUploadedFile());
    log.debug("There are " + numPages);
    return numPages;
  }
```

	Case 6
Ground truth	countPages
Token Models	UNK
Sequence-to-Sequence Sub-token Model ($k = 1$)	getPagesCount
Sequence-to-Sequence Sub-token Model ($k = 100$)	count

Table 6.12: Case 6: Partially correct Prediction by the Sequence-to-Sequence Sub-token Model

	Case 7
Code Snippet	`public void f (Map<String, String> msg) { String userId = (String) msg.get("userId"); String ejectedBy = (String) msg.get("ejectedBy"); IScope scope = Red5 .getConnectionLocal() .getScope(); application.ejectUserFromMeeting (scope.getName(), userId, ejectedBy); }`
Ground truth	ejectUserFromMeeting
Token Models	UNK
Sequence-to-Sequence Sub-token Model (k = 1)	removeUser
Sequence-to-Sequence Sub-token Model (k = 100)	removeUser

Table 6.13: Case 7: the Sequence-to-Sequence Sub-token Model is able to capture the function's semantics

	Case 8

Code Snippet

```
public int f
  (File presentationFile) {
    int numPages = 0;
    String COMMAND = SWFTOOLS_DIR +
      "/pdf2swf -I "
      + presentationFile
        .getAbsolutePath();

     Timer timer = null;
     Process p = null;
     try {
        timer = new Timer(true);
        InterruptTimerTask interrupter =
          new InterruptTimerTask
            (Thread.currentThread());
        timer.schedule
          (interrupter, 60000);

        p = Runtime.getRuntime()
          .exec(COMMAND);
        BufferedReader stdInput =
          new BufferedReader
          (new InputStreamReader
            (p.getInputStream()));
      BufferedReader stdError =
        new BufferedReader
        (new InputStreamReader
          (p.getErrorStream()));
      String info;
      Matcher matcher;

  ...

  }
```

Ground truth	countNumberPages
Token Models	UNK
Sequence-to-Sequence Sub-token Model ($k = 1$)	getDefaultNumCancel
Sequence-to-Sequence Sub-token Model ($k = 100$)	encode

Table 6.14: Case 8: Useless Predictions (by all the models)

6.3 Different Input Parameters

To answer RQ3, I run the GRU Token Model and the Sequence-to-Sequence Sub-token Model with different input parameters and evaluate them on the Allamanis dataset. I chose the GRU Token Model because it scored the best F1-Score for the original input (RQ1). Table 6.15 illustrates how the F1-Score changes when I adjust the input for the GRU Token Model. Row 1 represents the optimal input (and was used to evaluate RQ1). If I only include the first 4 parameters and no tokens from the method body, the F1-Score decreases 20.51%. Furthermore, if only the first 6 or 18 tokens of the method body are included, the F1-Score decreases 9.3% or 4.44% respectively.

	F1-Score of the GRU Token Model	Relative Increase
Method Body = 22 Parameters = 4 including Type	0.47	0%
Method Body = 0 Parameters = 4 including Type	0.39	−20.51%
Method Body = 6 Parameters = 0 including Type	0.43	−9.3%
Method Body = 18 Parameters = 0 including Type	0.45	−4.44%

Table 6.15: Token F1 Score for different Inputs

Table 6.16 shows a similar picture: The F1-Score decreases substantially if the model had to predict method names solely based on the first 4 sub-tokens of the parameters. Additionally, the F1 Score decreases 15.79% if I only include the first 6 sub-tokens from the method body and 4.76% if only the first 18 sub-tokens are included.

I therefore conclude that an optimal model needs to know about the function's parameters and the method body.

	F1-Score of the Sequence-to-Sequence Sub-token Model	Relative Increase
Method Body = 28 Parameters = 4 including Type	0.44	0%
Method Body = 0 Parameters = 4 including Type	0.31	−41.94%
Method Body = 6 Parameters = 0 including Type	0.38	−15.79%
Method Body = 18 Parameters = 0 including Type	0.42	−4.76%

Table 6.16: Sub-token F1 Score for different Inputs

Chapter 7

Threat to Validity

In this chapter, I highlight some of the difficulties that arise when I applied deep learning models to source code.

7.1 Out-of-Vocabulary Tokens

OOV (Out of Vocabulary) words are tokens that do not occur during training and are a general difficulty in NLP. When applying deep learning models to source code the problem is even more common: Identifier names, such as class, method or variable names often consist of neologisms, i.e. a newly invented word or a new meaning for an existing word. Consider the following function. It is clear to the reader that this method returns a reversed ArrayList. The model might have seen the function name "getReversedList" but not with the acronym "Rev" instead of "Reversed". Hence, during testing, the token models would replace "getReversedList" with the UNK identifier. To mitigate this threat, I split method names into sub-tokens.

```java
public ArrayList getRevList(ArrayList orig)
   {
      ArrayList reversed = new ArrayList();
      for(int i= orig.size()-1; i>=0; i--)
      {
         Object obj = orig.get(i);
         reversed.add(obj);
      }

      return reversed;
   }
```

Listing 7.1: An example of a Neologism in a Java Method Name

Furthermore, a good model might suggest "get, reversed, list" which is an accurate suggestion. But during evaluation, "rev" and "reversed" is considered to be wrong. To mitigate this threat, I included a qualitative analysis of the models.

7.2 Quality of Training Data

During training, another threat is that the model learns project-specific method naming conventions or even worse the training set used during training consists of bad naming of method names. Such an example is provided below. In the Allamanis dataset, a project (Platform Frameworks Base) consists of 1000 "getXXX" method names. These methods are not useful for the model. Hence, I removed these methods before training.

7.3 Splitting of Training and Test Set

To accurately evaluate a model's performance it is important to state how the training and test set is split across the open-source projects in the dataset. Allamanis et al. [22] split each project in a training set (70%) and a test set (30%). Alon et. al [25] used 80% of the projects during training, 10% during validation and the rest for testing. Therefore Alon et. al [25] trained and evaluated the models on different source code projects. Thus, I report the metrics for both datasets.

Chapter 8

Implementation Details

In this chapter, I will discuss the technologies I used and the general structure of my codebase. All the code in this work is written with Python 3. I used the Pandas library to deal with big data tables [10] and the Numpy library to deal with tensors (nd-Arrays) [2]. All token models are implemented with Keras, a high-level API that runs on top of TensorFlow [7]. The Sequence-to-Sequence Sub-token Model is also implemented with Keras. I had difficulties implementing the attention mechanism for the Sequence-to-Sequence with Attention Sub-token Model with Keras and therefore had to use Tensorflow (eager execution) [13]. Moreover, I tried to follow typical system quality attributes, such that the software is maintainable, readable and extensible. All models follow this general structure:

1. Preparation of the training data and training the model or loading a pre-trained model
2. Preparing the test set and evaluating the model

Because each experiment depends on many parameters (input size, model parameters, training parameters) I use an external JSON configuration file which lets me quickly adjust these parameters. Furthermore, for each experiment I run, I create a report folder where I save the most important metrics.
Preparation of the data and the evaluation of the models differ between the token models and the sub-token models. Therefore, I further separate the codebase in a token approach and a sub-token approach.

8.1 Implementation of the Token Approach

In this section, I will explain the code structure of the token approach. The top file to run any experiment is called "run_token_model.py". I use a configuration file to specify which model should be run, set the size of hidden states, determine the size of the input, etc.
The following paragraph is a code walkthrough of a typical experiment, where I first train a model and then evaluate it.

First, I need to prepare the data for training. I do that by creating a dictionary of tokens for the training data and then turn the samples into tensors which serve as the input for the model. To not do these operations every time I want to train and evaluate a model, I save the prepared data (Numpy nd-Arrays) and the dictionary to disk.
I ran into many memory errors when I tried to load the entire training set at once, therefore I split the data in chunks. First, I divided the training data into chunks of the batch size (64), but this resulted in too many I/O operations. Hence, I split the training data in chunks of 32000 samples.

After preparing the training data, I build the token model. I organized the code such that each token model implements an abstract model class. This abstract model class is only a boilerplate and saves the architecture of the model to disk. The abstract method "build_model" is implemented by the individual token model classes.
After the model is built, I train it. To do so, I pass the model to a newly created trainer instance. The trainer object is independent of the specific token model. After training is completed, I save the trained model.
Subsequently, I evaluate the freshly trained model. First, I load the test set. Thereafter, the model makes a prediction for each sample in the test set. I then save the inputs, predictions and ground truth to a CSV file. Afterwards, I compute the accuracy, top-5-accuracy, and F1-Score.
For faster experimentation, I added flags to my main file. It allows me to quickly adjust the configuration file. Table 8.1 illustrates how the flags can be used to control the program with the command line.

Command	Description
python run_token_model.py --model=GRU --epochs=17	Trains the GRU Token Model over 17 epochs
python run_token_model.py --model=LSTM --batch_size=64	Trains the LSTM Token Model with a batch size of 64
python run_token_model.py --model=GRU --data=Java-small	Trains the GRU Token Model on the Java-small dataset

Table 8.1: Examples of running a token model with different configurations (from the command line)

8.2 Implementation of the Sub-token Approach

Command	Description
python run_seq2seq.py --mode=eval	Evaluates on a pretrained Seq2Seq Sub-token Model
python run_seq2seq.py --batch_size=64	Trains Seq2Seq Sub-token Model with a batch size of 64
python run_seq2seq_attention.py --data=Java-small	Trains a Seq2Seq Sub-token Model with the Java-small dataset

Table 8.2: Examples of running a sub-token model with different configurations (from the command line)

In this section, I will describe the sub-token code structure, which is similar to the token approach. For the sub-token approach it was not possible to control everything from one top file because I ran into compatibility issues between Keras and Tensorflow's eager execution. The following paragraph is a code walkthrough of a typical experiment for the Sequence-to-Sequence with Attention Sub-token Model, where I first train a model and then evaluate it.

First, I load the training data. I then build the Seq2Seq with Attention Sub-token Model according to the specified settings in the configuration file. I instantiate the encoder and decoder of the model (which are separate classes). Thereafter, I train the model. Afterwards, I evaluate the model and save the predictions and metrics to disk.
Similarly to the token approach, I use flags for faster experimentation. Table 8.2 gives some examples of how to control the sub-token models with flags.

Chapter 9

Conclusion

Predicting accurate method names based on its functionality is not a trivial task. Intuitively, a good model needs to be able to comprehend the method's type, parameters, and body to meaningfully label it. I suggest 6 different neural network architectures to predict methods across different projects. These models can be categorized in token based and sub-token based approaches. The core idea behind the sub-token based models is to further split the method name into sub-tokens.
I present a quantitative and qualitative analysis which give different insights on the results obtained. The quantitative analysis demonstrates the strength and weaknesses of the different models with different metrics. On the other hand, it has the difficulty of not crediting the models if they can capture the method's functionality with different tokens. For example, if a sub-token model predicts a synonym to the correct sub-token.
The qualitative analysis provides another insight: It illustrates what the models in this work can predict and in which case the models' predictions are useless.
I conclude that the models' effectiveness depends on their practical context. For example, sub-token models are especially useful when predicting on projects previously unobserved. Similarly, if the developer actively asks for a method name suggestion, sub-token models should be used, because their suggestion is often closer to the ground truth especially with more complicated methods.
In a different setting, a more conservative model makes more sense: if the model has been partially trained on a project and should now help to highlight bad method names. In this case, a token model could be sufficient because it tends to predict the UNK identifier more often in unsure situations. Therefore, the model could be configured such that it only highlights particularly bad method names where an obvious better result is at hand.
In general, the RNN token models were less effective than initially anticipated. I expected the RNN models to have difficulties with processing the input because I omit a lot of the code snippet's structure. Therefore, I conclude that for an RNN to have substantial gain over more traditional feed-forward neural networks in analyzing source code, a significant amount of effort has to be made to map the structure of a code snippet to a sequence. This is beyond the scope of this thesis. Alternatives are Tree-LSTMs or graph gated neural networks that are more flexible in learning the input's structure than unidirectional or bidirectional LSTMs. Another difficulty poses decoding for the sub-token models. There, deep reinforcement learning algorithms could be used to help the model make better decisions.
To be of practical use, a tool on top of such a model could be built that serves as an extension in an IDE and a possibility should be given to partially train the model on the current software project.

Bibliography

[1] Attn: Illustrated attention. https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3. Accessed: 03-07-2019.

[2] Base n-dimensional array package. http://numpy.org/. Accessed: 20-07-2019.

[3] Beyond accuracy: Precision and recall. https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c. Accessed: 09-07-2019.

[4] Hyperparameter optimization for keras models. https://github.com/autonomio/talos. Accessed: 05-07-2019.

[5] Introduction to word embedding and word2vec. https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa. Accessed: 22-06-2019.

[6] Java parser and abstract syntax tree for java. https://github.com/javaparser/javaparser. Accessed: 05-07-2019.

[7] Keras is a high-level neural networks api. https://keras.io/. Accessed: 20-07-2019.

[8] Lstm and gru – formula summary. https://isaacchanghau.github.io/post/lstm-gru-formula/. Accessed: 22-06-2019.

[9] Neural machine translation with attention. https://github.com/tensorflow/tensorflow/blob/r1.13/tensorflow/contrib/eager/python/examples/nmt_with_attention/nmt_with_attention.ipynb. Accessed: 03-07-2019.

[10] Python data analysis library. https://pandas.pydata.org/. Accessed: 20-07-2019.

[11] Seq2seq with attention and beam search. https://guillaumegenthial.github.io/sequence-to-sequence.html. Accessed: 22-06-2019.

[12] Suggesting meaningful method names. https://github.com/yrutis/suggesting-identifiers. Accessed: 22-07-2019.

[13] Tensorflows eager execution is an imperative programming environment that evaluates operations immediately, without building graphs. https://www.tensorflow.org/guide/eager. Accessed: 20-07-2019.

[14] Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 22-06-2019.

[15] A very simple convenience wrapper around hyperopt for fast prototyping with keras models. https://github.com/maxpumperla/hyperas. Accessed: 30-05-2019.

[16] What is teacher forcing for recurrent neural networks? https://machinelearningmastery.com/teacher-forcing-for-recurrent-neural-networks/. Accessed: 22-06-2019.

[17] What is the difference between a batch and an epoch in a neural network? https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/. Accessed: 22-06-2019.

[18] What's the difference between a matrix and a tensor? https://medium.com/@quantumsteinke/whats-the-difference-between-a-matrix-and-a-tensor-4505fbdc576c. Accessed: 22-06-2019.

[19] M. Allamanis, E. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51, 09 2017.

[20] M. Allamanis, E. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51, 09 2017.

[21] M. Allamanis, E. Barr, and C. Sutton. Learning natural coding conventions. 02 2014.

[22] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting Accurate Method and Class Names.

[23] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. 11 2017.

[24] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. 02 2016.

[25] U. Alon, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. 08 2018.

[26] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3, 03 2018.

[27] V. Arnaoudova, L. M. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Guéhéneuc. Repent: Analyzing the nature of identifier renamings. *Software Engineering, IEEE Transactions on*, 40:502–532, 05 2014.

[28] Y. Bengio, R. Ducharme, and P. Vincent. A neural probabilistic language model. volume 3, pages 932–938, 01 2000.

[29] C. Boogerd and L. Moonen. Assessing the value of coding standards: An empirical study. pages 277 – 286, 11 2008.

[30] J. Campbell, A. Hindle, and J. Amaral. Syntax errors just aren't natural: Improving error reporting with language models. *11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings*, 05 2014.

[31] X. Chen, C. Liu, and D. Song. Tree-to-tree neural networks for program translation. 02 2018.

[32] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. 09 2014.

[33] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[34] F. Chollet. *Deep Learning with Python*. Manning, Nov. 2017.

[35] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. 12 2014.

[36] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *Under Review of ICLR2016 (1997)*, 11 2015.

[37] Y. Gal and Z. Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. 12 2016.

[38] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks : the official journal of the International Neural Network Society*, 18:602–10, 07 2005.

[39] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[40] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. pages 2073–2083, 01 2016.

[41] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Whats in a name? a study of identifiers. volume 2006, pages 3– 12, 07 2006.

[42] M. Luong, E. Brevdo, and R. Zhao. Neural machine translation (seq2seq) tutorial. *https://github.com/tensorflow/nmt*, 2017.

[43] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[44] T. Mikolov, G. Corrado, K. Chen, and J. Dean. Efficient estimation of word representations in vector space. pages 1–12, 01 2013.

[45] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang. Convolutional neural network over tree structures for programming language processing. 09 2014.

[46] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). pages 574–584, 11 2015.

[47] B. Ray, V. Hellendoorn, Z. Tu, C. Nguyen, S. Godhane, A. Bacchelli, and P. T. Devanbu. On the "naturalness" of buggy code. *CoRR*, abs/1506.01159, 2015.

[48] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45:2673 – 2681, 12 1997.

[49] K. Sheng Tai, R. Socher, and C. Manning. Improved semantic representations from tree-structured long short-term memory networks. 1, 02 2015.

[50] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[51] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 06 2014.

[52] E. W. Høst and B. M. Østvold. Debugging method names. pages 294–317, 07 2009.

[53] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. Yu. Improving automatic source code summarization via deep reinforcement learning. pages 397–407, 09 2018.

[54] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. pages 87–98, 08 2016.

[55] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. pages 334–345, 05 2015.

[56] yrutis. yrutis/suggesting-identifiers: Source Code for the Bachelor's Thesis, July 2019.