Bauhaus-Universität Weimar
Faculty of Media
Degree Programme Computer Science and Media

Generating Code Suggestions

Master's Thesis

André Karge Matriculation Number 110033
Born Aug 01, 1990 in Jena

1. Referee: Prof. Dr.-Ing. Norbert Siegmund
2. Referee: Prof. Dr. Benno Stein

Submission date: April 2, 2018

Declaration

Unless otherwise indicated in the text or references, this thesis is entirely the product of my own scholarly work.

Weimar, April 2, 2018

. . . .

André Karge

Abstract

Synthesizing programs is a long dream of software developers. Although there have been several recent developments in machine learning and software engineering, it has a long way to go. However, there are already important application scenarios which would benefit from source-code generation: code suggestions in Integrated Development Environments (IDEs).
In this thesis, we explore the generation of source-code suggestions by developing a formal framework that allows us to grasp the difficulty of a generation task. Based on that framework, we define our application scenario: generating code suggestions fo Java classes that import and make use of the same Java library *swing*. We train three different neural network models based on different capabilities of incorporating available information defined in our formal framework. We represent source-code as an Abstract Syntax Tree (AST). We show that one of our models is able to generate useful source-code with the support of context information.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For many years, it was the dream of developers to write programs that can generate programs or source code on their own. This has lead to a research field that concentrates on different aspects of code generation or code synthesis: generating software patches to automatically fix bugs [Weimer et al., 2009; Devanbu, 2013], generating new algorithms (e.g. for sorting [Balog et al., 2016]), and proposing code suggestions based on stack overflow threads [Campbell and Treude].
In recent years, the interest on the topic of source-code generation has increased rapidly. For example, [Campbell and Treude], [Bhoopchand et al., 2016], and [Liu et al., 2016] explored the generation of source code to help developers in their development process. Other approaches cover the synthesis of programs from documents [Sethi et al., 2017] or the translation of programs from a programming language to another [Bui and Jiang, 2018].

However, the current state of the art on program synthesis can generate only few lines of code [Balog et al., 2016] and still struggles to find whole programs in the huge space of possible program construct combinations. Hence, we approach the goal of program synthesis from another angle. We focus on a task which is limited to the synthesis of only few program-constructs for which rich information is available: automatic code completion. Modern software development heavily relies on Integrated Development Environments (IDEs), which support developers with special views, code analysis tools, and code suggestions (i.e., automatic code completion). When the programmer writes some lines of code, a popup window appears at the text cursor, providing a set of possible completion suggestions. If a desired suggestion appears, the developer can press a key and the corresponding suggestion is inserted into the editor. This mechanism is mostly based on an internal system containing all classes, methods, and variable names that it can reach from the current position. It is

invoked by the developer inside an editor and does comply to the type at the invocation point.
However, the suggestion system does not use any additional information coming from the semantic context of the program and all suggestions are limited to the current program construct. Clearly, it is beneficial to lift these limitations and provide more accurate and extensive code suggestions to the developer.
Therefore, the general question is what information is necessary to improve code suggestion mechanisms and how can we make use of them? Our approach is based on the following observation: many programs tend to implement similar source-code fragments when they work with the same library. For example, when a Java program uses the *swing* library, frequently recurring tasks are the creation of a frame window or the addition of a button to a class. These similarities can be used when a programmer writes a new program using the same library. The code suggestion system can be enriched with generated code snippets based on the context of the new application (i.e., working with a specific library) and the knowledge of programs snippets with similar purpose. However, existing approaches do not address the scenario of inter-project similarities with the same libraries.

Additionally, the representation of the used source code is important when it comes to code generation (i.e., suggestions). The most common representation is plain text which gives the programmer an easy way of expressing his intentions. Another form that allows for better code analyses is the internal representation after parsing: the Abstract Syntax Tree (AST). For example, [Liu et al., 2016] and [Bui and Jiang, 2018] used an AST instead of plain source code to incorporate the hierarchical structure of code as additional information to generate code suggestions or to translate programs from one programming language to another.

In this thesis, we propose a technique based on neural networks to explore how code suggestions can further be improved. As explained before, we focus on specific libraries and make suggestions only for projects or classes that make use of library functions. Hence, we consider Java programs only. We train our neural network on a large corpus of partial ASTs extracted from a set of Java files that import the same standard library *swing*. It is a library for building graphical user interfaces. In order to extract the partial ASTs, we developed a tool that extracts the sub trees of any desired AST node type (e.g., variable declarations). Those partial ASTs represent the starting point of code suggestions and thus are the input to the neural network. In addition, we need to encode the context of the current program. We encode it as a feature vector to guide the generation of proposals. The feature vector contains

a representation of all methods and variables which are available for a certain AST node type. Next to the training, the feature vector also incorporates the specifics of the library.
The aim of the trained model is to generate new AST nodes that can be transformed back to source code in order to suggest new code snippets when developing *swing* applications.
Our contributions are the following:

1. Development of an AST extraction tool to build and encode a training corpus;

2. Implementation of a basic and an advanced LSTM network; and

3. Evaluation of the networks with respect to their ability of generating new AST nodes that can be used to suggest code snippets.

Chapter 2

Background and Related Work

This chapter focuses on the background information about deep learning on source-code generation and language models and gives an overview to related work.

2.1 Background

In this section, we explain the theoretical background of the applied techniques. It is divided into the background knowledge of the used artificial neural network approaches and language model methods.

2.1.1 Artificial Neural Networks

Artificial neural networks (ANNs) are a machine learning method which is based on the learning of data representations. An ANN consists of different layers which themselves consist of multiple units called perceptrons. In order to understand how a network is able to learn, it has to be specificated how such a perceptron works.

Perceptron

The fundamental theory of the perceptron was developed in 1958 by Frank Rosenblatt [Rosenblatt, 1958].
In simple terms, it is a simplified neural network which models the functionality of a biological neuron. Tom Mitchell describes it as a unit which takes a vector $\vec{x}$ of real-valued inputs of size $n: \vec{x} = x_1, ..., x_n$; calculates a linear combination of these inputs, finally outputs a 1 if the result is greater than some threshold and -1 otherwise [Mitchell, 1997].

More precisely:

$$o(x_1, ..., x_n) = \begin{cases} 1 & if\ w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n > 0 \\ -1 & otherwise \end{cases} \tag{2.1}$$

where w_i is a weight which controls the contribution of the corresponding x_i to the output $o(\vec{x})$ of the perceptron. To simplify the equation, a constant $x_0 = 1$ allows to express the inequality of 2.1 as $\vec{w} \cdot \vec{x} > 0$.
Here, the application of a threshold is called activation function. It controls the output of the perceptron with a simple step function by returning a defined value for a certain input range and another value for a different input range. A visualization of the functionality of the perceptron can be seen in figure 2.1.

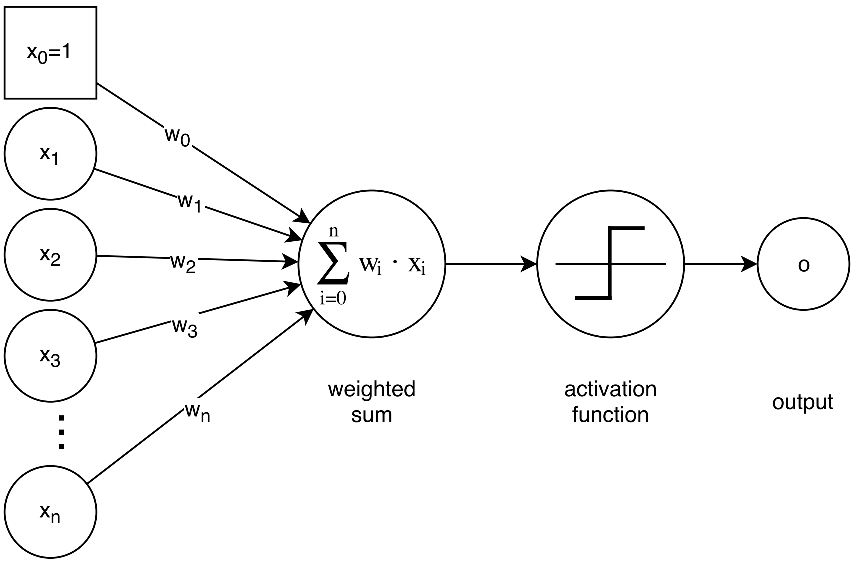


Figure 2.1: Diagram of a perceptron

In order to learn with a perceptron, appropriate values for $\vec{w} = w_0, ..., w_n$ have to be chosen. The space H of all candidate hypotheses for weight vectors $\vec{w}$ is the set of all possible real-valued weight vectors [Mitchell, 1997]:

$$H = \{\vec{w} \mid \vec{w} \in \mathbb{R}^{(n+1)}\} \tag{2.2}$$

The usage of perceptrons allows to create a hyperplane through the n-dimensional space of instances which classifies points on each side of it as 1 or -1. If a set of examples can be classified that way, it is called a *linear separable set of examples.*

But, there are sets which cannot be separated linearly by a perceptron, for example the result of the binary XOR-operator. This was analyzed by Marvin Minsky and Seymour Papert [Minsky and Papert, 1969] in 1969 which is why the research on artificial neural networks was declined by many researchers until the 1980s.
However, with the introduction of the Backpropagation algorithm [Rumelhart et al.] in 1986, the problem could be avoided by using multi-layered perceptrons in combination with Backpropagation.

Multi-Layer Perceptron

A multi-layered perceptron is a kind of an ANN where multiple perceptron units are stacked together to form a meshed network. An example of such a network is shown in figure 2.2. It consists of at least 3 layers: an input layer,

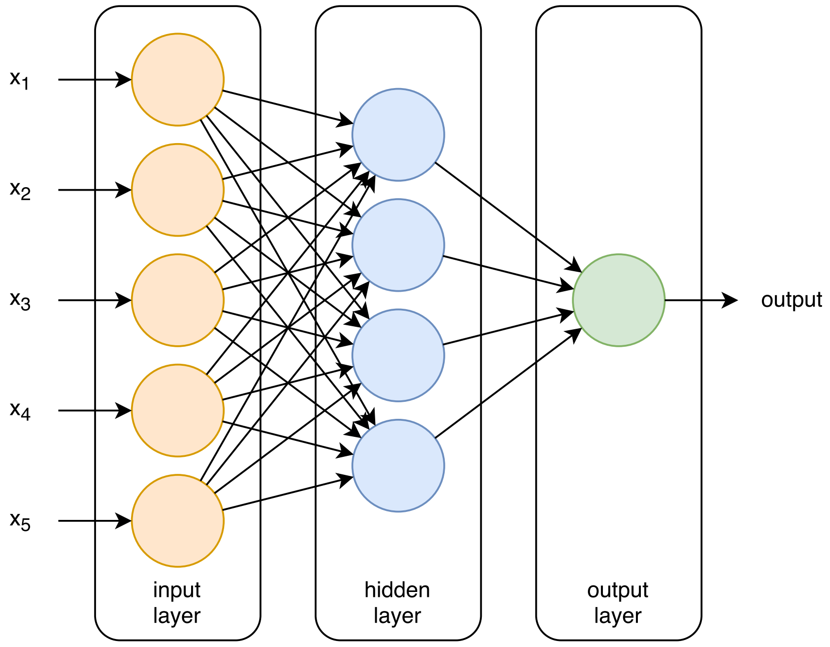


Figure 2.2: A fully-connected multi-layer perceptron with 5 input values and one output value

one or more hidden layers, and an output layer. Each layer consists of a defined set of perceptron units which can also be called *neurons*.
An input layer consists of i input neurons where i denotes the dimension of the input parameter. Input neurons forward only their feeded value to all neurons which they are connected to because they do not have an activation function. The input is simply weighted, summed and then forwarded.
Between the input and output layers are the hidden layers. They are called

hidden due to the fact that they are not visible to an external user of the network.
An output layer consists of o output neurons where o denotes the dimension of the desired output values. As the result of the network, the output neurons produce a signal in form of a value or a vector which has to be interpreted.
There are many types of ANNs, for example, fully connected networks, recurrent networks, convolutional networks, and more. The most common type is the fully connected network. In this type, each neuron of a specific layer is connected with each neuron of the next layer.
Each connection between two neurons models a weighted information pass which is visualized in figure 2.3. The key concept of learning is to adjust the

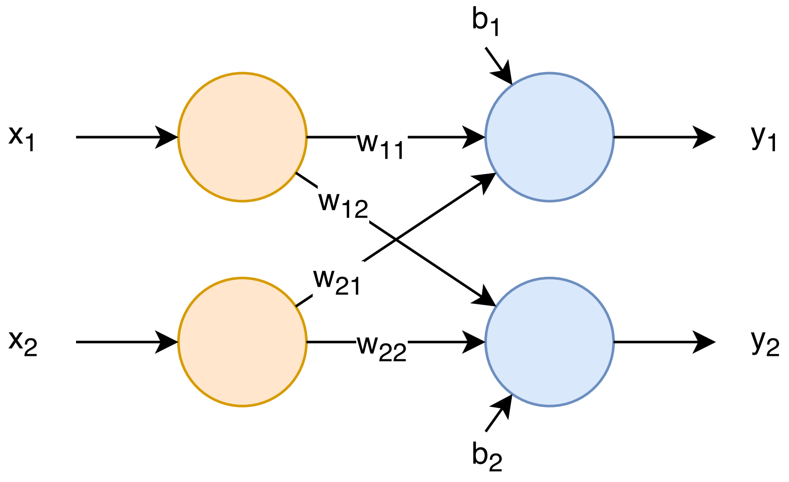


Figure 2.3: Diagram of the influence of weights and biases in an ANN

weights in the training process. An additional bias value for each hidden neuron can control how sensitive the neuron is and at which value it starts to give an output. In order to calculate a forward pass of a layer, the following equation is used:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = activation(W \cdot \vec{x} + \vec{b}) = activation\left(\begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}\right) \tag{2.3}$$

An output neuron can also be considered a hidden neuron after the terminology of Christopher Bishop [Bishop, 2006]. The network in figure 2.2 is a 2-layer network because the amount of weighted connection layers are important for determining the properties of a network.

A major difference of the multi-layer perceptron to the classical perceptron is that it does not use a step function as activation and can use any function that is continuously differentiable; for example, a sigmoidal function (equation 2.4).

$$sig(t) = \frac{1}{1 + e^t} \tag{2.4}$$

This activation function enables the application of the Backpropagation algorithm since the activation function is continuously differentiable.

Recurrent Neural Networks

Recurrent neural networks (RNN) are a good option when it comes to learn on timed series such as text sequences. A property of RNNs is that the hidden nodes share information across parts of the model. Figure 2.4 illustrates a basic network with one RNN layer. The layer consists of a RNN unit which

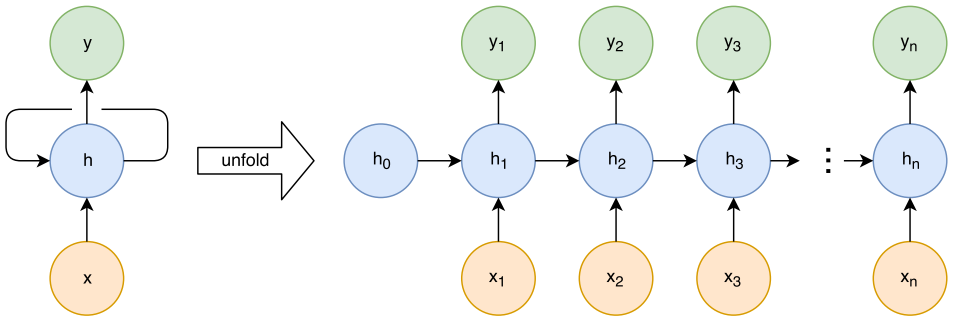


Figure 2.4: Illustration of a simple RNN with its unfolded representation

cycles through a defined amount of iterations (left side). In order to get a better visualization, the unit can be unfolded (right side).
It takes an input sequence $x = (x_1, x_2, x_3, ..., x_n)$ of size n and predicts an output $y = (y_1, y_2, y_3, ..., y_n)$. At each time step $t \ : \ t \in n$ the next prediction y_t is based on the current input x_t and the previous hidden unit state h_{t-1}. The usage of the previous hidden state in the calculation of the next prediction enables encoding the knowledge of past sequence elements.
Given the incomplete sentence “I like machine learning because it is ...”, an RNN should predict the next element which is likely to be “awesome”. But if the sentence now is slightly changed: “I hate machine learning because it is ...”, it should result in a different prediction since the word “hate” is at the second position and the RNN encodes that knowledge in its hidden state.

There are different types of RNNs: many-to-many, many-to-one, one-to-many, and one-to-one. The terminology of the types refer to input-to-output elements, so many-to-one would have many input values x but only one output value y. Hence, the network of figure 2.4 is a many-to-many network since for each time step t, an output y_t is generated using x_t and h_{t-1}. A one-to-many network could be, for example to predict a music sequence based one a value which encodes the genre of the music. The many-to-one architecture can be used to generate the next element of a given sequence or to classify the sequence.
A weakness of classic RNNs is the vanishing gradient problem. It describes the fact that when a network becomes bigger it is harder for the Backpropagation algorithm to affect the weights of the first nodes. That means, the influence of the element at $t_{i-j} \mid (j > 0)$ to the prediction at t_i becomes smaller the greater j becomes.
For example, given a long sentence: "The games which I had as a child and played with my friends and ... was very funny". Remembering that "games" was written in plural is hard for the RNN which should result in a "were" and not a "was" at the end of the long sentence.
The solution for the vanishing gradient problem is the introduction of an another type of RNN unit: the LSTM unit.

LSTM

A Long Short Term Memory (LSTM) unit is a kind of an RNN unit which aims at solving the vanishing gradient problem of RNNs. It was developed in 1997 by Sepp Hochreiter et. al. [Hochreiter and Schmidhuber]. With the usage of this unit, a network is able to remember long range dependencies of elements.
A LSTM unit consists of memory cells and different gates: a forget gate Γ_f, an update gate Γ_u and output gate Γ_o. In order to apply Backpropagation, the gates are defined by a sigmoidal function σ. Each gate has its own weights W and biases b which are adjusted during training, providing the unit the ability to learn when it should remember or forget certain information.
Figure 2.5 contains a diagram of a LSTM cell. It shows three inputs x_t, a_{t-1} and c_{t-1} where x_t stands for an element of the input vector x at time t. The values c_{t-1} and a_{t-1} are the two hidden values of the LSTM which are the output of the previous cell. c is a memory cell and a is a activation cell (which is equivalent to the hidden state of the classical RNN).

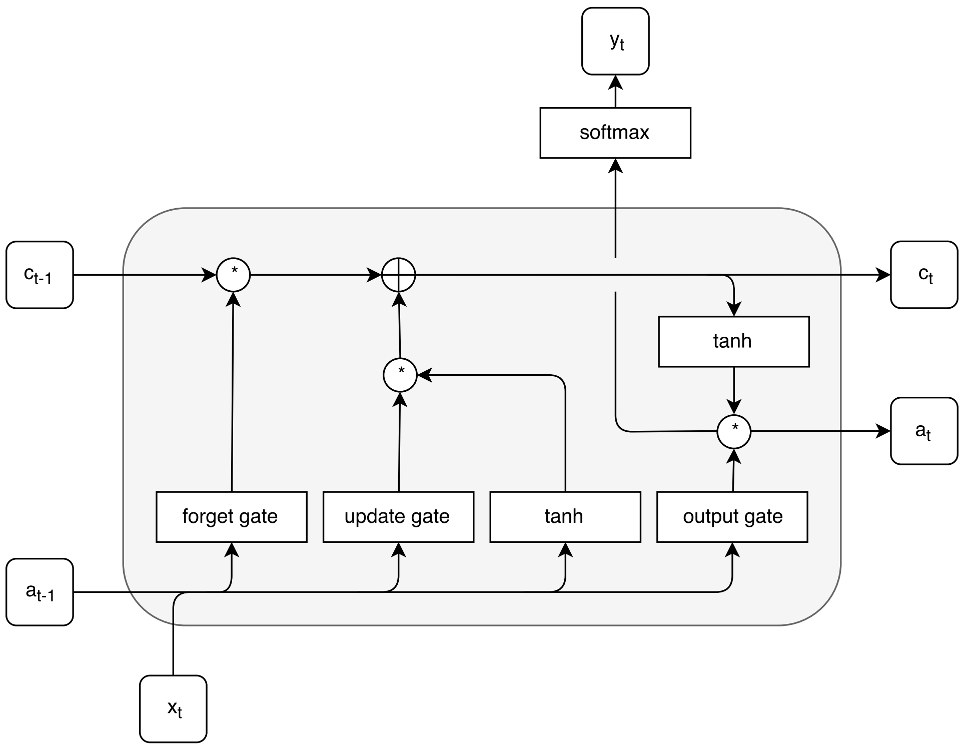


Figure 2.5: Diagram of a LSTM cell

All Equations of the LSTM cell are shown in eq. 2.5.

$$\begin{aligned} c_t^n &= tanh(W_c[a_{t-1}, x_t] + b_c) \\ \Gamma_u &= \sigma(W_u[a_{t-1}, x_t] + b_u) \\ \Gamma_f &= \sigma(W_f[a_{t-1}, x_t] + b_f) \\ \Gamma_o &= \sigma(W_o[a_{t-1}, x_t] + b_o) \\ c_t &= \Gamma_u \cdot c_t^n + \Gamma_f \cdot c_{t-1} \\ a_t &= \Gamma_o \cdot tanh(c_t) \end{aligned} \tag{2.5}$$

tanh is the activation function for the new memory value c_t^n.
The existence of the memory cell c provides the LSTM the ability to store information over a long range of time steps.

2.1.2 Information retrieval

To model the input of a RNN, a language model has to be defined beforehand. This section deals with the formal definition of language models, gives a short outlook on the word embedding model and shows how documents can be compared.

Language Models

A Language model is a probability distribution over a sequence of text elements such as characters, words or word pairs.
It consists of a finite vocabulary set V containing all elements of the language. Using V, one can build a sentence x of elements $< w_1, w_2, ..., w_n >$ with a sentence length of $n \geq 1$ and $w_i \in V$ for $i \in \{1, 2, ..., (n-1)\}$. Hence, we can build an infinite set V^+ which is the set of all possible sentences x under different sentence lengths n. This gives us a probability distribution p where its properties are defined in equation 2.6 [Collins, 2018].

$$\begin{aligned} &\text{for any } x_1, x_2, ..., x_n \in V^+ : p(x_1, x_2, ..., x_n) \geq 0 \\ &\text{and} \sum_{x_1, x_2, ..., x_n \in V^+} p(x_1, ..., x_n) = 1 \end{aligned} \tag{2.6}$$

Language models can be applied on different levels, e.g.: on character level or on word level. One of the simplest language models is the probability distribution over character level sequences. Its vocabulary is very small since it contains only of 26 different elements of the latin alphabet and some special characters like whitespaces. But, the amount of characters can differ from language to language. For example, chinese has different alphabets with more than 7,00 characters.
It is also possible to use different sizes of sequence lengths for each vocabulary element. A sequence of elements with length n is called a n-gram. Models using this form are called n-gram language models. On character level, 3-gram examples for the sentence "hello world" would be: "hel", "ell", "llo", "lo ", "o w", " wo".
When a language model is created on word level, the vocabulary size becomes big since every word of a language has to be stored. This is why it is not possible to store every word in the vocabulary. A good way to overcome this problem is to sample the frequencies of all words in the given documents and only keep those with a frequency higher than a certain threshold or only keep the top-k frequent words. If a word occurs which is not in the vocabulary, a special element has to be added which models all unknown words. This can be for example: "<UNK>".

Like on character level, it is possible to create n-grams on word level, for example this 3-grams of the sentence “the quick brown fox jumps over the lazy dog”: “the quick brown”, “quick brown fox”, “brown fox jumps”. n-grams can encode semantic information for words which occur often in pairs. For example, in a text about american cities, the 2-gram “Los Angeles” is more likely to occur than the two unigrams “Los” and “Angeles”.
A challenge for word level language models is how to define a word. With English, a word in a sentence is commonly surrounded by white spaces or punctuation. But other languages may define a word in different way. Hence, the definition of a word is highly dependent on the language itself.

Word Embedding

Word Embedding is a language model where words of a given vocabulary are represented as vectors. A basic approach for word embedding is a simple index mapping. For example, the sentence: “the quick brown fox jumps over the lazy dog” which consists of 8 different words builds the vocabulary:

[“the”, “quick”, “brown”, “fox”, “jumps”,“over”, “lazy”, “dog”]

A vector representation of the word “fox” then would be a mapping to the word in the vocabulary:

$$[0, 0, 0, 1, 0, 0, 0, 0]$$

However, this representation only identifies what a word is and not what it means. With another embedding approach it is also possible to encode the meaning of a word as part of speech. This approach is called Word2vec and was created by Google Research in 2013 [Mikolov et al., 2013]. It makes use of the distributional hypothesis which makes the assumption that words which appear in similar contexts are related to each other.
Word2vec provides two architectures: the continuous bag-of-words (CBOW) and the continuous skip-gram. The CBOW model encodes a word based on its surrounding context and the continuous skip-gram encodes surrounding context based on a given word. The size of the surrounding context can be defined as a parameter n of the model.
Using CBOW, the example sentence would build the following tuple (at context size 2 for each side):

[“the”,“quick”,“fox”,“jumps”],[“brown”]

As can be seen, the word “brown” has the context of the words “the”, “quick”, “fox” and “jumps”.

This encoding then is used to build a dense vector representation of the words. In the vector space, words with similar meaning are close to each other (which can also be seen in figure 2.6). With a trained Word2vec embedding it is also

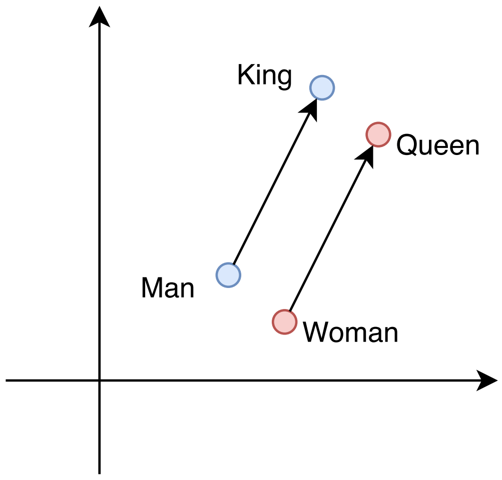


Figure 2.6: Visualization of word embeddings

possible to perform calculations in the resulting vector space, since analogies are encoded in differences between vectors:

$$v_{king} - v_{man} + v_{woman} \approx v_{queen}$$

TF-IDF and Cosine Similarity

A common way to measure the similarity of documents is the usage of cosine similarity. It is a method to compare two documents on their equality by using a vector representations of the documents.
To vectorize a document, the Term Frequency - Inverse Document Frequency (TF-IDF) weighting is used. TF is a score of how often a term t occurs in a document D, e.g. given a document with the text “blue blue blue red”, the $TF(\text{“blue”}, D) = 3$ and $TF(\text{“red”}, D) = 1$. The term scores are normalized for each document based on the term with the highest occurence in the document $max_{t' \in D}$:

$$TF_{norm}(t, D) = \frac{TF(t, D)}{max_{t' \in D} TF(t', D)} \tag{2.7}$$

IDF measures the relevance of a term t in the given set of documents:

$$IDF(t) = log \frac{N}{df_t} \tag{2.8}$$

where N is the number of documents and df_t the amount of documents containing the term t. As a result of equation 2.8, rare terms get a high IDF value and frequent words get a low IDF value. To weight the terms for each document, TF and IDF are combined to TF-IDF:

$$TF\text{-}IDF(t, D) = TF(t, D) * IDF(t) \tag{2.9}$$

It is the weight of a term t in a document D based on the overall term frequency multiplied with the normalized frequency in the given document.
Now, we can view each document D in the set as a vector $\vec{D}$ with the length of all words in the vocabulary where each word is weighted based on the TF-IDF value for the document. Those vectors can be compared by computing the cosine similarity between them:

$$cossim(D_1, D_2) = \frac{\vec{D_1} \cdot \vec{D_2}}{|\vec{D_1}||\vec{D_2}|} \tag{2.10}$$

For example if we compare the vector of a document to itself, the cosine similarity would result in 1 since the cosine between the same vectors is one. If two document do not share a single term, then the similarity is 0 since the cosine between two orthogonal vectors is zero.

2.2 Related Work

The generation of source code can be applied in different ways. One way is to train a neural network which then generates code for a given task. Other approaches apply genetic programming on their tasks.
Additionally, the field of source-code generation can be divided into areas with different goals. For example, it can be used to help programmers with their efficiency while programming to speed up the whole programming process. In contrast, machines can be used to automatically generate code to approximate the behavior of programmers.
The first section deals with approaches which use neural networks to solve different source-code generation tasks. After that, a short outlook on genetic programming is given. Finally, other approaches are outlined.

2.2.1 Code Generation via Neural Networks

Today, generating text structures with sequence learning by neural networks is a commonly used practice. However, source-code generation is a relatively new field inside the topic of sequence learning. There are different approaches on code generation which involve the usage of neural networks.

Recently, the interest in source-code generation increased. The article [Bhoopchand et al., 2016] proposes a guidance on how to generate Python code suggestions by analyzing the plain source code. A similar approach was made by [Liu et al., 2016] who used the AST instead of plain text on JavaScript source code. The work of Bhoopchand et. al. aims to improve the efficiency of programmers on their programming flow by using a deep neural network with attention models [Bahdanau et al., 2014]. An attention model is an improvement of an so called encoder-decoder RNN network [Sutskever et al., 2014].
The encoder steps through the input sequence, encoding it into a vector of fixed length and the decoder steps through the output sequence. Due to limitations of the fixed vector length, long sequences are harder to decode. Attention does not encode the input to a single vector with fixed size but uses a vector of the last K vectors which are filtered for each output step [Bhoopchand et al., 2016; Bahdanau et al., 2014].
The authors wanted to generate code suggestions for IDEs to give programmers a better integrated code suggestion tool. They state that code suggestion is the most useful feature, especially when the programmer has no knowledge about the codebase. For this context, a corpus of 41 million lines of Python code was created. It was achieved by using a crawler which acquired projects from Github[1] using a heuristic. This heuristic collects code based on the popularity of repositories. Github provides two metrics for that: the amount of stars (bookmarks) and the number of forks (user copies) of a repository.
Their model aims to exploit long-range dependencies of identifiers with selective attention on AST elements via a sparse pointer network. A sparse pointer network is an adaption of the attention model in order to have access to a large history of Python tokens. It enables a modelling of long-range dependencies of identifiers.
A vocabulary was created based on the words of the acquired corpus. Due to the high amount of variable identifiers, a normalization was introduced, replacing the identifiers with more meaningful ones.
As a result, they found out that their model is able to outperform n-gram language models. With the addition of attention, the model improves even further on the accuracy of identifier prediction from 2.1% to 30.2%. But their approach is incomplete, since it only is a proof of concept and needs to be applied into IDEs.

The approach of [Liu et al., 2016] aims at creating code completion with dy-

[1]`https://github.com` - accessed April 2, 2018

namically typed languages. This work uses the AST for the dynamically typed programming language JavaScript. The main difference to the previous work [Bhoopchand et al., 2016] is that they want to predict sequential nodes by traversing the AST. They use the fact that non-terminal nodes can have both non-terminal nodes and terminal nodes as child, but terminal nodes cannot have any children.
The work uses the partial AST to train a neural network in order to predict nodes based on the current pointer position in the tree. This allows to complete block code structures rather than only text tokens. In their experiments, a corpus of JavaScript files was used which was provided by the research of Veselin Raychev et. al. [Raychev et al., 2016]. On that data, they trained a single layer LSTM network. To evaluate results, a code completion engine for the IntelliJ IDE was proposed.
They conclude that their model was able to outperform the baseline model of Veselin Raychev et. al. by 3.8%. Also, the model worked better with longer source-code sequences than with shorter sequences.

In contrast to the previous works, [Mou et al., 2015] envisions the scenario of generating code using RNNs in order to generate complete programs. They wanted to let users express their intentions and pass them to an RNN which then generates source code based on these intentions. This work aims to approximate programmers behavior, since it tries to generate a whole program based on a given task.
As a basis, the model of [Sutskever et al., 2014] was used which takes character level token sequences as an input and tries to predict the next character tokens. A disadvantage of this kind of network is that it only can generate text that looks like real source code but cannot be compiled.
For their scenario, they acquired code from a pedagogical online judgement system in which students submit their versions of programs which solve a specific task. The submissions then are automatically judged by the system itself. With the resulting corpus, a set of experiments was conducted. They trained the network with a programming task as input and labeled it with the source code of the programs which solve the task.
After running the experiments, it was verified that the code could not be compiled. None of the generated source codes were similar to code from the corpus, but it seemed that they were composed of the elements which solve the problem. Also, with few changes they were able to be compiled. The final conclusion of this work is that writing programs by humans differs significantly from RNN generated programs. However, the authors envision that is could be possible in the future to generate programs which solve a given task.

The research of [Balog et al., 2016] deals with an approach on training a neural network in order to predict properties of a program which generates a defined output from a given input. This approach focuses on the generation of complete programs like the previous work of [Mou et al., 2015].
Two ideas were introduced in this work. The first idea was to train a neural network in order to induce programs. This should be accomplished by learning strategies over a corpus of program induction problems in order to generalize over problems. The second idea was to use neural network architectures in combination with search-based techniques.
They call the approach of their work LIPS (Learning Inductive Program Synthesis). LIPS consists of 4 general components:

1. a "domain specific language" (DSL) with specific restrictions to shrink the search space,
2. a procedure for generating data,
3. a machine learning model, and
4. a procedure which searches the program space

Based on that general definitions, the tool DeepCoder was introduced. It uses a DSL which is inspired by the Structured Query Language (SQL) and Language Integrated Query (LINQ) with only a small set of functions, proposes a way of generating input-output data using the DSL and introduces an encoder-decoder neural network to train on the generated data. Instead of predicting source code, the authors wanted to use a neural network to guide the search process for a program.
With this specifications, they conducted experiments and compared the results with a baseline which consisted of probabilities of the functions global occurrence in the corpus. This showed that DeepCoder was able to improve the runtime synthesis for inductive programming at least by a factor of 3 compared to the baseline. However, the approach can only synthesize simple problems of code competition sites due to the restricted DSL. A great amount of problems need more complex solutions than the DSL could provide.

In order to propose a formal guide on the creation of source-code generation frameworks, [Alexandru, 2016] explored the adaption of code synthesis techniques in the means of classification, comprehension and completion.
To create a formal guide on creating source-code generation frameworks, the author proposes three research questions. First, how could source-code examples be identified, classified and parsed to build a trainings set; second, how to find a suitable neural network architecture; and finally, how users should

interact with the model when developing in an IDE. A solution for the first question is outlined by finding snippets online and isolate them from other content, classify context and used language of the snippet and parse it, although it might be noisy. To deal with incomplete or noisy snippets, the usage of a RNN sequence-to-sequence network should help to parse them into a partial AST. The author proposes a mixture of bi-directional RNNs for the work. Additional context information should also be used for giving the network a strong bias on the selection of an architecture. However, finding an appropriate model is still the topic of current research.
Finally, the author assumes how an implementation could look like: a completion system that uses the knowledge of internet sources and local code information integrated into an IDE with the possibility to iterate through different completion candidates.
To evaluate models using this approach, a comparison of the ability to predict elements to other approaches or a user study on the result should be conducted. A user study should measure the time to solve a task on the proposed tool compared to other auto-completion tools or the usage of web resources.

Additionally, there is a variety of research work close to the field of code synthesis with neural nets. For example, the research of [Bui and Jiang, 2018] tries to learn the translation from one programming language to another. For that, they want to learn a cross language representation of two programming languages by using the hierarchical structure of ASTs and shared word embeddings. The practical use of this work, next to inter-language-translation, could be to classify code plagiarism or even synthesize code of a given language in another language.

Another approach is the work of [Sethi et al., 2017]. The aim of their work is to synthesize deep learning networks based on extracted network graphs from scientific research papers. The resulting tool scans a paper in pdf format, searches for relevant figures and tables of network designs, extracts the flow of the proposed model, builds an abstract computational graph and generates the source-code from that graph in Keras and Caffe. An evaluation of their approach resulted in a accuracy of 93% on extracting graphs from figures.

2.2.2 Code Generation via Genetic Programming

Genetic programming is a technique where parts of the source code are encoded as genes. These genes then are updated or changed by using an evolutionary strategy which is inspired by the biological term of evolution. As a result, a set of programs or code snippets is mutated until it can perform a given task.

The work of [Weimer et al., 2009] introduces a genetic program which tries to automatically find bug fixing patches for programs. The motivation behind this program is that it can be quite expensive to debug or improve software. Sometimes, new bugs are reported faster than the developer can fix them.
The input of the tool only consists of a program, a set of successful test cases and one failing test-case which marks a program defect. But since the search space of evolutionary algorithms is potentially infinite, they restricted their program. It should only use other structures of the given program to produce changes and mutate the program only at the areas which are relevant to the given problem.
The programs are represented in the form of an AST paired with a weighted program path. To change the programs, the following genetic operations were used: selection, mutation and crossover. They are also called *breeding* operations. The application of each operation creates a new AST which is compiled and checked against the test cases to assess its fitness. The fitness is calculated by the summing and weighting up the positive and negative test-cases that are passed.
In simple terms, the selection operation selects those programs with the highest fitness value and mutation changes parts of given programs to create new versions of them. Crossover is an operation which takes two programs, applies some operations on both and gets two changed results.
Finally, in order to evaluate the tool, they conducted experiments on open source benchmarks from several domains with known defects. As a result of the experiments, the tool found patches for the given programs with an average success rate of about 58% in under 200 seconds. However, the resulting patches highly depend on the given test-cases. That means, if the test-cases are poorly designed then it becomes harder for the tool to find an appropriate patch.

2.2.3 Other Code Generation Approaches

Source code can also be generated without the use of genetic programming or neural networks.
To improve the efficiency of programmers, [Campbell and Treude] addressed the problem of unnecessary context switches during the development process. The process of switching from an IDE to a web browser in order to look up a problem, search for solutions, switching back and insert the acquired data into the IDE implies too many context switches and may lead to a loss of concentration and productivity [Proksch et al., 2013].
In general, the research focuses on three points to solve that problem: create a

code snippet assist feature, integrate Stack Overflow snippets seamlessly into an IDE and evaluate a predefined set of intention queries.
The resulting tool NLP2Code is a content assistance plugin for the Eclipse IDE which provides code snippets based on a natural language intention of programmers. The plugin is integrated in the Eclipse code completion mechanism and tries to close the vocabulary gap between the programmers intention and Stack Overflow threads. Users can type their intention as plain text or use the code completion mechanism to get an intention query proposal. This query then is used to search for relevant Stack Overflow threads which are parsed. The source codes from relevant pages are retrieved and printed into the Eclipse Editor. It is also possible to switch through all results, enabling the user to search for the snippet which is most accurate for his or her needs. To evaluate their tool, a study with 10 participants was conducted which led to the conclusion that it helps programmers to find relevant code snippets without switching context. However, the evaluation made no assumption about time savings compared to the regular development pattern.

Chapter 3

Generating Code Suggestions

This chapter describes the overall design of the work.
First, we introduce the application area and the proposed workflow Afterwards, we describe our used corpus and explain why we rely on ASTs instead of plain source code and how the corpus is processed. Finally, we describe architectures of our neural network that we use for generating AST nodes.

3.1 Application Area

Generating source code on its own by machines is the dream of many software developers. But how should this be achieved? Which programming languages should be covered? Which group of users should be targeted in this approach? Next, we formalize the scopes of our work to frame the work in a bigger picture that draws the range of general code synthesis tasks.
We can hypothesize that the ability to generate new source code depends on several factors:

a) the language for which the code should be generated,

b) the availability of training examples for the selected language, and

c) the difficulty of the generation task.

Since each aspect spans a range of possibilities, we need to select a feasible and yet practically relevant setting. We define the difficulty of generating the correct code Θ as follows: $\Theta = \sigma + \gamma + \alpha$, where σ is the task difficulty, γ is the amount of available information, and α represents the ratio of what we know about the intention of the programming task. Let us describe all elements in further details.
The task difficulty σ depends on two factors: the domain complexity of the

task and the language in which the task should be realized. The domain complexity denotes the difficulty of grasping the problem and expressing the solution in a proper way. For example, writing a GUI for a Web shop is trivial in comparison to a scientific simulation. However, the used language interacts with the domain problem as the Web shop implementation might also be hard when Haskel is used.
When generating source code, it is clear that different programming languages have different program constructs and syntax, which also influence the difficulty of the programming task σ. For example, Haskel is entirely different to Java and Java to Prolog. Even languages that follow the same paradigm can have different language constructs, such as Java, C++, or JavaScript. Again, a task can be trivial to formulate in one language, but hard in another language. We define σ as the tuple (d, l), where $d \in D$ and $l \in L$. Let L be the set of all programming languages and let $l \in L$ be a tuple $l = (G, C)$, where G represents the grammatical representation of the language and C the set of higher-order language concepts. We further denote D as a set of numbers indicating the difficulty of the domain task. Furthermore, we note that a concrete task might have a different d for different users with different background knowledge. But here, we need to come to an objective difficulty classification; say, an average developer.
The second parameter for the difficulty of generating code is $\gamma = \gamma_{feature} + \gamma_{context}$. Clearly, the more information is available the more correct should be the generated code. It represents the information of features $\gamma_{features}$ and the information of context $\gamma_{context}$. With $\gamma_{features}$, we denote the current known variables, types, and callable methods, and with $\gamma_{context}$, we denote partial AST from which the next AST nodes should be generated. A very hard task is therefore, when omit all available information. However, the exact definition of γ is still a matter of research. For our work, we assume $\gamma_{feature}$ and $\gamma_{context}$ have the same contribution to γ.
Parameter α represents a ratio $\alpha \in [0, 1]$ that denotes how much we know about the intention of writing the current code. A ratio of 0 means that we know nothing about the task we should perform. This means that we have never seen before similar tasks. For example, consider that we have trained a neural network to generate variable declarations. Now, the task is to generate a method declaration. Here, we have no prior training data and the neural network would mistakenly try to generate a variable declaration. By contrast, a ratio of 1 means that we have an exact specification of the task and also have seen such a task in prior training runs such that we know the intention of the to be generated code. However, it is clear that a complete formalization of all aspects need further work, but lie outside of this thesis.

package	description
java.io	provides functions for standard input and output such as file write and read
java.lang	an automatically imported library providing indispensable classes like *String* or *Thread*
javax.swing	provides components for graphical user interfaces
java.util	provides types for data structures or functions connected with time

Table 3.1: Examples of packages in the Java standard library

Based on the formal framework above, we specify the concrete setting that we target in this work. We took Java as our target language l due to several reasons. It is the most common programming language according to the PYPL[1] and TIOBE[2] indices. They are ranking pages which rank programming languages or IDEs based on different search engine requests. This means that we have a vast amount of training data at our disposal positively affect both γ and α.

When a developer wants to write a new program in Java, he is most likely to use a bunch of libraries. Our assumption for this work is that all programs which import the same library are similar to each other and share a large part of their source code. There are two kinds of libraries in Java: standard class libraries and user-defined libraries. The standard class library is part of the core Java language and provides packages which are always available for each program and commonly used. It consists of 217 packages in total. Some selected standard packages can be seen in table 3.1.
To increase α (i.e. the ratio at which we know the intention of the to be generated code), we concentrate on Java classes that all make use of the library *javax.swing*. The idea is, when a programmer uses this import he is likely to write similar code to that other developers already implemented. Sometimes, developers reach a point where they do not know how to continue programming and search online for a solution [Campbell and Treude]. It is possible that they find other projects doing a similar thing and copy code snippets into their own code.
For example, when a user implements a *JButton* he will probably register an *ActionListener* to it. Since this is often the case, a suggestion plugin would analyze the program and provide a generated snippet based on the given con-

[1]`http://pypl.github.io/PYPL.html` - accessed April 2, 2018
[2]`https://www.tiobe.com/tiobe-index` - accessed April 2, 2018

text. The proposed *ActionListener* snippet then could include calls to methods which the user already implemented in this project.

3.2 Workflow

We divide the task of generating source code into different modules:
a AST extraction module, a *swing* extraction module and a code generation module. Figure 3.1 provides an overview of the modules and how they interact.

The first module analyzes the corpus and is described in section 3.3. Next, we implemented a java tool to extract AST nodes from the corpus. As a result, a dataset containing file pairs of serialized partial ASTs and its corresponding features was created. The details of this module are given in section 3.4. Additionally, we extract features of the different swing library classes and the corpus projects as explained in section 3.4.4.
Finally, we implemented several deep learning models using the data of the previous modules. The functions and structure of the models are explained in section 3.5.

3.3 Java Corpus

In order to train a deep learning model for code generation, a large corpus of training data has to be used. To this end, we reused a dataset that has been acquired by Anne Peter [A. Peter, 2017]. She created a crawler in order to get projects from Github using the provided API.
Github is a Web service based on git which is a version control system for source-code management. It enables users to commit changes to their projects and later to switch back to a earlier revision if he wants to check out an older version. There are nearly 12,000 repositories on Github[3] when using the search tag *swing* under the restriction that the repository mainly contains Java source code.

Only the first 100 resulting projects were acquired. Afterwards, they were cleaned up and scanned for an actual import of *javax.swing*. This resulted in 88 projects which build the *java-swing*-corpus.
Each project folder has several revisions included which differ from one to 33 revisions. All projects were filtered, picking only *.java* files containing the keyword *"import javax.swing"* from the last revision. This resulted in a total

[3] `https://github.com` - accessed April 2, 2018

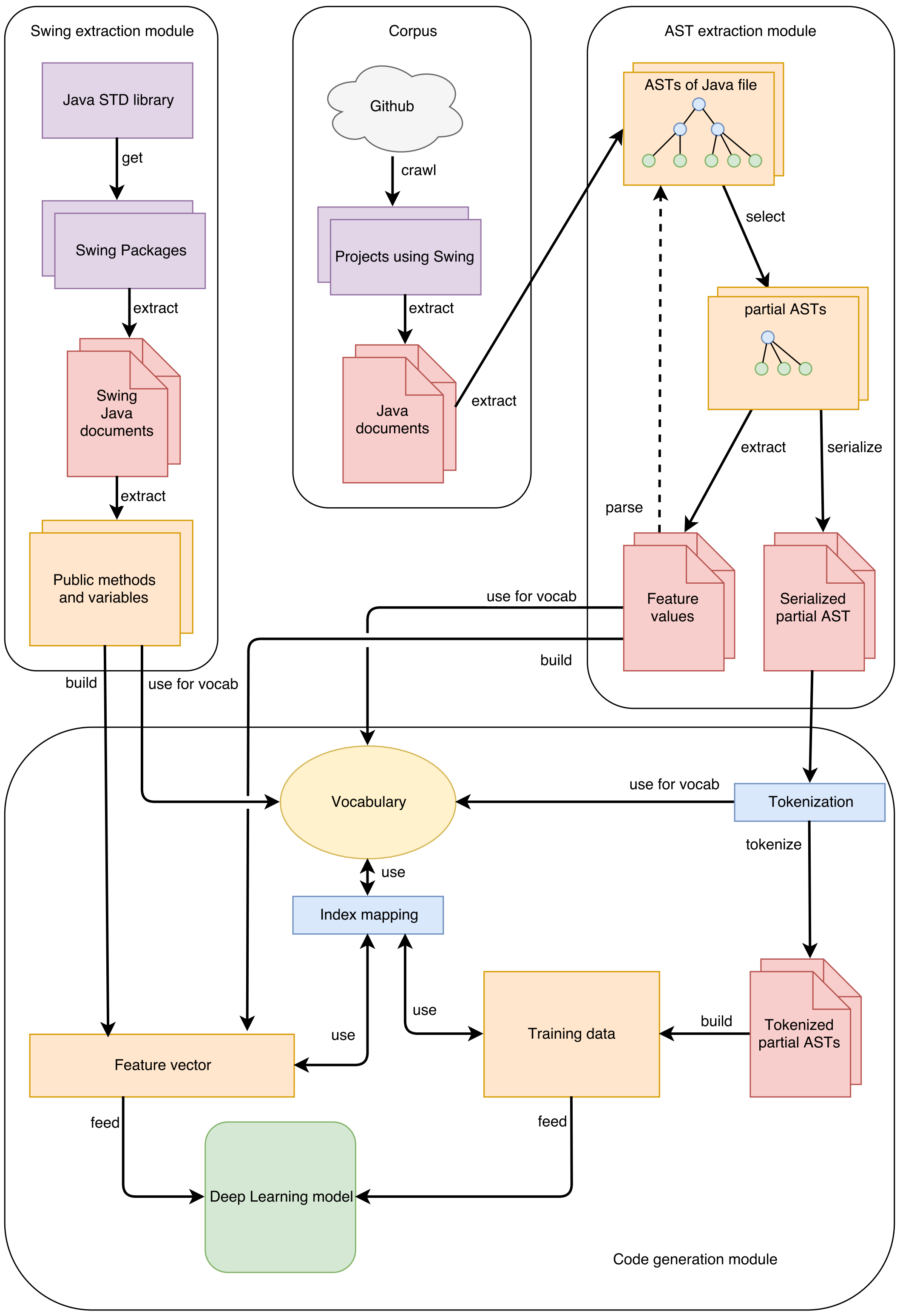


Figure 3.1: Overview of the workflow

	lines of code	difference to raw
raw *.java* files	1.161.833	-
without empty lines & indents & comments	718.671	443.162

Table 3.2: lines of code: raw vs. cleaned

amount 4,509 raw *.java* files.
Anne Peter further process this raw corpus to obtain three different variants: without comments, without empty lines and indents, and without empty lines, indents and comments. Table 3.2 shows how many lines of code are present in the corpus in total.
But, the representation as plain source code has the problem that it is a high level abstraction which aims at helping developers create programs. In this work, we make use of the AST instead.

3.4 Preprocessing Representation: Abstract Syntax Tree

An AST is an abstract representation of a program in a tree format. Every programming language is defined by a context free grammar G consisting of a finite set of commands and rules. This fact is used to translate a source-code file into its abstract representation. Fortunately, an AST can also be translated back into source code.
Usually, this format is used by compilers, debuggers, and validators to check the source code for semantic meaning. This can be of great value regarding source-code generation, since it models the semantic better than plain source code.
The tree representation encodes punctuation and parentheses of plain source code implicitely by parent-children relationships of the source-code elements. A great advantage is that the AST usually exists in an IDE and does not has to be separately created.

In general, each source-code element is represented as a node in the AST. There are two types of nodes: terminal nodes and non-terminal nodes. Every node which is not a leaf node of the tree corresponds to a non-terminal of the context free grammar [Liu et al., 2016], whereas the leaf nodes correspond to terminal elements. Terminal elements of the grammar are any identifiers, strings, or number literals. Non-terminal elements are all language specific elements, for example: *ExpressionStatement*, *VariableDeclaration*, *MethodDeclaration* or *CompilationUnit*. A complete list of Java non-terminals can be found in Chapter 19 of the Java Language Specification[4]. Each non-terminal node can have multiple child nodes and can be the child of another node itself. Due to the fact that each terminal node represents an arbitrary literal, there can be infinite terminal nodes. The amount of different non-terminal nodes is limited by the number of control elements of the grammar.

In Java, the global root node is a *CompilationUnit* which represents a *.java* file. An example of a *CompilationUnit* AST can be seen in figure 3.2 which is the representation of the source code in listing 3.1.

Listing 3.1: Source-code representation of figure 3.2

```java
public class SampleClass {
    int add() {

    }
}
```

The terminal nodes of the figure are shown in green and non-terminal nodes in blue. Of course, this tree is incomplete since the body of the first *MethodDeclaration* and the definition of the second *MethodDeclaration* are ommited. It can be seen that some non-terminals such as *MethodDeclaration* are grouped together into an array. This enables an arbitary amount of children of the same type for the parent node which means that a class can have multiple methods.

Since we train a deep neural network using the ASTs of the Java files in the corpus of 3.3, we had to extract them beforehand.

3.4.1 AST Extraction

To extract the AST of any given Java file, we developed the Java application *ParseCorpus*. This tool takes the path to a Java file, processes it and returns the AST of any given root node type.
In order to create the AST, we used the library *javaparser*[5] which is a tool to generalize, analyze and process Java code. It provides an easy API to get AST nodes of given Java files and provides the functionality to serialize them.
To extract not only the *CompilationUnit* but also any other type of node, we

[4]`https://docs.oracle.com/javase/specs/jls/se9/html/jls-19.html` - accessed April 2, 2018
[5]`http://javaparser.org/` - accessed April 2, 2018

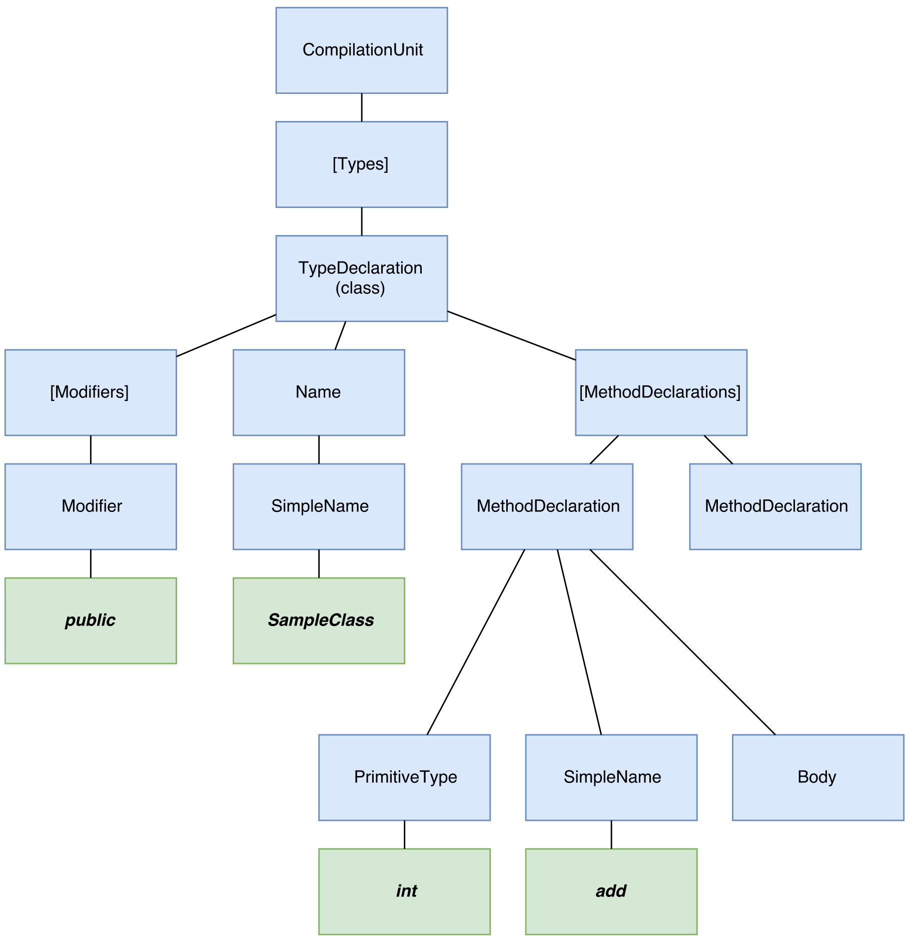


Figure 3.2: An example of an incomplete AST with a *CompilationUnit* as root node

implemented a *nodeWalker* function. It takes a *CompilationUnit*, extracts its AST, walks through the tree depth first, and visits each child node until the searched node type is found. At this point, it stops visiting any further child nodes at the current subtree, adds the subtree to a list, and proceeds to walk all remaining nodes.
With this function, it is possible to extract all different kinds of granularity. The coarsest granularity in Java is represented by a *CompilationUnit* which can contain all other grammar elements. By contrast, one of the finest granularities are *SimpleNames*. They have only an identifier attribute and are succeeded only by a literal node.

Due to the large amount of nodes types in the *CompilationUnit*, we decided to first take finer granularities to train our deep neural net. The selection

	CompilationUnit	*MethodDeclaration*	*VariableDeclaration*
total number of trees rooted at the given type	4,426	46,626	73,080
average child elements	1,382	101	17
number of different child elements	490	462	292
number of different identifiers	108,351	88,552	51,727

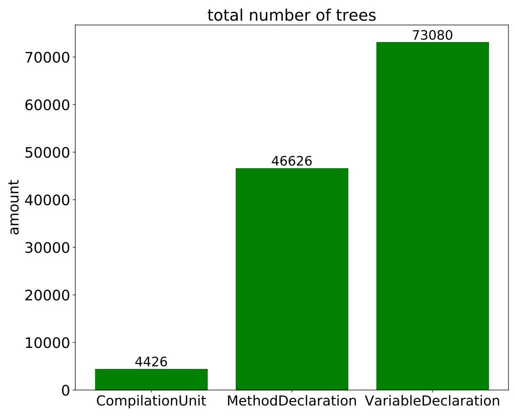
Table 3.3: AST set analysis

of the granularity is strongly connected to the knowledge ratio α of our formal model. We want to achieve a high constant α by selecting a subset of the complete problem. Finer granularities have smaller subtrees; and hence, lead to a reduced difficulty for the learning process. Networks trained on finer granularities can later be used as parts for networks which generate coarser granularities, splitting the overall task into smaller parts. Additionally, finer subtrees result in a reduced set of higher order language concepts C and less grammatical elements G.

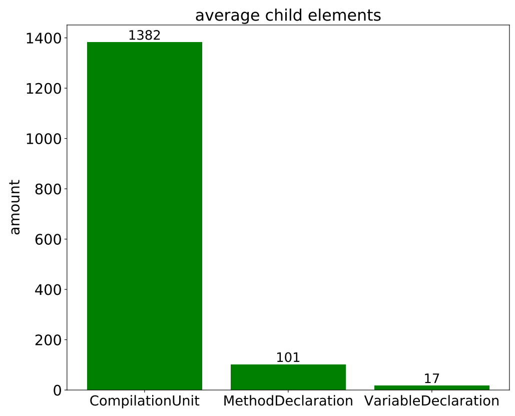
For this work, three different root node types were extracted:

- *CompilationUnit*
- *MethodDeclaration*
- *VariableDeclaration*

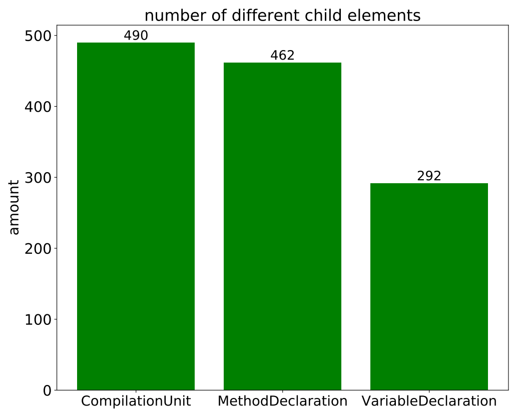
All Java files of our corpus were fed into the *ParseCorpus* tool. This resulted in three different AST sets. An analysis of those sets is presented in table 3.3 which is also plotted in figure 3.3.
Altough the set of *VariableDeclaration* has more trees in total, it becomes clear that it is less complex than the other two sets. The set provides a smaller count of different child elements and has nearly 57,000 different identifiers less than the *CompilationUnit*. A smaller number of unique identifiers may reduce the complexity for the training process and leads to a higher knowledge ratio α. Fortunately, the increased amount of different trees gives us more potential training data. The amount of the higher order language concepts $c \in C$ and



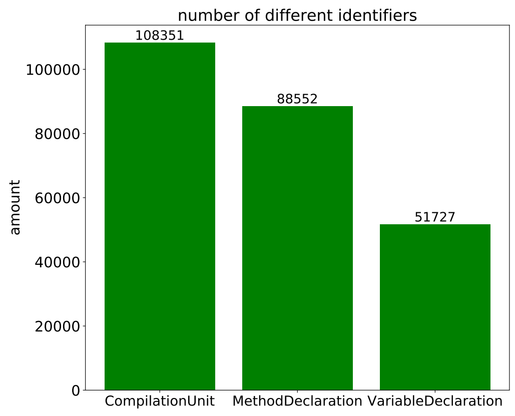
(a) Number of different trees



(b) Average child count for the set



(c) Number of different types in the set



(d) Number of different identifiers in the set

Figure 3.3: Plots of the AST set analysis

grammatical elements $g \in G$ decreases with respect to the selection of lower granularities which leads to an reduced difficulty Θ.
Hence, we focus on the *VariableDeclaration* set in this work.

The resulting AST trees of our *ParseCorpus* tool are represented as Java objects only inside Eclipse. That is, to train our network, we needed to serialize them first.

3.4.2 Serialization

The process of serialization is mapping structured abstract data into a sequential representation. Its inverse is the deserialization process.
The *javaparser* library has a mechanism to print a given AST node into a character string. Using this mechanism, the resulting string can have only one of a set of predefined formats:

- Extensible Markup Language (XML)
- JavaScript Object Notation (JSON)
- Yet Another Markup Language (YAML)
- Dot-Format
- plain source code

XML, JSON, and YAML are text document formats aiming at human and machine readability. Dot is a representation for the graph visualization tool Graphviz[6] and source code translates the AST back to its source-code representation.
Both XML and JSON introduce format specific text elements. XML encodes a tree via nested elements which can be defined in an XML-scheme, whereas JSON encodes it with use of parentheses. Figure 3.4a contains an example Java function. Parsing this function with *javaparser* results in the figure 3.4b for JSON, figure 3.4c for XML and figure 3.4d for YAML.
In this work, we decided to use the YAML format as AST serialization. Simply by observing the three different versions, it can be seen that YAML is the most compact variant of all serializations.
The main difference of YAML to XML and JSON is that it does not need additional elements to encode the tree structure. A node is represented by the string in a line. The node type is declared as the word in front of the parentheses and its corresponding AST element is inside the parentheses.
There are two other kinds of nodes: arrays and parameter. The symbol "-" indicates the start of an array node which can have multiple child nodes of the same type.
A parameter contains a node type followed by an identifier in quotation marks. Hierarchical dependencies are modeled by indentation.

To simplify the YAML notation, we encoded the indents in each line. A neural network should be able to learn the indentation but without counting the

[6]`https://www.graphviz.org/` - accessed April 2, 2018

```java
public int add() {
    return 0;
}
```

(a) Example Java function

```
<root type='MethodDeclaration'
    >
  <body type='BlockStmt'>
    <statements>
      <statement
          type='ReturnStmt'>
        <expression
            type=
              'IntegerLiteralExpr'
            value='0'>
        </expression>
      </statement>
    </statements>
  </body>
  <type type='PrimitiveType'
      type='INT'></type>
  <name type='SimpleName'
      identifier='add'></name>
</root>
```

(c) XML AST representation

```json
{
  "type":"MethodDeclaration",
  "body":{
    "type":"BlockStmt",
    "statements":[
      {
        "type":"ReturnStmt",
        "expression":{
          "type":
            "IntegerLiteralExpr",
          "value":"0"
        }
      }
    ]
  },
  "type":{
    "type":"PrimitiveType",
    "type":"INT"
  },
  "name":{
    "type":"SimpleName",
    "identifier":"add"
  }
}
```

(b) JSON AST representation

```yaml
1  root(Type=MethodDeclaration):
2    body(Type=BlockStmt):
3      statements:
4        - statement(Type=
               ReturnStmt):
5            expression(Type=
                 IntegerLiteralExpr):
6              value: "0"
7    type(Type=PrimitiveType):
8      type: "INT"
9    name(Type=SimpleName):
10     identifier: "add"
```

(d) YAML AST representation

Figure 3.4: Source-code representations of an example Java function

whitespaces. So, we defined a new element which contained the amount of leading whitespaces with an additional character: *w<integer>*. For example, line 3 of listing 3.4d results in “w4 statements:” since it contains 4 leading whitespaces. In that case, a neural network has to predict only one element and not as many whitespaces that can occur.

3.4.3 Tokenization

To train a neural network, we have to specify how its input should be presented.
As stated in chapter 2.1.2, we need to describe a language model. To create a vocabulary V of the given serialized ASTs, we have to define words x in this language.
We had to decide between a character-level and a word-level model and chose word-level since it provides the possibility of encoding more information than a simple character-level model.
There are different methods to tokenize a serialized AST file.
The first method which was used in this work was the *word_tokenize* func-

tion of the python package Natural Language Toolkit[7] (*NLTK*). It is a library which provides functions to process human language data such as classification, tagging, and tokenization.
The *word_tokenize* method is the recommended word tokenizer of *NLTK* and is based on the "Punkt-tokenizer" mechanism which tokenizes a given input string resulting in words and punctuation tokens.
For example, tokenizing the YAML string of figure 3.5 with *word_tokenize* would result in the token sequence of figure 3.6.

```
w0 root(Type=VariableDeclarationExpr):
w4 variables:
w8 - variable(Type=VariableDeclarator):
w12 initializer(Type=ArrayCreationExpr):
```

Figure 3.5: An example of a serialized AST node

$$tokens = \begin{bmatrix} \text{[w0], [root], [(], [Type=VariableDeclarationExpr], [)], [:],} \\ \text{[w4], [variables], [:],} \\ \text{[w8], [-], [variable], [(], [Type=VariableDeclarator], [)], [:],} \\ \text{[w12], [initializer], [(], [Type=ArrayCreationExpr], [)], [:]} \end{bmatrix}$$

Figure 3.6: NLTK tokenized AST of figure 3.5

Another approach is the tokenization based on regular expressions (regex). Since the YAML representation is highly structured, different elements always occur in the same formation.
After analyzing the serialized AST nodes, we were able to identify 5 different element types and counted their occurence (tab. 3.4). On that base, we created regular expressions to catch each case (see appendix A.1).
Compared to *NLTK* tokenizing, the regex tokenization needs on average 68 tokens per file less for all AST files in the *VariableDeclaration* set. For example, tokenizing the same AST node of figure 3.5 with regex tokenization results in a sequence with less tokens which can be seen in figure 3.7.
It can also be verified that there is a huge difference between the amount of non-terminals and terminals in the dataset due to the infinite possibilities for terminal elements of the context free grammar (ref. chapter 3.4).

[7]`https://www.nltk.org/` - accessed April 2, 2018

YAML element	Occurence	Example
AST elements (non-terminal)	215	*root(Type=VariableDeclarationExpr)*
Array declarators (non-terminal)	17	*catchClauses:*
Array elements (non-terminal)	76	*- parameter(Type=Parameter)*
Identifiers (terminal)	51,727	*identifier: "dragNewBox"*
Indentations	1,494	*w12*

Table 3.4: Different YAML elements on the *VariableDeclaration* set

$$tokens = \begin{bmatrix} \text{[w0], [root(Type=VariableDeclarationExpr):],} \\ \text{[w4], [variables:],} \\ \text{[w8], [- variable(Type=VariableDeclarator):],} \\ \text{[w12], [initializer(Type=ArrayCreationExpr):]} \end{bmatrix}$$

Figure 3.7: Regex tokenized AST of figure 3.5

3.4.4 Feature Extraction

A partial AST alone provides information only about itself. For example, an AST describing a variable declaration has information about the type, name, method calls, and expressions of the variable it describes. But, since we are interested in its context, we need additional information. Any additional data is called a feature for a given sub-AST and therefore increase the amount of available information γ. However, the features are only a part of γ and we call this part $\gamma_{feature}$.
At the point where the AST of a Java file is extracted and the subtrees of the root node types are selected, the tree is also parsed in reverse to find other nodes. We are interested in following features:

- all imported *swing* classes,
- global variables,
- any method of the current class,
- any method parameter of the current method, and
- previous declared variables of the same scope

In order to get all features from the *swing* library, its source code was acquired. Again, all *swing* Java files were parsed and for each file a summary of all public

methods and variables has been created.
All methods and variables corresponding to a subtree are stored in additional files with their type and name. So, for each serialized AST there is an additional file containing its features.
If a variable declaration has no further information (e.g. the first variable in a new class), then there is no feature available and $\gamma_{feature} = 0$.

3.4.5 Vocabulary Creation

In chapter 3.4.3, we declared the words of our language model. To create a vocabulary with this words, we parsed the serialized AST files and tokenized it with a Python script. We created a dictionary to store the different tokens and count its occurence in the dataset. Subsequently, we add new tokens and increment the occurence count if the token already existed in the dictionary. Additionally, the types and names of the feature files were added.
As a result, the vocabulary of the *VariableDeclaration* set has a size of 13,163 with regular expression tokenization and 53,662 with *NLTK* tokenization.
As stated in the definition of language models (section 2.1.2), not all tokens have to be stored in the vocabulary. Tokens with a low frequency in the dataset can be modeled with a *<UNK>* token. But, since it is crucial for the neural network to be able to generate all kinds of non-terminals, only the amount of terminal nodes should be reduced.
According to Zipf's law, each token in a set of documents can be ranked [Zipf, 1935]. It states that the rank of a token decreases rapidly with its frequency. This can be seen in the plot of identifier token frequencies (figure 3.8). On the left side, the frequency of the most frequent tokens are shown with rapidly decreasing frequency further to the right. The top 20 most frequent identifier tokens are also plotted and presented in figure A.1. Only 6,147 of the 51,727 identifier tokens have a frequency greater than 5 and more than 30,000 tokens have a frequency of 1 over all documents.
This is why we decided to remove all identifier tokens with a frequency less than 5 and replaced them with a single *<UNK>* token.
Additional tokens were added to the vocabulary to model the start (*<SOF>*) and the end (*<EOF>*) of a serialized AST file. Finally, the vocabulary of the *VariableDeclaration* set consisted of 6,771 tokens.

3.4.6 Input Preprocessing

Since we train a RNN on serialized AST files, we must model the format of the input data. It is usually split into separate datasets: training, validation, and testing.

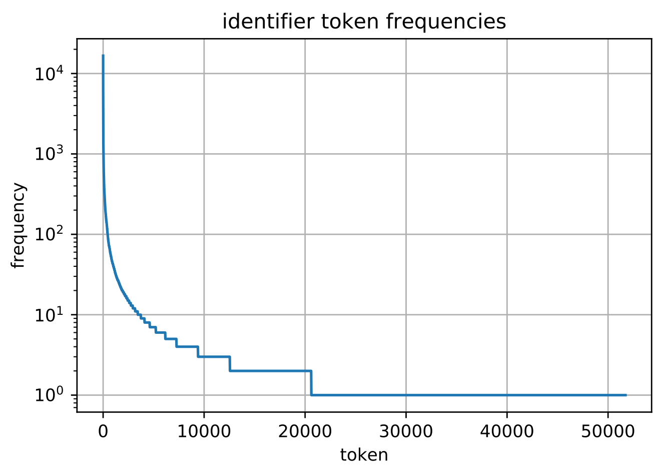


Figure 3.8: Sorted identifier frequencies of the *VariableDeclaration* set with regex tokenization

Only the training dataset is used to train the network. The validation set is used to perform checks in specific intervals in order to measure the performance of the network on generalizing its learned features with unseen data. As final result, the network state with the best performing validation is used. Additionally after the training, the network can be checked against the test dataset to validate its overall performance.
The sizes of the datasets depend on how much data is available. Less data leads to smaller datasets and a small training set leads to a greater variance on the estimated parameter which results in a bad performance of the network. But if there is enough data, it can be divided such that the training set is big enough to minimize the variance.
Choosing appropriate sizes for the sets is more or less arbitrary as long as the training set is big enough. But according to [Shahin et al.], a good splitting is: 20% of the whole dataset for validation leaving 70% of the remaining set for training, and 30% for testing.
In our work, we divided the serialized AST files according to the proposed splitting of [Shahin et al.] by randomly selecting files. Table 3.5 shows how many files, lines, and tokens are present in each dataset.

After the dataset separation, we used the a tokenization (chapter 3.4.3) on all files to create the vocabulary (chapter 3.4.5). Since a neural network does not train directly on word tokens, we implemented an index mapping which maps each token in the vocabulary to its index and backwards. If a token is not in the vocabulary, the index of the *<UNK>* token is used instead.

Dataset	File count	Line count	Token count
Training	40,925	700,903	1,402,182
Validation	14,616	249,019	498,189
Testing	17,539	302,249	604,640

Table 3.5: Splitted *VariableDeclaration* set

We used the index mapping to translate the tokens of the serialized AST files to their corresponding vocabulary indices. However, a serialized AST file consists of an arbitrary amount of word tokens. But, a recurrent neural network expects its input to be of a fixed context size c.
This is why we used a context window to extract input-target pairs. Figure 3.9 shows how the data is sliced at slicing step n and the next step $n + 1$. The context size c of a RNN controls the size of the context window (orange

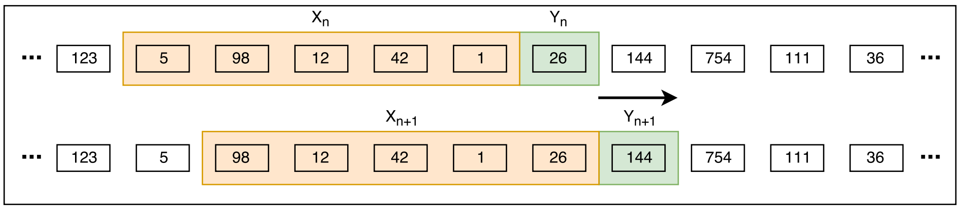


Figure 3.9: Sliding window on slicing step n and $n + 1$

window). Additionally, the output size of a RNN controls how many tokens should be predicted (green window) based on the given context. In each slicing step, the context window moves one token further.
However, it can only start with the first token in a file, enabling the first tokens x_0 to be used only as context for the first output y_0. The existence of any parts of any tokens controls the value of $\gamma_{context}$. In order to be able to also predict the first tokens, we implemented a padding. The start of a file is padded with $<SOF>$ tokens as long as the first output y_0 is the first real token of the file. Predicting the first token maps to the case when $\gamma_{context} = 0$ or $\gamma = 0$ (if also no feature vector is available).
Additionally, the end of a file is indicated with an appended $<EOF>$ token. The resulting (x,y) tuples for all serialized ASTs are stored in a file to not have to generate them for each training run.

3.5 Deep Learning Networks

To train code suggestions, we use deep neural networks with recurrent layers. There are many frameworks available which provide the functionality to model a neural network architecture, for example: Tensorflow, Torch, Pytorch, Caffe, Keras, and Theano. For this work, we decided to use Tensorflow[8] which is an open-source library developed in 2015 by the Google Brain research team.

We implemented three different network architectures to analyze different settings of our code generation difficulty Θ.
In general, these networks make use of a RNN with LSTM units and train on serialized AST files as time series data.
The training process iterates over all examples of the training set. One iteration is called an epoch. To optimally train an ANN, the training iterates multiple times over the training set which can be adjusted by the epoch size.

3.5.1 LSTM Model

Our first network consists of two LSTM layers. Since a LSTM layer has only one hidden state (with two hidden values), the usage of two or more layers provides additional capability to memorize certain structural information. Figure 3.10 shows a diagram of the network. We aim to predict the next token of a

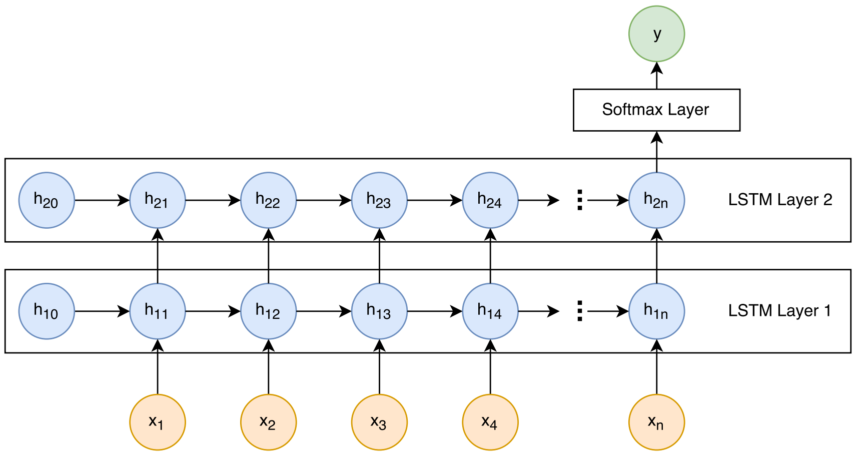


Figure 3.10: Architecture of the LSTM model

given token sequence. This is why it has a many to one architecture.
The input values of our network are x and $\hat{y}$ where x is an array of context token indices and $\hat{y}$ is a one-hot encoded vector with length of the vocabulary.

[8]`https://www.tensorflow.org/` - accessed April 2, 2018

One-hot encoded means that every element of the array is set to zero except a one at the index of the corresponding target token. The hidden states h_{10} and h_{20} are initialized with zero vectors. At the end, the last output of the second LSTM layer is processed by a softmax layer.
Softmax is an activation function which is commonly used on output layers of neural networks. It returns a vector y with a probability distribution over a set of classes by calculating:

$$\sigma(w_i) = \frac{e^{w_i}}{\sum_{j=1}^{m} e^{w_j}} \text{ for } i = 1, .., m \tag{3.1}$$

where w_i is an output class index and m is the amount of classes. In our case, the classes are the different indices of our vocabulary. The probability vector y is the final output of the network and the index with the highest value represents the next prediction.
In order to train the network, it requires an error function and an optimizer. Since we want to predict the correct class (element in our vocabulary) of the next token, we use cross entropy to calculate the error between predicted vector y and target vector $\hat{y}$. It is calculated as follows:

$$H(\hat{y}, y) = -\sum_{i=1}^{m} \hat{y}_i * log(y_i) \tag{3.2}$$

We fed the computed error to the optimizer function which updates the trainable network variables based on partial derivatives of the error. In our work, we use an Adam optimizer which adjusts learning rates for each network parameter during training. A learning rate describes how much the variables will be adjusted by the optimizer.
To validate our network, we feed the preprocessed data of size m of the validation set as input, check if the prediction is correct, accumulate the results, and calculate the mean:

$$acc = \frac{1}{m} * \sum_{i=1}^{m} \begin{cases} 1 \; if \; argmax(y_i) = argmax(\hat{y}_i) \\ 0 \; otherwise \end{cases} \tag{3.3}$$

Additionally, we introduce the beam-search method as a second validation metric. Beam-search applies a breadth first search on the top k predictions for each validation step and results in a probability tree for each $j \in k$. That means, each j is appended to the current sequence and the next probability distribution is generated, which is sampled again. To control the sampling, beam-search uses two parameters: beam-depth and beam-width. Beam-depth controls the depth of the tree and beam-width controls k for each depth.

Then, each path from a root node j to its leaf nodes results in a probability path. In order to normalize a path, we introduce weights for each level of the tree. The probability of a path is the weighted sum of all nodes along the path. The path with the highest probability is the most likely next path when using j as the next prediction.
Figure 3.11 shows an example of beam-search with beam-width 2 and beam-depth 3. It can be seen that the first prediction with the highest probability

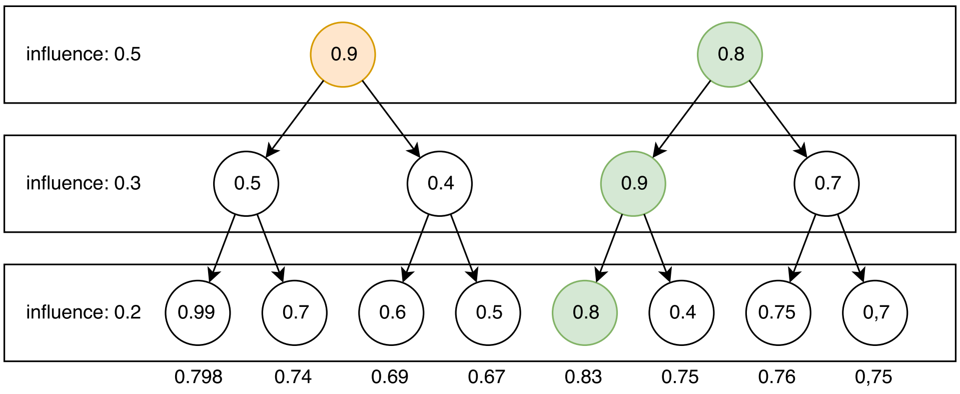


Figure 3.11: A beam search example

is not the best choice for the next token since all of its successor paths have a lower probability than the first path of the second prediction.
The intention behind this metric is that, even if a certain prediction has the highest probability to be the next token, another prediction with lower probability may lead to a better sequence.
In our work, we chose $beamdepth = 2$, $beam{-}width = \{4, 3, 2\}$, and $influence = \{0.5, 0.3, 0.2\}$.

In order to improve the training process, we use a mini-batch approach. It stacks multiple input values together to form a batch and optimizes the network based on its average error.
This model is only capable of generating AST nodes without the knowledge of the context. Only knowledge of the variable declaration itself is available, if parts of it are given. Hence, the value of γ is very low (or 0 if we start with no context information).
However, the knowledge ratio $\alpha = 1$, if we want to generate variable declarations since we train on the *VariableDeclaration* corpus.

3.5.2 Feature LSTM Model

Our second network extends the first on in that it introduces two additional modules: a feature module and a merge module. The structure of the network can be seen in figure 3.12. To enhance the generation of AST nodes, we intro-

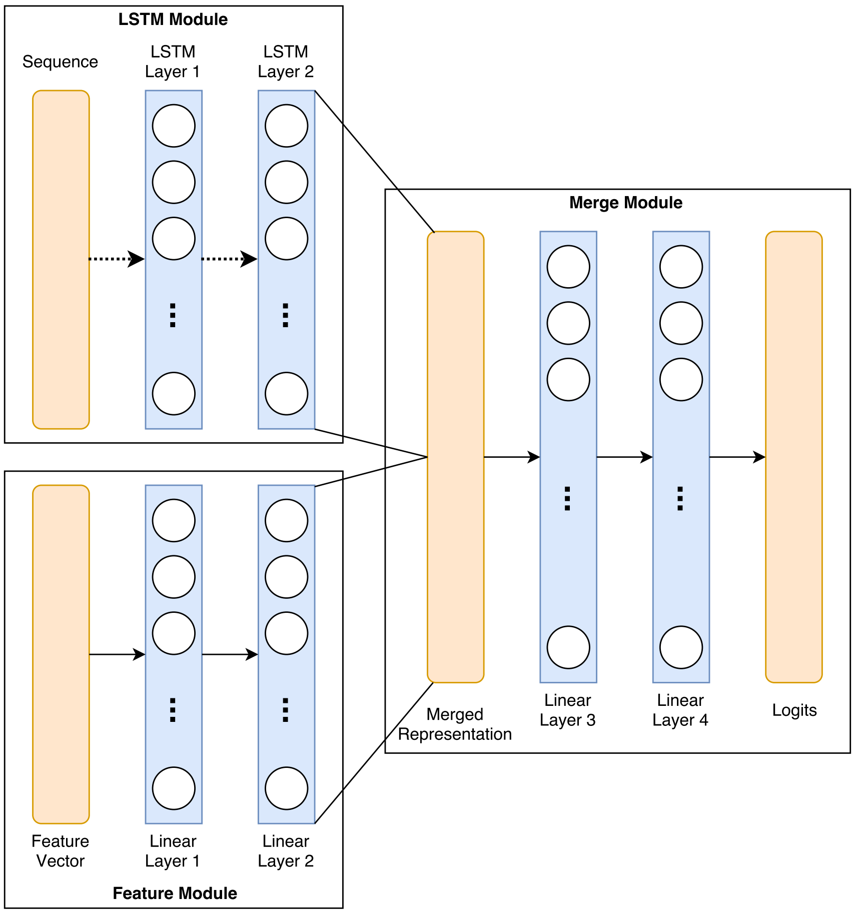


Figure 3.12: Architecture of the Feature LSTM model

duce the extracted context features. This leads to more additional information for the generation; and hence, a higher γ value. We use an encoder for the features and concatenate its output with the output of the LSTM module.
First, the features have to be encoded into vector format. To this end, we created a multi label binary encoded vector with the length of the vocabulary and set each index corresponding to a feature of the feature file to one. This vector is fed into a module consisting of two fully connected linear layers.
Addional to the softmax output of the LSTM, the hidden states are returned to use its knowledge in the following layers. The outputs of the feature module and the LSTM module are concatenated to a large vector which is used as

input for the merge module. Finally, this merged representation is processed by two fully connected linear layers and then passed to a softmax layer which maps the output to the probability distribution.
Again, the same metrics as in chapter 3.5.1 were used to calculate the error to perform optimization and to validate.

3.5.3 Feature Injection Model

Our final model is the Feature Injection model which is shown in Figure 3.13. The idea here is that we inject the encoded feature vector into the initial hid-

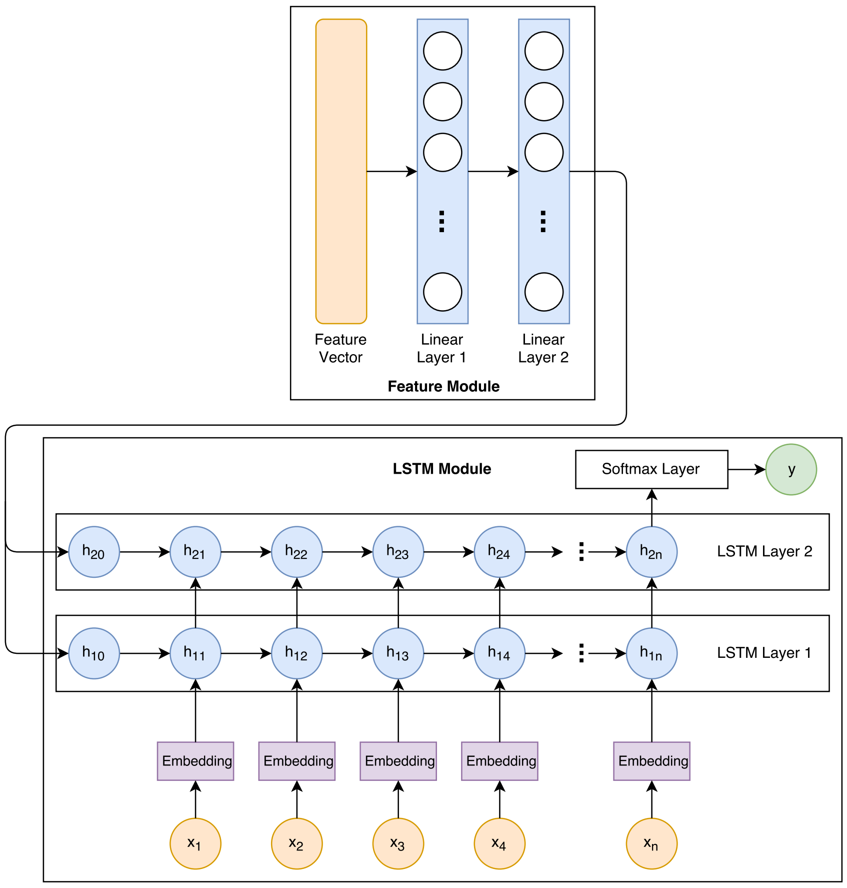


Figure 3.13: Architecture of the Feature Injection model

den states and increase γ of the LSTM module. By default, all values of the hidden cells of an LSTM are initialized with zero vectors. We assume that the existence of features imply contextual knowledge γ and, hence, feed that knowledge to the memory.
To feed the output of the feature module to the hidden states, they must be

of the same dimensions as the dimensions of the hidden states. Since LSTM cells have two hidden values (see chapter 2.1.1), only the initial memory value c_0 is overwritten and the initial activation value a_0 is left untouched.
An additional update is the application of an embedding layer for the input values of the LSTM module since it models contextual relations of the tokens.

To be able to embed the input variables, we introduce an additional preprocessing step which traines the continuous bag of words (CBOW) embeddings (see chapter 2.1.2) with window size 4 over all serialized AST files.

3.6 Generation

After the training of a network, we used its best performing state to generate new AST tokens. The generator takes a sequence and, depending on the network architecture (i.e., the setting of γ), a feature vector as input which should be in the same format as the training data (see chapter 3.4.6) and returns the most likely next token.
If the input sequence is left empty, a sequence of *<SOF>* tokens is used instead to invoke the generation of the first token. Otherwise, based on the defined sequence length n of the network, only the last n tokens of the input sequence are used. Additionally, the feature vector is built the same way as in the training phase.
Since the network generates only one token at a time, we created a loop for the generator where the next generated token is appended to the sequence until the *<EOF>* token is generated. Again, beam search (see chapter 3.5.3) was used to improve generation. As a result, a complete text sequence is returned.

In order to check if a generated AST is valid, we had to deserialize the text representation to Java source code. But due to *javaparser* lacking the functionality for deserializing serialized AST files to Java, we implemented our own deserialization tool *Backparser*. It takes a YAML file and replaces all indentation encoded tokens to whitespaces. The serialized tree then is deserialized with the *Jackson* library. *Jackson* is a parser which was designed to parse JSON files to so called plain old java objects (POJO). Fortunately, this library can also parse YAML files to java objects. A parsed file results in an POJO tree with the given hierarchy of the tree. If any given file has an invalid YAML syntax or contains an invalid tree, the parser simply rejects it.
The problem is that the resulting POJO tree cannot be used directly as AST. This is why our tool walks it from top to bottom and creates *javaparser* AST objects based on the value of the current POJO node.

As a result, we get an AST tree which can simply be translated to source code. A diagram of this workflow can be seen in figure 3.14.

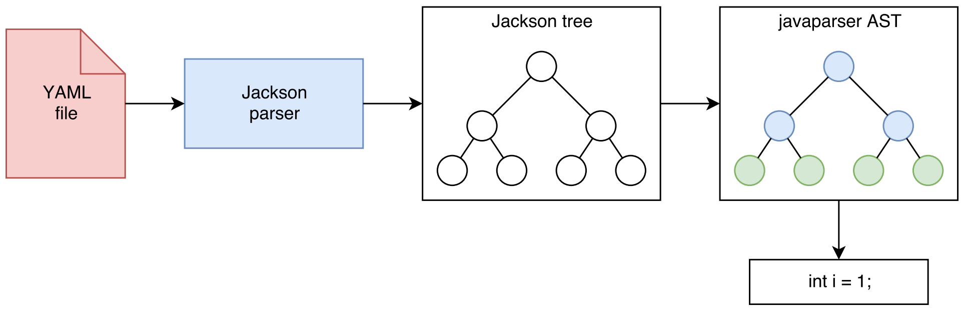


Figure 3.14: Diagram of the *Backparser* workflow

Chapter 4

Experiments

In this chapter, we describe the experiments we conducted in our work. The first experiments evaluate the text correlation of the serialized AST files. Afterwards, we evaluate the three neural networks with different settings. To this end, we first describe the experimental procedure and then results.

4.1 Text Correlation

Since there are 51,727 different identifiers and 1,494 different indentation tokens in our variable declaration set, we wanted to figure out their impact on the similarity of the serialized ASTs and; hence, the difficulty of our neural networks to learn them. It may be harder for the networks to learn a generalization if the amount of identifiers and indentations create a great variance.
Thus, we used the TF-IDF score to measure the cosine similarity (chapter 2.1.2) between the serialized variable declaration ASTs of chapter 3.4.2. However, cosine similarity gives only a distance value between two documents.
To check the files of our corpus, we would have to compare each of the 73,080 documents with all other documents which would result in 2,670,306,660 combinations. This is why we compared only a subset of 5,000 documents with 12,497,500 comparisons.
We used the python library *scikit-learn* to extract the TF-IDF scores for all files in our dataset. It provides a *TfidfVectorizer* function which takes a tokenizer definition, a minimum document frequency (*min_df*) value, and the text of all files of the corpus as input. The minimum document frequency controls the minimum amount of documents a term have to occur in. Terms with a lower document frequency are ignored. As a result, the function generates a bisymmetric similarity matrix containing all cosine similarities of each document to all other documents. At the end, we extract the sub matrix with the cosine similarities of the 5,000 files.

We created four similarity measure test setups of our variable declaration dataset:

a) with *min_df* $= \{1, 5, 10\}$ and our regex tokenizer,

b) with *min_df* $= 1$, our regex tokenizer, and replacing all identifier terms with an *<IDENTIFIER>* token,

c) with *min_df* $= 1$, our regex tokenizer, and replacing all indentation terms with an *<INDENTATION>* token, and

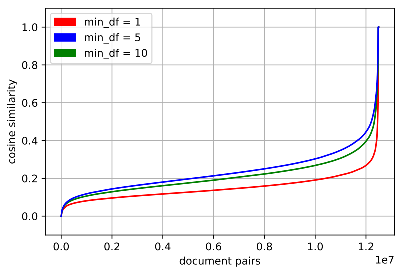
d) with *min_df* $= 1$, our regex tokenizer, and replacing both identifiers and indentations.

with identifiers	with indentation	min. document frequency	mean	median	standard deviation
✓	✓	1	0.15	0.14	0.07
✓	✓	5	0.21	0.19	0.10
✓	✓	10	0.22	0.22	0.11
	✓	1	0.62	0.62	0.18
✓		1	0.41	0.41	0.09
		1	0.86	0.87	0.09

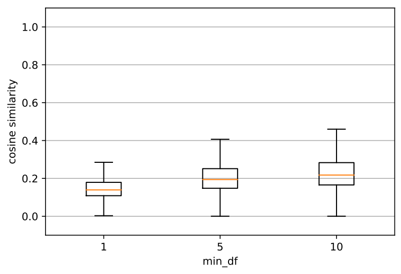
Table 4.1: Cosine similarities of 5000 AST files

The results of the similarity measures of all tests are shown in table 4.1 and plotted in figure 4.1.
It becomes apparent that the AST files mostly share 15% to 23% of their content based on the minimum document frequencies. We assume that a large portion of the differences between the documents follow the large variance of the identifier tokens. When we remove tokens with a low document frequency (which are mostly rare identifiers in our dataset), the similarity between the documents increases.
Also, we can see that the introduction of indentation tokens reduce the overall similarity about 25%. As expected, the influence of identifiers is very strong with an increased similarity of 47%.

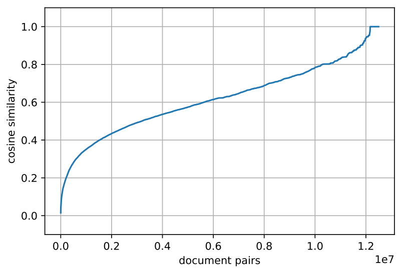
Following the results of this analysis, we can assume that a learn based approach is appropriate to learn ASTs, but the task becomes more difficult if a neural network has to learn many unique tokens. This is why we reduced our



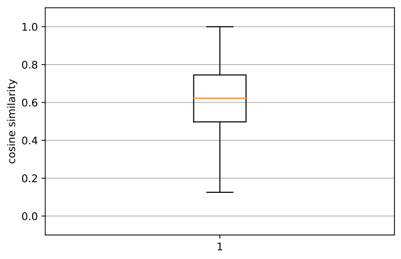
(a) Document pairs with different minimum document frequencies



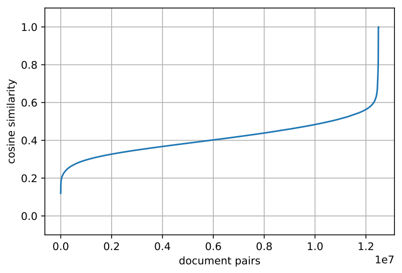
(b) Distributions of different minimum document frequencies



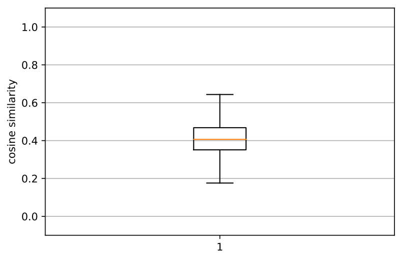
(c) Document pairs with no identifiers



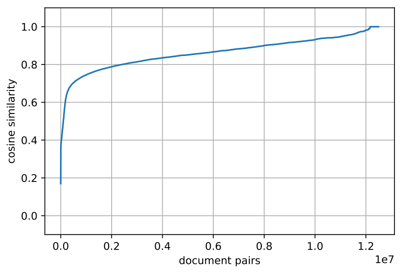
(d) Distribution with no identifiers



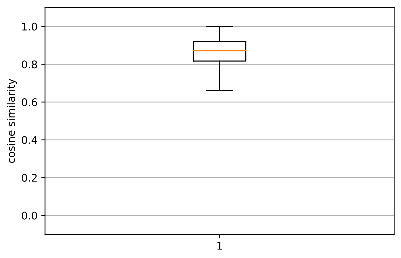
(e) Document pairs with no indentation



(f) Distribution with no indentation



(g) Document pairs with no identifiers and no indentation



(h) Distribution with no identifiers and no indentation

Figure 4.1: Diagrams of cosine similarities

vocabulary by removing identifiers with a low document frequency and indentation tokens with a high indentation value. However, this leads to a reduced γ because we remove knowledge from the dataset.

4.2 Model Evaluations

The ability of a network to predict AST nodes is the main criteria for the evaluation of the different models. In this work, we compare each model by using two metrics:

1. Accuracy on predicting the next token for a given sequence
2. Accuracy on generating a valid variable declaration AST

To evaluate the quality of our models, we ask the following research questions:

Q_1 Does γ have an effect on the quality of the generation?

Q_2 Is one of the tokenization schemes superior to the other?

Q_3 Can a model generate a valid, complete variable declaration AST node, which is usable in an IDE as code suggestion?

4.2.1 Experimental Setup

We used two different machines to train our networks: a workstation and a deep learning server. Table 4.2 shows the specifications of both machines. The

	Machine 1	Machine 2
CPU	Intel Core i7-7700K @ 4,2GHz	32x Intel Xeon E5-2683 @ 2,1GHz
GPU	Nvidia Geforce GTX1080Ti @ 12GB RAM	8x Nvidia Geforce GTX1080 @ 8GB RAM
RAM	32GB	1,5TB

Table 4.2: Hardware used for training

server provides multiple GPUs, which enabled us to train multiple network configurations in parallel. All networks were trained with a mini-batch-size of 128.
To compare our networks with each other, we created a setup for the experiments. The same task d was used for all models to generate *VariableDeclaration*s. We want to compare the network performance on the different tokenization schemes *NLTK* and regex for all models which were described in

chapter 3.4.3. This is why we created a vocabulary for each scheme. Additionally, since we wanted to know if a larger $\gamma_{context}$ improves the generation (Q_1), we preprocessed the corpus with both vocabularies and three different token context sizes $n \in \{10, 20, 30\}$. This resulted in 6 different input sets: nltk_10, nltk_20, nltk_30, regex_10, regex_20, and regex_30.
To let each network train sufficiently long, we chose an epoch size of maximal 200 epochs for each training. But, training is stopped earlier if there is no change in the validation accuracy or the model starts to overfit.
This is realized by using a patience count and patience threshold. The training is stopped if the change in the validation accuracy does not reach the patience threshold for a defined amount of validations. Additionally, the trainings were stopped by hand if they showed signs of overfitting.
We set the interval for validation to 1,000 steps. That means, a validation on the complete validation set is processed every 1,000 mini-batches. At each validation run, the overall accuracy of the validation set is checked and the network state is stored if the accuracy is better than the highest value seen so far.
Also, we test the accuracy of the validation set with beam search. But since the beam validation is very slow, we apply it only every 5 validations on a subset of the validation set.

To check the ability of the model to generate valid AST nodes, we created an additional dataset. It consisted of 100 partial ASTs and their feature files of the testing set and is again divided into subsets. To vary $\gamma_{context}$ for each subset, we removed content from the files. The text lines of each serialized AST file are cut off after a defined percentage according to a cutting scheme. This resulted in 10 subsets with cutting percentages ranging from 100% to 10% in 10 steps. That means, when we cut 100% of the original files, our networks get 0% contextual pre knowledge ($\gamma_{context} = 0$).
To provide the feature networks with feature vectors ($\gamma_{feature}$), we copied the feature files from the original dataset and fed them to the networks. Each trained model has to complete the given subset files. We compare the resulting AST files with its original counterpart by using cosine similarity. To prevent the generator getting stuck in a deadlock during AST generation, we set a limit for the maximal amount of tokens. If this limit is reached, the generation for the current AST is stopped. Due to the different vocabulary sizes, we set the limit for regex to 200 and for *NLTK* to 500.
The results are then passed to our *Backparser* which tries to translate them to *javaparser* AST objects. If this is not possible, we count the failed translation attempts. Additionally, we use the "print to source code" function of *javaparser* to extract the resulting source code of the generated and the original AST files.

We use the Levenshtein distance to check how accurate the generated source code is. The Levenshtein distance takes two character strings and measures how many characters have to be changed to transform the first string into the second string.

4.2.2 Results

All three models were trained with all 6 input sets. Every model was trained at least 24h before it started to overfit or no change in the accuracy was detected. But due to a larger size of the *NLTK* vocabulary, the training of those datasets took longer than the training with the regex datasets.
The results of all trainings can be seen in table 4.3 where the highest accuracy values of each model and tokenization scheme are colored in green. A high training accuracy means that a model trained well on its training data and an additional high validation and beam accuracy means that is was able to generalize the learned data to deal with unseen cases. It makes apparent that all models trained well enough to achieve a high validation and beam accuracy. However, it seemed that the *NLTK* training runs performed better compared to their regex counterparts.
After the training of all models, we used them to complete the cut AST files. The original and completed AST files are processed by our TF-IDF score and checked on cosine similarity. All similarity results can be seen in figure 4.4. The higher the similarity value, the more similar are the original - generated pairs of each cut set. However, a file which was cut off after a certain percentage of lines should at least reach a value value close to its cutting percentage when compared to the original, which was not always the case. All tests, which did not achieve at least their cutting percentage, are shown in red.
Finally, we processed all generated ASTs with our *Backparser*. The results of the failed attempts and mean Levenshtein distances for all experiments can be seen in figure 4.5. All fails are colored in red and the smallest Levenshtein distance for each model is colored in green. A fail count of 10 means that all completed ASTs of the cut subset could not be parsed to source code. Furthermore, a small Levenshtein distance means that the generated source code has to be changed only slightly to transform it to its original counterpart.
This can also be seen in table 4.5 where the regex version of the Feature Injection model had a smaller Levenshtein distance for lower $\gamma_{context}$. However, all networks seemed to generate more tokens than necessary at $\gamma_{context} = 90\%$ which resulted in cosine similarity values below 90.
Figure 4.2 shows the result of the Feature Injection model with regex tokenization on context size 30 and 50% $\gamma_{context}$. It was able to predict all necessary identifiers at the correct positions, resulting in the source code in figure 4.2a.

Model	**tokenization**	**Context size** ($\gamma_{context}$)	**max. training acc.**	**max validation acc.**	**max beam acc.**
			on mini-batch	on valid. set	on valid. subset
LSTM $\gamma_{feature} = 0$	regex	10	96.88%	89.18%	89.00%
		20	97.66%	89.53%	89.53%
		30	97.66%	90.53%	89.80%
	NLTK	10	98.75%	92.85%	92.47%
		20	98.72%	93.44%	92.60%
		30	98.80%	93.80%	93.80%
Feature LSTM $\gamma_{feature} = 1$	regex	10	87.50%	80.96%	68.90%
		20	85.94%	79.14%	75.98%
		30	89.06%	80.80%	73.85%
	NLTK	10	94.43%	87.07%	87.07%
		20	91.41%	84.21%	83.91%
		30	96.88%	88.00%	87.90%
Feature Injection $\gamma_{feature} = 1$	regex	10	97.66%	88.05%	88.05%
		20	97.66%	89.73%	89.18%
		30	98.44%	89.75%	89.75%
	NLTK	10	98.44%	92.83%	92.33%
		20	98.44%	93.97%	93.67%
		30	99.22%	94.41%	93.61%

Table 4.3: Training results

Code elements colored in violet were generated by the network. Figure 4.3 shows examples of all three models with regex tokenization and a context size of 30.
We present other examples of generated source code in Appendix D.

4.2.3 Discussion

We assume that the *NLTK* tokenization produces smaller tokens (e.g.: "(", ")",":", or "-") which occur very often in the dataset. The networks learn this fact and generate a correct prediction for those cases which increases the accuracy. Our regex tokenization uses the fact that elements of the Java grammar are inside of parentheses or identifier are marked with quotation marks and, hence, does not need to learn for example the opening and closing of parentheses.

```
\\generated on 50% context
JWebBrowser webBrowser = htmlEditor.getWebBrowser()
\\original
JWebBrowser webBrowser = htmlEditor.getWebBrowser()
```

(a) Source code of the completed cut file

```
w0 root(Type=VariableDeclarationExpr):
w4 variables:
w8 - variable(Type=VariableDeclarator):
w12 initializer(Type=MethodCallExpr):
w16 name(Type=SimpleName):
w20 identifier: "getWebBrowser"
w16 scope(Type=NameExpr):
w20 name(Type=SimpleName):
w24 identifier: "htmlEditor"
w12 name(Type=SimpleName):
w16 identifier: "webBrowser"
w12 type(Type=ClassOrInterfaceType):
w16 name(Type=SimpleName):
w20 identifier: "JWebBrowser"
```

(b) Full AST

```
w0 root(Type=VariableDeclarationExpr):
w4 variables:
w8 - variable(Type=VariableDeclarator):
w12 initializer(Type=MethodCallExpr):
w16 name(Type=SimpleName):
w20 identifier: "getWebBrowser"
w16 scope(Type=NameExpr):
```

(c) Cut AST at 50%

Figure 4.2: Example of Feature Injection generation with regex tokenization, context size 30 and 50% pre knowledge

```
\\LSTM context 30 regex with 0% pre knowledge
UNK UNK = UNK.UNK()

\\Feature LSTM context 30 regex with 0% pre knowledge
UNK UNK = UNK.UNK(UNK, UNK)

\\Feature Injection context 30 regex with 0% pre knowledge
TabContentPaneBorderKind kind = SubstanceCoreUtilities.UNK(this.tabPane)
```

Figure 4.3: Generated example source code with 0% pre knowledge

To compare our three models, we check the ability to generate source code with different $\gamma_{context}$. When we take a look at the mean pair similarities and the *Backparser* results, we can see that the Feature LSTM network with context size 20 produced bad examples. This may be due to an early stop of the training process because the training and validation accuracies are worse compared to a smaller context size of the same model which interrupts the overall improvement trend. The experiment results of this network had a very bad performance which led to the generation of invalid ASTs when no AST knowledge was available and mostly invalid ASTs when there was more than 10% of $\gamma_{context}$ available.
Table 4.4 shows that networks trained on regex tokenization achieved a better

result for $\gamma_{context} < 60\%$ compared to their *NLTK* counterpart.
Figure 4.3 shows that both LSTM and Feature LSTM do not generate useful source code with no contextual information. Only the Feature Injection model was able to generate mostly useful source code (except an UNK token for a method call) when γ is low. Additionally, it was able to complete the given cut AST with $\gamma_{context} = 50\%$ in figure 4.2. Hence, we assume that our Feature Injection model outperforms the other two networks.
All other trainings achieved a better validation and beam accuracy with increasing context size. Only the Feature LSTM showed no increase of accuracy. However, the accuracy of the LSTM model and the Feature model increased by at least 1% for both *NLTK* and regex when using a bigger context size.
It can be seen that the maximum beam accuracy for all nets is only slightly worse than the validation accuracy (except for the Feature LSTM model with regex tokenization). This is why we assume that our networks are able to predict the next token that starts the correct sequence.

There is no significant improvement in the cosine similarities between the LSTM model and the other two models. We assume that our metric may not be the best choice to compare two trees.

Table 4.5 shows that the models trained with *NLTK* tokenization generate often invalid AST nodes which cannot be parsed back. The regex networks seem to generate more valid nodes. This can be due to the greater vocabulary of the *NLTK* tokenization which may lead to a higher probability to generate a wrong token at a critical position in the AST.

We used the proposed formal model to describe our approach. To reduce complexity, we set the task difficulty σ on generating code suggestions for Java IDEs. Additionally, we defined a high knowledge ratio α by using only variable declarations. In our experiment, we checked the influence of γ on the quality of generating correct source code. We can assume that the variation of $\gamma_{context}$ and $\gamma_{feature}$ changes the quality. However, we tested only the absence and availability of $\gamma_{feature}$ in our experiments.
We can conclude that our formal model is a good way to define code generation frameworks and can easily be extended to cover other difficulties.
Nevertheless, it has to be defined further to score approaches which would provide a way to compare them on their performance to solve a task. To achieve this, all variables of Θ have to be declared explicitly.

$\mathbf{Q_1}$ The larger $\gamma_{context}$, the better the generation and a more effective usage of pre knowledge. Although, the Feature LSTM model uses $\gamma_{feature}$, it produces no better results than the pure LSTM model. However, this observation holds only for larger γ. We observed that the Feature Injection model which also uses $\gamma_{feature}$ was able to generate better results than the LSTM model especially when γ is low. Hence, we can answer Q_1 positively.

$\mathbf{Q_2}$ The regex models seemed to generate more valid ASTs but resulted in a higher Levenshtein distance than their *NLTK* counterparts. Nevertheless, the *NLTK* models resulted in more invalid ASTs. The answer to this question depends on what we want to achieve. If we want to generate valid ASTs with higher discrepancy, then regex is better. Otherwise, we can use *NLTK*. This is why we leave the answer to this question open for further investigation.

$\mathbf{Q_3}$ All models with regex tokenization were able to produce mostly valid ASTs. Additionally, the Feature model was also able to generate useful variable declarations even with no $\gamma_{context}$ available. Hence, we can answer Q_3 positively.

Model	**tokeniz.**	**Context size** ($\gamma_{context}$)	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
LSTM $\gamma_{feature} = 0$	regex	10	35.42	31.98	42.85	39.07	50.89	58.40	65.18	70.08	82.89	84.13
		20	35.89	37.98	43.53	46.12	52.23	58.64	61.11	68.79	87.57	84.19
		30	35.35	36.44	42.92	44.88	57.39	62.47	70.14	67.69	82.78	84.25
	NLTK	10	21.74	24.83	30.02	36.83	43.19	50.52	65.13	77.06	88.38	85.57
		20	25.65	23.38	32.13	32.15	43.28	46.76	70.34	77.79	83.94	87.28
		30	27.53	26.46	35.28	30.05	30.74	47.97	58.40	77.78	83.40	87.29
Feature LSTM $\gamma_{feature} = 1$	regex	10	17.87	23.87	30.86	32.04	44.37	56.17	58.16	66.50	86.42	83.15
		20	33.57	36.84	46.78	46.70	52.18	59.16	63.69	66.54	88.69	83.04
		30	33.71	36.89	45.61	42.41	48.68	59.48	64.08	67.63	84.94	83.69
	NLTK	10	15.52	11.59	25.21	27.33	36.61	45.56	66.31	74.41	90.12	84.73
		20	5.28	6.63	15.20	9.97	12.80	21.50	53.62	69.18	77.90	77.62
		30	10.62	17.66	15.87	13.34	19.55	41.49	57.97	74.69	73.56	86.11
Feature Injection $\gamma_{feature} = 1$	regex	10	34.76	34.26	44.54	39.67	46.34	59.83	59.81	70.38	86.90	84.05
		20	33.51	35.52	43.07	42.35	53.73	62.50	70.90	67.87	87.01	84.51
		30	28.99	27.65	41.84	39.05	49.52	63.59	69.77	68.80	88.78	84.56
	NLTK	10	21.45	24.46	31.54	36.70	39.21	45.41	64.03	77.67	85.75	85.47
		20	21.52	26.12	31.55	33.48	37.14	45.78	67.55	77.12	88.32	87.00
		30	22.28	27.45	26.88	31.35	44.08	46.68	67.22	78.48	87.96	87.46

Table 4.4: Mean similarities for original - generated pairs on the cut sets

Model	**tokeniz.**	**context size**	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
LSTM $\gamma_{feature} = 0$	Regex	10	0/35	0/37	1/40	1/44	0/67	0/22	0/17	0/16	1/23	0/11
		20	0/37	0/39	0/39	0/34	0/66	0/23	0/18	0/16	1/22	0/11
		30	0/37	0/38	0/39	0/35	0/30	0/20	0/15	0/17	1/22	0/11
	NLTK	10	0/35	0/35	0/36	0/33	0/33	0/33	0/21	1/11	2/4	2/16
		20	0/34	0/34	1/37	1/43	0/31	0/22	1/10	2/4	3/14	3/1
		30	0/34	0/34	0/36	1/38	1/57	1/24	1/32	2/4	1/12	3/1
Feature LSTM $\gamma_{feature} = 1$	Regex	10	0/170	0/156	1/145	2/106	1/77	0/40	0/53	0/17	2/30	0/13
		20	0/40	0/38	1/41	0/46	1/35	0/24	0/20	0/17	1/24	0/12
		30	0/38	0/40	1/42	0/46	1/37	0/24	0/21	0/16	1/38	0/12
	NLTK	10	0/38	0/39	1/40	2/45	0/34	0/27	1/11	2/4	1/11	3/3
		20	10/0	9/41	7/43	9/48	8/42	5/32	3/16	2/5	2/17	4/3
		30	9/41	2/37	1/54	1/62	4/55	0/23	2/17	2/5	2/21	3/2
Feature Injection $\gamma_{feature} = 1$	Regex	10	0/35	0/37	1/41	0/33	0/83	0/24	0/34	0/15	0/18	0/12
		20	0/36	0/37	0/38	0/72	0/62	0/20	0/15	0/16	1/22	0/11
		30	0/48	0/46	0/38	1/35	0/31	0/19	0/13	0/16	1/21	0/11
	NLTK	10	0/35	0/35	1/39	0/33	1/42	1/26	1/32	2/4	1/11	3/2
		20	0/49	0/42	1/47	1/44	1/34	0/20	1/12	2/4	2/17	3/1
		30	0/49	0/42	1/49	1/45	2/58	2/21	1/14	2/3	2/15	3/1

Table 4.5: Fail count / mean Levenshtein distance of the subsets using the *Backparser* on the cut sets

Chapter 5

Conclusion

To conclude our work, we review the proposed models in this chapter. Finally, we provide an outlook for future work on generating source code with neural networks.

5.1 Remarks

In this work, we proposed a framework to describe the factors that influence the accuracy of source-code generation and used it to build and evaluate different settings which was described in chapter 3. Based on the proposed framework, we specified our application area. Afterwards, we built a tool to create a corpus of variable declarations of Java-based ASTs for projects with *swing* library usage.
To learn neural networks, we serialized the ASTs and defined a language model on the serialized format. For that, we introduced two tokenization schemes to process the AST files in order to define the input of our networks.
Afterwards, we proposed three neural network models with different architectures that represent different settings of our formal framework. The first network aimed at generating AST nodes with no contextual information and the other two models aimed at producing AST nodes based on a given code context.

To evaluate our framework, we conducted two experiments in chapter 4.2. The first experiment checked the serialized AST files on similarity. We found that identifiers have a great impact on the similarity of the corpus and, hence, on the training complexity of our models. This is why a learn based approach is an appropriate choice. Additionally, we conclude that the introduction of new indentation tokens lead to additional noise in our dataset.

Our main findings are that an increased context size improves the results and our our Feature Injection model was able to generate useful variable declarations.

5.2 Outlook and Future Work

During the process of this thesis, we left several aspects unanswered which have to be studied in future work.
The results of this work have to be combined with an Eclipse plugin which uses the context of the current project and generates appropriate code completion proposals. A code completion plugin enables to conduct a user study in a real-world scenario.
In addition, other approaches have to check other language models by introducing different serialization schemes. We assume that other AST serialization schemes may result in a better performance than the YAML format.
Other AST representations without serialization have to be tested.

Clearly, the proposed models can be improved further or other neural network models can be introduced. The usage of a sequence to sequence model may further improve the generation of serialized AST nodes, or attention models can improve the generation of identifiers. Additionally, a Generative Adversary Network (GAN) model may lead to an enhanced network performance.
Furthermore, other parameters of the formal framework have to be tested. For example, another encoding for contextual features could lead to better results.
To get a better evaluation of generated ASTs, another metric has to be introduced which deals with trees instead of text documents. A useful approach would be the application of the Tree Edit Distance (TED) .

Furthermore, new approaches have to increase the granularity level to generate language constructs of higher order than variable declarations (e.g., method declarations). The results of this work can also be used in a module which generates higher granularity constructs to split the overall complexity of them into smaller parts.
Additionally, the approach of this work could be applied to other programming languages such as C++ or C#. However, untyped languages like Python or JavaScript may be challenging because the types of variables are defined by their context.

Bibliography

A. Peter. *Analyse und Vergleich von verschiedenen Vorverarbeitungsschritten und Architekturen von künstlichen neuronalen Netzen zur Codegenerierung.* Bachelor thesis, Bauhaus-Universität Weimar, 2017. 3.3

C. Alexandru. Guided Code Synthesis Using Deep Neural Networks. In *Proceedings of the International Symposium on the Foundations of Software Engineering FSE*, pages 1068–1070. ACM, 2016. 2.2.1

D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv preprint arXiv:1409.0473*, 2014. 2.2.1

M. Balog, A. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. DeepCoder: Learning to Write Programs. *arXiv:1611.01989*, 2016. 1, 2.2.1

A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel. Learning Python Code Suggestion with a Sparse Pointer Network. *arXiv:1611.08307*, 2016. 1, 2.2.1

C. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006. 2.1.1

N. Bui and J. Jiang. Hierarchical Learning of Cross-Language Mappings through Distributed Vector Representations for Code. *arXiv preprint arXiv:1803.04715*, 2018. 1, 2.2.1

B. Campbell and C. Treude. NLP2Code: Code Snippet Content Assist via Natural Language Tasks. In *Proceedings of the International Conference on Software Maintenance and Evolution ICSME*, pages 628–632. IEEE, 2017. 1, 2.2.3, 3.1

M. Collins. Language Modeling, 2018. URL `http://www.cs.columbia.edu/~mcollins/lm-spring2013.pdf`. accessed 02.04.2018. 2.1.2

P. Devanbu. Evolution vs. Intelligent Design in Program Patching. Technical report, UC Davis: College of Engineering, 2013. URL `https://escholarship.org/uc/item/3z8926ks`. accessed 02.04.2018. 1

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, pages 1735–1780. MIT Press, 1997. 2.1.1

C. Liu, X. Wang, R. Shin, J. Gonzalez, and D. Song. Neural Code Completion. April 2016. 1, 2.2.1, 3.4

T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. *Proceedings of the International Conference on Learning Representations ICLR Workshop Papers*, 2013. 2.1.2

M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, 1969. 2.1.1

T. Mitchell. *Machine Learning.* McGraw-Hill Book Co, 1997. 2.1.1, 2.1.1

L. Mou, R. Men, G. Li, L. Zhang, and Z. Jin. On End-to-End Program Generation from User Intention by Deep Neural Networks. *arXiv:1510.07211*, 2015. 2.2.1

S. Proksch, V. Bauer, and G. Murphy. How to Build a Recommendation System for Software Engineering. In *Software Engineering*, Lecture Notes in Computer Science, pages 1–42. Springer, Cham, 2013. 2.2.3

V. Raychev, P. Bielik, M. Vechev, and A. Krause. Learning Programs from Noisy Data. In *Proceedings of the Symposium on Principles of Programming Languages POPL*, pages 761–774. ACM, 2016. 2.2.1

F. Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. *Psychological Review*, pages 65–386, 1958. 2.1.1

D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, pages 533–536. Nature Publishing Group, 1986. 2.1.1

A. Sethi, A. Sankaran, N. Panwar, S. Khare, and S. Mani. DLPaper2Code: Auto-generation of Code from Deep Learning Research Papers. *arXiv:1711.03543*, 2017. 1, 2.2.1

M. Shahin, H. Maier, and M. Jaksa. Data Division for Developing Neural Networks Applied to Geotechnical Engineering. *Journal of Computing in Civil Engineering*, pages 105–114. American Society of Civil Engineers, 2004. 3.4.6

I. Sutskever, O. Vinyals, and Q. Le. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014. 2.2.1

W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the International Conference on Software Engineering ICSE*, pages 364–374. IEEE Computer Society, 2009. 1, 2.2.2

G. Zipf. *The Psycho-Biology of Language; an Introduction to Dynamic Philology.* Houghton Mifflin company, Boston, 1935. 3.4.5

Appendix A

Tokenization

case	**regular expression**
Array elements	`\-\s\w+\(\w+\=\w+\)\:`
Indentations	`w[0-9]+`
AST elements	`\w+\(\w+\=\w+\)\:`
Identifier	`\w+\:\s\"\w+\s\w+\"`
Array declarators	`\w+\:`
Identifier	`\w+\:\s\".*\"`

Table A.1: Regular expressions for tokenization

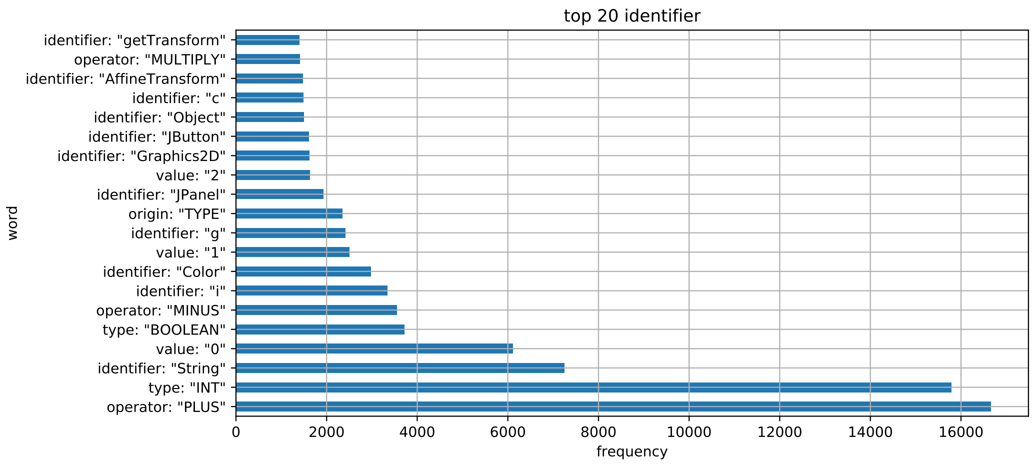
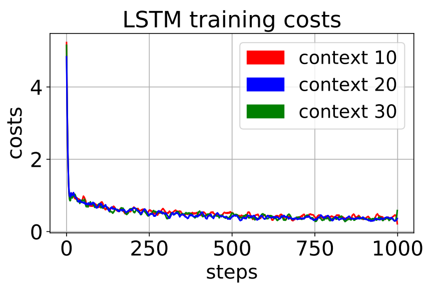


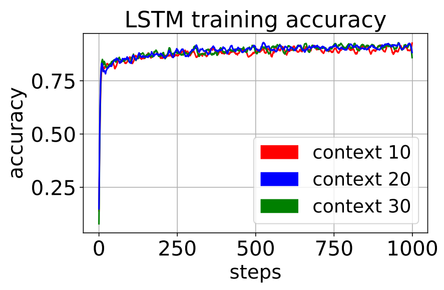
Figure A.1: Top 20 frequent identifier tokens with regex tokenization on the *VariableDeclaration* set

Appendix B

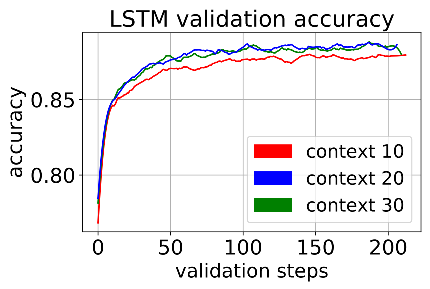
Training Runs with Regex Tokenization



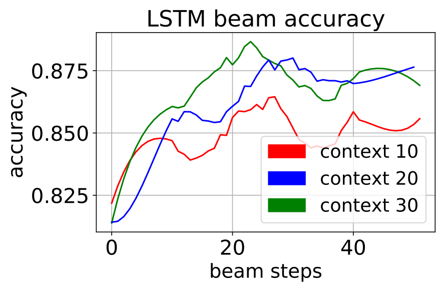
(a) LSTM training costs



(b) LSTM training accuracy

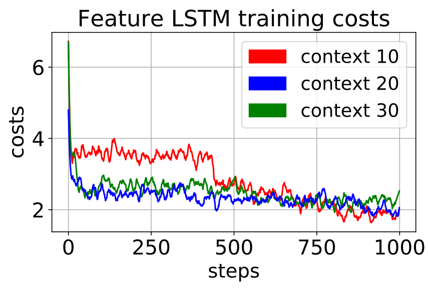


(c) LSTM validation accuracy

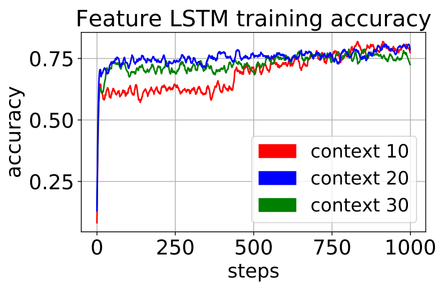


(d) LSTM beam accuracy

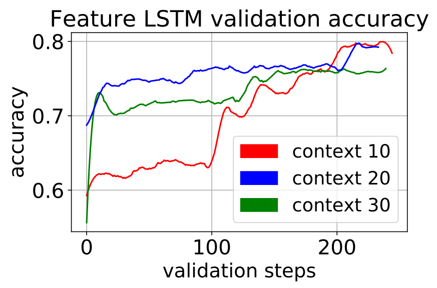
Figure B.1: Training runs of the LSTM model



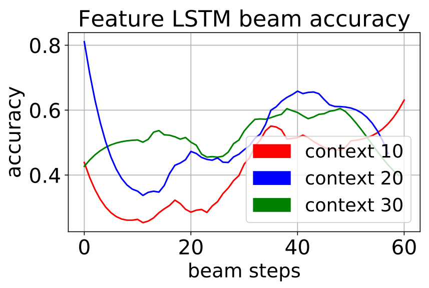
(a) Feature LSTM training costs



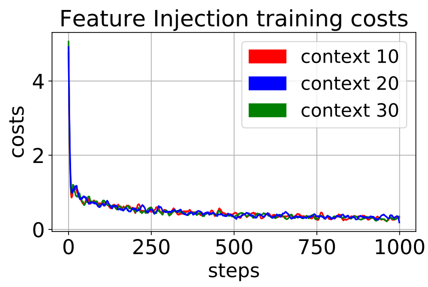
(b) Feature LSTM training accuracy



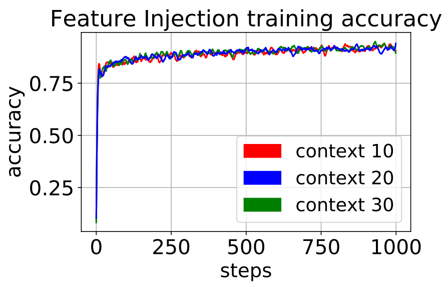
(c) Feature LSTM validation accuracy



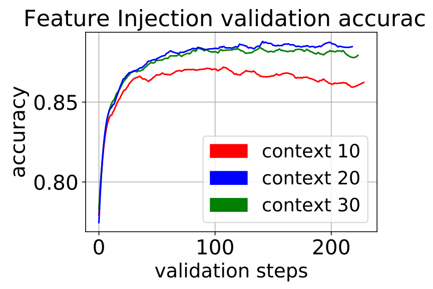
(d) Feature LSTM beam accuracy



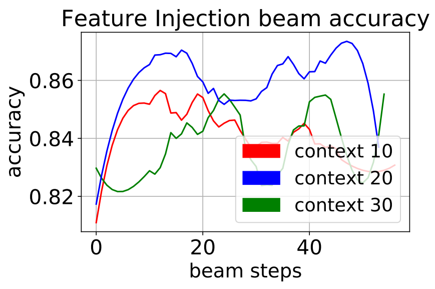
(e) Feature Injection training costs



(f) Feature Injection training accuracy



(g) Feature Injection validation accuracy

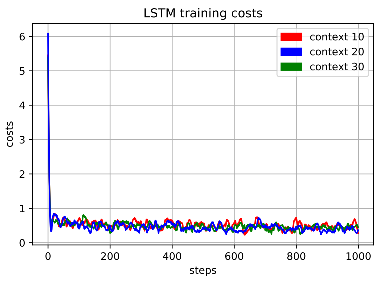


(h) Feature Injection beam accuracy

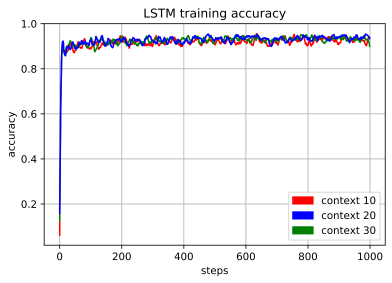
Figure B.2: Training runs of the Feature LSTM model and the Feature Injection model

Appendix C

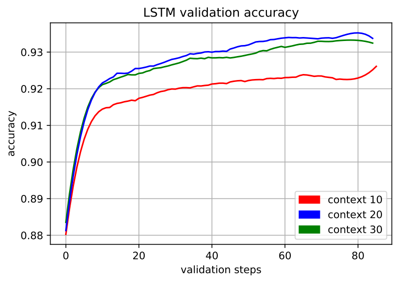
Training Runs with NLTK Tokenization



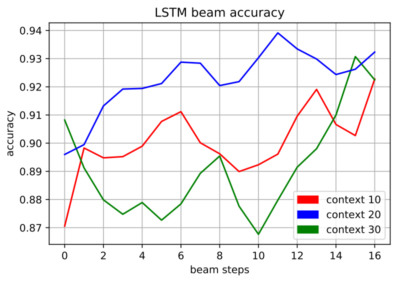
(a) LSTM training costs



(b) LSTM training accuracy

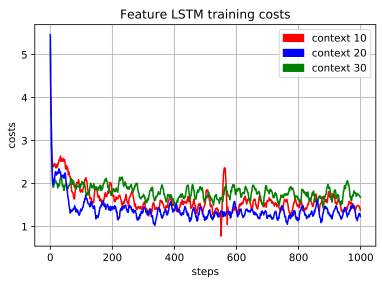


(c) LSTM validation accuracy

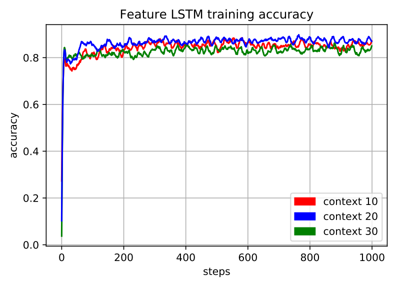


(d) LSTM beam accuracy

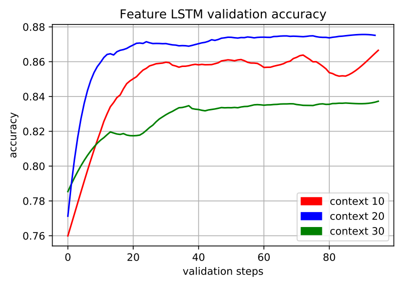
Figure C.1: Training runs of the LSTM model



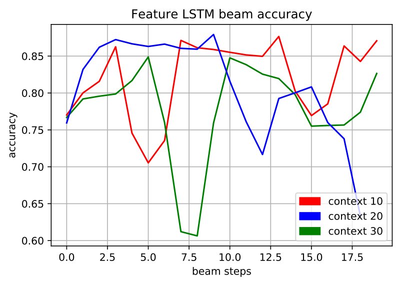
(a) Feature LSTM training costs



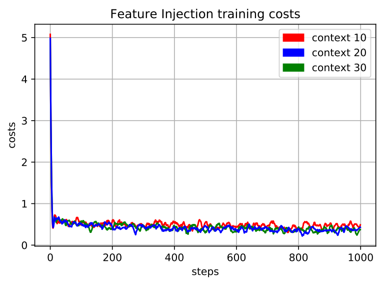
(b) Feature LSTM training accuracy



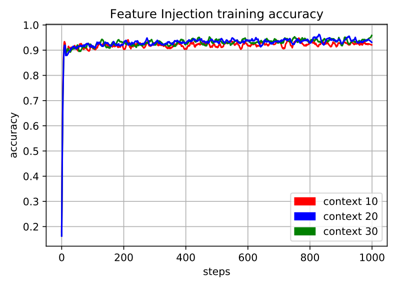
(c) Feature LSTM validation accuracy



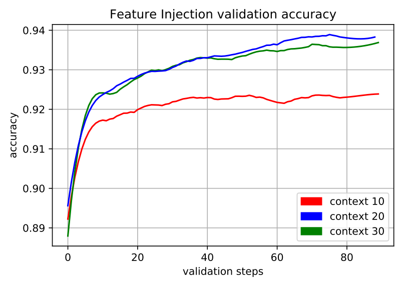
(d) Feature LSTM beam accuracy



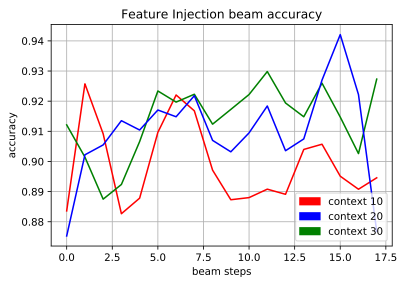
(e) Feature Injection training costs



(f) Feature Injection training accuracy



(g) Feature Injection validation accuracy



(h) Feature Injection beam accuracy

Figure C.2: Training runs of the Feature LSTM model and the Feature Injection model

Appendix D

Source-code Examples

```
\\generated on 0% context
UNK UNK = UNK.UNK()
\\original
ColorUIResource BLUE_FOCUS = BLUE_MEDIUM_LIGHTEST

\\generated on 50% context
String UNK = UNK.getHorizontalAlignment()
\\original
int align = label.getHorizontalAlignment()

\\generated on 90% context
String cat = (String) UNK.getSelectedItem()
\\original
String cat = (String) categoryBox.getSelectedItem()
```

Figure D.1: LSTM context size 30 regex

```
\\generated on 0% context
int i = g.getTransform()
\\original
ColorUIResource BLUE_FOCUS = BLUE_MEDIUM_LIGHTEST

\\generated on 50% context
String textArea = g.getWebBrowser()
\\original
JWebBrowser webBrowser = htmlEditor.getWebBrowser()

\\generated on 90% context
String operator = keyMapping.get(c)
\\original
Operator operator = keyMapping.get(c)
```

Figure D.2: LSTM context size 30 nltk

```
\\generated on 0% context
UNK UNK = UNK.UNK(UNK, UNK)
\\original
boolean disposingProgressWindow = false

\\generated on 50% context
String UNK = UNK() + "null" + "empty"
\\original
String mString = getProcessTitle() + ": <br><br>" + messages.getMessages()

\\generated on 90% context
String cat = (String) UNK.getSelectedItem()
\\original
String cat = (String) categoryBox.getSelectedItem()
```

Figure D.3: Feature LSTM context size 30 regex

```
\\generated on 0% context
String getTransform
\\original
Rectangle r5 = new Rectangle(B, B)

\\generated on 50% context
boolean getTransform = String.getWebBrowser()
\\original
JWebBrowser webBrowser = htmlEditor.getWebBrowser()

\\generated on 90% context
boolean fontSize = SubstanceSizeUtils.getComponentFontSize(splitPane)
\\original
int fontSize = SubstanceSizeUtils.getComponentFontSize(splitPane)
```

Figure D.4: Feature LSTM context size 30 nltk

```
\\generated on 0% context
TabContentPaneBorderKind kind = SubstanceCoreUtilities.UNK(this.tabPane)
\\original
Pattern pattern = Pattern.compile("Hello")

\\generated on 50% context
AffineTransform UNK = g.getTransform()
\\original
AffineTransform defaultTransform__0_115 = g.getTransform()

\\generated on 90% context
long serialVersionUID = -0
\\original
long serialVersionUID = -2061559337477351379L
```

Figure D.5: Feature Injection context size 30 regex

```
\\generated on 0% context
AffineTransform defaultTransform__0_0 = g.getTransform()
\\original
ColorUIResource BLUE_FOCUS = BLUE_MEDIUM_LIGHTEST

\\generated on 50% context
RText frame = LafUtils.setupStroke(g2d, name)
\\original
Stroke stroke = LafUtils.setupStroke(g2d, this.stroke)

\\generated on 90% context
String operator = keyMapping.get(c)
\\original
Operator operator = keyMapping.get(c)
```

Figure D.6: Feature Injection context size 30 nltk