Master Thesis

September 15, 2017

Deep Learning for Code Completion

Yury Belevskiy

of Moscow, Russia (15-718-117)

supervised by

Prof. Dr. Harald C. Gall

Carol V. Alexandru

University of Zurich UZH

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

Master Thesis

Deep Learning for Code Completion

Yury Belevskiy

Master Thesis

Author: Yury Belevskiy, yury.belevskiy@uzh.ch

Project period: 15th March 2017 - 15th September 2017

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

Abstract

We present a novel technique for method call completion in dynamically typed programming languages. Existing completion systems typically rely on language-specific heuristics or runtime information, because object types can rarely be identified from plain-text source code alone. Our approach uses recurrent neural networks for predicting method names based on the preceding context available in plain-text source code. Using source code of $1,000$ Python projects, we propose three preprocessor strategies that identify parts of the source code relevant for code completion and evaluate them quantitatively. We then compare the best of the resulting models to industry-leading code completion assistants. Our findings show that the proposed approach, based soley on plain-text source code, offers a level of quality for method name suggestions comparable to more complex state-of-the-art techniques. We further demonstrate that our approach can be applied to other dynamically typed programming languages without significant adaptation effort.

Zusammenfassung

Wir präsentieren eine neuartige Methode zur Auto-Vervollständigung von Methodenaufrufen in dynamisch typisierten Programmiersprachen. Existierende Vervollständigungssysteme benötigen zumeist sprachen-spezifische Heuristiken oder Laufzeit-Informationen, weil Objekttypen selten nur auf Basis des Klartext-Quellcodes identifiziert weden können. Unser Ansatz verwendet Recurrent Neural Networks um Methodennamen allein auf Grund des verfügbaren, vorangehenden Kontexts im Quellcode vorherzusagen. Mittels dem Quellcode aus 1000 Python Projekten erzeugen wir drei Datenverarbeitungs-Strategien, die die für die Vervollständigung relevanten Quellcodeteile identifizieren, und evaluieren diese quantitativ. Dann vergleichen wir das beste der resultierenden Modelle mit industrieführenden Code-Vervollständigungs-Assistenten. Unsere Ergebnisse zeigen, dass unser Ansatz, basierend allein auf Klartext-Quellcode, eine vergleichbare Qualität der vorgeschlagenen Vervollständigungen aufzeigt wie moderne, komplexere Ansätze. Des weiteren zeigen wir, dass unser Ansatz ohne signifikanten Arbeitsaufwand für andere dynamisch typisierte Sprachen eingesetzt werden kann.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

The vast majority of modern integrated development environments (IDEs) offer a code completion feature to assist developers in writing code. Code completion has proven to be one of the most popular IDE features standing amongst the top 5 most frequently used IDE commands [MKF06]. Reasons behind the wide usage of code completion systems are diverse. First, it provides "on the fly" suggestions for method names and variable types that are syntactically correct in the current context. For example, given a variable of a particular type, a code completion system would suggest methods that are available only to that type and it's ancestor types. Furthermore, code completion systems in modern IDEs such as Eclipse, IntelliJ or XCode are often implemented as a pop-up dialogue displaying a list of all available completions which serves as a compact version of the class documentation and allows for API exploration. Completion assistance does not only save developer's time, but also promotes the use of encapsulation and more descriptive method naming [HP11].

Mainstream code completion systems use static type information to generate a list of completion suggestions such as method names, class names or language-specific keywords. The resulting list is complete, but possibly unnecessarily large, including rarely used methods and methods inherited from superclasses. Existing code completion systems are especially unhelpful when class definitions are large or there is a long inheritance chain. There has been a lot of research and development effort on improving existing code completion systems done in past years [NNN$^+$12] [BMM09] [RL08]. For example, statistical techniques such as k-nearest-neighbors have been successfully applied in intelligent code completion plugins [cod] that have significantly improved developer experience [BMM09].

Most of the development and research in the field of code completion has been concentrated around statically typed programming languages such as Java or C++. IDEs for programming languages with dynamic type system such as Python or Ruby often offer no code completion feature at all. The reason is that in dynamically typed languages, type information for variables is available only at runtime, thus making it impossible to make completion prediction based on type information during writing or editing code. There are few implementations of auto-completion tools for dynamically typed languages available. For example, PyCharm is a commercial Python IDE featuring code completion functionality [pyc] or Jedi, open-source auto-completion/static analysis library for Python, which has been integrated into many popular code editors [jed]. Existing solutions perform completion task using static code analysis techniques in order to infer variable types from available context. This approach has proven successful, however, it is subject to many limitations. Firstly, completion suggestions offered are often too broad and irrelevant. Secondly, the implementation is highly dependent on the language syntax and, therefore, cannot be easily adopted for to suggest method names for other programming languages with dynamic type system.

In this paper, we present an intelligent code completion system for dynamically typed programming languages that uses a trained neural network model to make smart completion predictions. Particularly, this thesis is going to focus on developing a model for predicting method names in dynamically typed programming languages and, for the rest of the paper, the term 'code completion' will refer to method name completion.
Therefore, we aim to answer the following research question:

RQ: Can sequential neural models, trained on plain-text source code, be used to predict function calls in dynamic programming languages?

In order to train a neural network to predict method name suggestions, it is crucial to identify and extract parts of the code file that are relevant for method name prediction. This paper is going to propose and evaluate three different preprocessor strategies for extracting relevant information from the code. These strategies will be described and explained in detail in Section 3.2. For each strategy there is a corresponding source code preprocessor script which extracts relevant data from the dataset containing source code files in accordance with the rules defined by the stragety. As an output, the script generates source and target datasets that are used for training the neural network model. To evaluate these models, we have downloaded $1,000$ Python repositories containing over $100,000$ source code files from the GitHub code-hosting web service (see Section 3.1). The collected data has been processed using the aforementioned preprocessor scripts and six neural-based models have been trained, one for each model/dataset pair. To demonstrate the efficiency of the suggested approach, a code completion plugin for NetBeans IDE has been developed which uses a trained neural network to make intelligent completion predictions.

The suggested approach to use recurrent neural networks to solve method name completion task benefits over existing attempts that are using static code analysis in multiple ways. Firstly, recurrent neural networks are able to accurately capture which methods developers are using more frequently. For example, given piece of code like presented below and suppose that developer has a caret after `os.`:

```
import os

def important_function():
        thesis_folder = "C:\Users\ThesisUser\MyThesis\"
        thesis_file = "thesis.txt"
    os.path.isfile(os.)
```

PyCharm code completion library and Jedi that are using static code analysis for method name completion would offer a long, alphabetically ordered list of methods available in Python `os` package. In contrast, the proposed approach would list `os` module methods that other developers frequently use if the preceding code was `os.path.isfile`. Completions suggestions predicted using our model would be sorted based on how frequently they are used within the training dataset.

This thesis also demonstrates that our approach to the method name completion task can generalize to other programming languages with dynamic type system.

1.2 Related Work

A lot of research has been done in the area of code completion recommender systems. By contrast, the field of code completion for programming languages with dynamic type systems, specifically, is relatively new and undiscovered. To our knowledge, there have been no known attempts to generate method name suggestions using neural network models.
However, there are multiple research papers that apply intelligent statistical techniques including deep learning to enhance the performance of other productivity tools offered by modern integrated development environments.

1.2.1 Code Completion Systems

In this section, we present an overview of existing approaches on how code completion assistants can be improved.

An enhanced method completion suggestion system that leverages the information about code changes was proposed by *Nguyen et al.* [NHC$^+$16]. The contribution of the paper is a code completion assistant that can predict the most suitable method name completion given the surrounding context and a history of code changes. The results show that the suggested approach improves the quality of top-1 suggestions between $30 - 160\%$ compared to existing state-of-art techniques.
The *locality* property of the source code was exploited in order to increase the accuracy of method name completion suggestions [TSD14]. *Tu et al.* argue that a source code is *locally repetitive* i.e. it has useful local regularities that can be captured in a locally estimated cache and leveraged for software engineering tasks. The N-gram statistical language model with the proposed cache component was demonstrated to outperform the standard version of the N-gram model between 21.91% and 27.38%.
Bruch et al. demonstrated an *example based code completion system* that aims to improve the quality of code completion suggestions [BMM09]. The proposed system uses *Best Matching Neighbors* algorithm to match current completion context with code examples from the training dataset using *k-Nearest Neighbor* machine learning technique. Based on precision, recall and F1 metrics, the presented approach significantly outperformed the code completion system integrated into the Eclipse IDE, one of the most popular IDEs for Java programming language.
Lee et al. proposed a novel approach to method name completion by adding a temporal dimension [LHKM13]. The basic idea is to locate context that is relevant to the completion assistant in past versions of the software and use that information to make the output of completion engine more precise.
An example of intelligent code completion system was proposed by *Proksch et al.* [PLM15]. The contribution of the paper is a *Pattern-based Bayesian Networks* technique which leverages structural context features from a program file to generate context-relevant method suggestions. Indeed, the approach was demonstrated to offer more accurate and relevant method name suggestions compared to intelligent code completion assistants that are based on *Best Matching Neighbors* algorithm.
The novel code completion system, named *Abbreviated Completion*, supporting completion of mul-

tiple keywords, was proposed by *Han et al.* [HWM09]. The approach is based on a Hidden Markov Model that was trained on a corpus consisting of many code files. Apart from being capable of predicting multiple keywords, the proposed completion engine can also make predictions using non-abbreviated input i.e. given `gval(r,c)`, the system will propose `getValueAt(row,col)`. Whilst achieving 98.9% accuracy, the system also demonstrated impressive 30.4% in time savings.
An innovative technique called *Context Sensitive Code Completion* for improving the performance of a code completion engine was presented by *Asaduzzaman et al.* [ARSH14]. The proposed algorithm considers four code lines surrounding the line, where the completion prediction should be generated, as a relevant completion context. Using code examples from the training dataset, the system is able to suggest relevant method name completions. The approach was evaluated against five state-of-art intelligent code completion recommenders and was found to outperform them.
It was demonstrated that leveraging a change-based data from the program history can improve the accuracy of code completion suggestions [RL08]. By modeling a software evolution process as a sequence of changes to the program code and archiving those changes, a historical data from code completion engine could be extracted that contained information which of the previously suggested method names were chosen in past providing strong hints which methods are best offer given the context. The results show that the proposed system was able to include first choice method name in top-3 suggestions in 75% of cases.

1.2.2 Intelligent Software Productivity Tools

This section presents a brief summary of state-of-art techniques and solutions in a field of intelligent software productivity tools.

Raychev et al. demonstrated that recurrent neural networks can be successfully applied to synthesize completion suggestions for gaps in a program file [RVY14]. The shortcoming of the proposed approach that the hole completion feature is rarely used and, typically, is not a part of the functionality offered by most IDEs.
A decision-tree based approach to make predictions about the program structure was proposed by *Raychev et al.* [RBV16]. They have demonstrated that the problem of learning probabilistic model of code can be reduced to learning a decision tree on the abstract syntax tree generated from the code. The demonstrated result can be used to augment existing code completion functionality, however, can not completely replace it.
Hellendoorn et al. evaluated widely applied N-gram statistical technique against neural network based approaches such as RNN and LSTM in the context of source code modeling tasks [HD17]. The results show that the N-gram model that is carefully adapted for source code can outperform deep learning techniques.
A type inference engine for Python programming language was presented by *Xu et al.* [XZC$^+$16]. Python features a dynamic type system, therefore, the task of deriving an exact variable type in the given context is challenging. The innovative technique uses probabilistic inference to leverage type hints in a program code such as variable naming and accessed attributes in order to obtain probabilities on potential candidates for a variable type. The results demonstrate that proposed type inference engine outperforms known solutions by 79.09%.
Gu et al. presented a novel deep learning based approach to produce API usage examples for a given natural language query [GZZK16]. The proposed system adopts recurrent neural network with the encoder-decoder architecture that is used to convert an input language query into the representation of fixed dimensionality and to generate an API usage example from the obtained representation. The proposed approach was evaluated against existing API learning tools that

use statistical techniques demonstrating 264% improvement in the suggestion quality.
The concept of integrating an additional information about the object into pop-up dialogue windows that are used by code completion assistants was proposed by *Omar et al.* [OYLM12]. The paper introduces *palettes* which are highly-specialized, interactive interfaces allowing developer to receive extra information about the methods for given object type.
Hou et al. presented a set of strategies for organizing method name proposals in code completion pop-up dialogues [HP11]. Fourteen different strategies were evaluated and analysed on nine open source Java applications. The contribution of the paper is a set of design recommendations for future code completion assistant implementations.

Chapter 2

Background

This chapter introduces basic terminology in the source code analysis field and presents an overview of deep learning and statistical techniques that form the basis of this thesis.

2.1 Programming Languages

This section introduces terminology for relevant topics in the field of source code analysis.

2.1.1 Variable Scope

Variables in a program cannot be always accessed from all parts of the program depending on where the variable was defined. *Scope* is a part of a program where a variable is accessible from. The scoping rules are determined by the programming language semantics. In this section, we consider scoping rules in the Python programming language because Python is the main programming language that we use throughout the project.
The scope resolution in Python is done using *LEGB* rule. The *LEGB* rule is defined by [Lut08] as follows:

- **L**: Names assigned in any way within a function (`def` or `lambda`), and not declared global in that function.

- **E**: Names in the local scope of any and all enclosing functions (`def` or `lambda`), from inner to outer.

- **G**: Names assigned at the top-level of a module file, or declared global in a `def` within the file.

- **B**: Names preassigned in the built-in names module. For example, `open`, `len`, `RuntimeError`.

When a variable is referenced in the code, Python searches for it in the *LEGB* order starting from *L*.
The scope resolution for classes in Python is categorized under the *L* part of the *LEGB* rule. However, the scope resolution rules for instance variables in Python is an unusual case. Consider the example below:

```
1 class SomeClass(self):
2
3         def __init__(self):
4             self.list = []
5
6         def do():
7             list = [5,6]
8             list.extend(3)
9             self.list.extend(list)
```

Figure 2.1: An example of Python class

The `list` variable declared on the line 7 is accessible only within `do()` function scope according to the *LEGB* rule. However, the variable `list` which is declared on line 4 is accessible in the `do()` function even though it points to the different underlying object rather than `list` variable declared on the line 7. The reason is that the `list` variable instantiated on the line 4 is an instance variable of the `SomeClass` class. To access an instance variable anywhere within a class, it is sufficient to prepend `self.` before the variable name.

2.1.2 Abstract Syntax Trees

An abstract syntax tree, AST, is a tree-like data structure that represents the abstract syntactic structure of the program. One of the advantages of the AST source code representation is that an AST does not include parts of the original program that do not contribute to the semantic meaning of the program. These parts include semicolons, braces, comments, dots, spaces and more.
To understand how a sample AST looks like, we consider the following Python code snippet:

```
list = []

def main():
    #adding 5 to list
    list.append(5)
```

Figure 2.2: An example code snippet

The AST corresponding to this code snippet is presented on Figure 2.3. The AST shown on Figure 2.3 captures full semantic meaning of the code snippet shown on Listing 2.2. We can observe that the parts of the code snippet that do not add to the semantic meaning of the program, such as comments, spaces, braces, colons, are removed from the AST representation.

2.2 Neural Networks

This section overviews different types of neural networks that we will be operating in later sections. In Section 2.2.1, we are going to introduce a basic model of *neuron* and cover basic operation principles behind *neural networks* to form the basis for later discussion. Section 2.2.2 presents *recurrent neural networks* that are specific type of neural networks capable to deal with arbitrary-sized

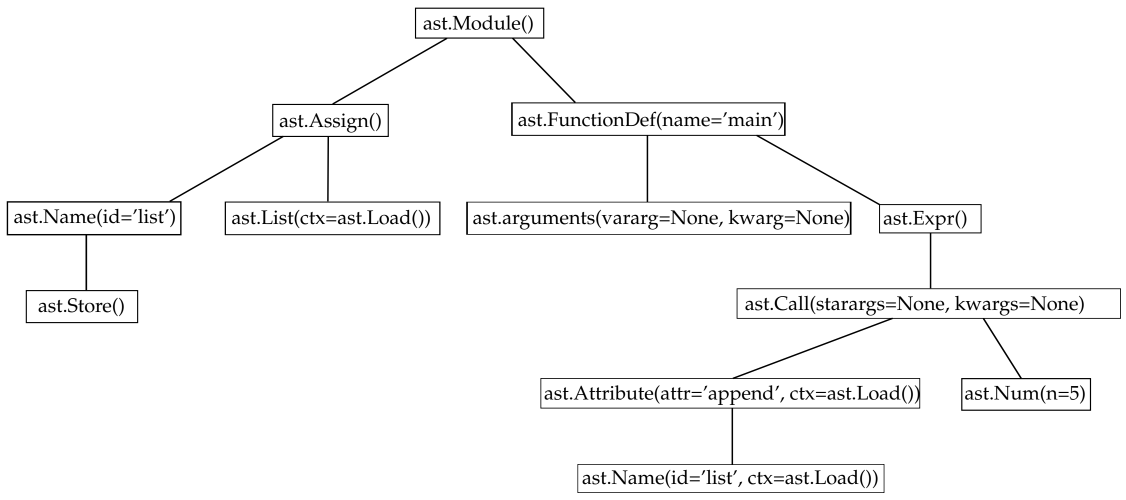

ast.Module()
ast.Assign()
ast.FunctionDef(name='main')
ast.Name(id='list')
ast.List(ctx=ast.Load())
ast.arguments(vararg=None, kwarg=None)
ast.Expr()
ast.Store()
ast.Call(starargs=None, kwargs=None)
ast.Attribute(attr='append', ctx=ast.Load())
ast.Num(n=5)
ast.Name(id='list', ctx=ast.Load())

Figure 2.3: The AST for the example code snippet

input and output vectors. *Long Short Term Memory network*, also known as LSTM, is a variant of recurrent neural network adapted to deal with long-term dependencies which performs particularly well on sequence-to-sequence translation tasks. LSTM will be discussed in detail in Section 2.2.3.

2.2.1 Introduction to Neural Networks

2.2.1.1 Basic Model of Neuron

Neural networks have received first mention in scientific literature in early 1943. [MP43] have discovered that, for any logical expression satisfying certain conditions, called "net" exhibiting same behavior can be found. The organization and structure of the "net" was inspired from the neurobiological model of the human brain. The basic unit of the "net", an *artificial neuron*, has a similar organization to the biological neuron that is a basic building block of the human brain.
Figure 2.4 and Figure 2.5 showcase diagrams representing the structure of the biological neuron and the structure of an artificial neuron respectively. In biological neurons dendrites are used

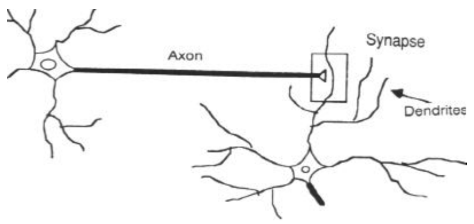


Figure 2.4: Structure of biological neuron [syn]

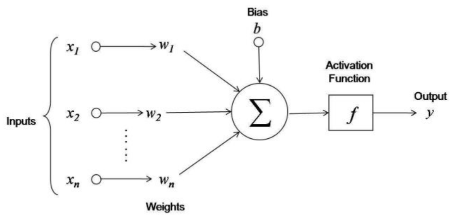


Figure 2.5: Structure of artificial neuron [art]

to receive electrical signals from the axons of other neurons. In artificial neurons these electrical signals are represented as numerical values and act as input values into the neuron. At the synapses between the dendrite and cell body, incoming electrical signals are modulated in various amounts. Artificial neuron models the 'modulation' of incoming signals by storing such a weight matrix so that every input value has a corresponding entry in the weight matrix. When an input value is received, it is multiplied by the corresponding weight matrix entry. A neuron inside human brain fires an output signal only when the total strength of the input signals exceeds a certain threshold. Artificial neuron behaves in a similar way to the biological neuron: instead of storing an activation threshold value, it feeds weighted sum of inputs into the activation function and outputs a value that is within a fixed numerical range, typically, $[0, 1]$, however, precise range values depend on the type of activation function. In mathematical form, an artificial neuron can be expressed:

$$\begin{aligned} f - &\ \texttt{activation function} \\ W - &\ \texttt{weight matrix} \\ x - &\ \texttt{vector of input values} \\ c - &\ \texttt{normalization constant} \end{aligned}$$

$$\texttt{output} = f(Wx + c)$$

There are different types of activation functions that are used in practice. The most common and well-known activation function is a *sigmoid* function that takes any numerical input and "squashes" it between zero and one. Figure 2.6 demonstrates a plot of *sigmoid* function. Despite

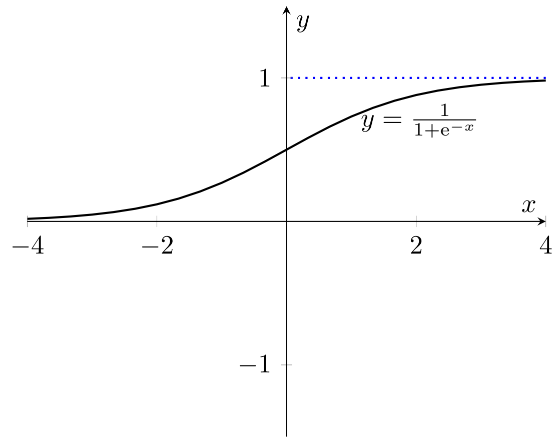


Figure 2.6: Sigmoid function

being one of the most popular activation functions, *sigmoid* function possesses certain disadvantages:

- **Gradient saturation problem**: the gradient of *sigmoid* function at tails is nearly zero. This is undesirable because during the update of neurons weights, known as *backpropagation* (discussed in Section 2.2.1.2), when the gradient of gate's output is multiplied by the local gradient i.e. the gradient of *sigmoid* function, low value of the local gradient would "saturate" the value of gradient at the gate's output, resulting in minor weight updates. In other words, "saturation" gradient problem significantly slows down the network learning rate.
- **Output isn't zero-centered**: Input layer neurons will always output a positive value (recall that the *sigmoid* function outputs values in $[0, 1]$ range) resulting in always positive input

for neurons in successor layers. If a neuron always receives positive input values, then, during backpropagation, the gradient on the weights will become either all positive or all negative. This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights [neu].

Another common choice for an activation function is the *tanh* function. *Tanh* function takes any numerical input and produces an output in $[-1, 1]$ range. The plot representing *tanh* function is demonstrated on Figure 2.7. The choice of *tanh* function as an activation function solves the prob-

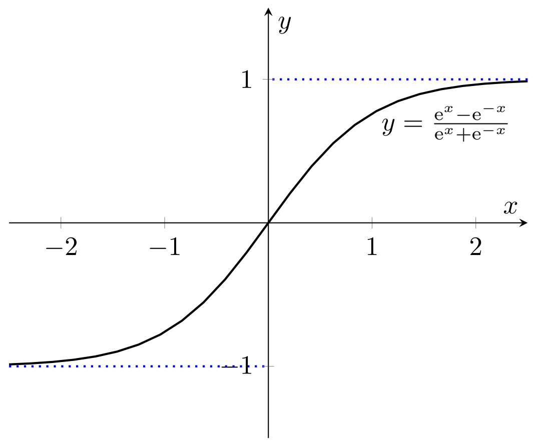


Figure 2.7: Tanh function

lem of non zero-centered output that has been discussed before, however, the gradient saturation problem still persists. There are different types of activation functions which solve all of the aforementioned issues, for example, *ReLU* or *maxout* activation functions. *ReLU* non-linearity is widely applied in convolutional feed-forward neural networks, however, our approach is to use LSTM cells as a basic building block when constructing a neural network (discussed in Section 2.2.3). LSTM cells internally use *sigmoid* and *tanh* activation functions, so we are not going to consider alternative activation functions in detail.

2.2.1.2 Backpropagation and Learning

Having discussed the basic structure of artificial neuron in Section 2.2.1.1, this section is going to explain how neural networks learn during the training process. The process of training a neural network can be broken down into multiple steps:

1. Feed a data sample into the network and make a forward-pass through the network
2. Once the forward-pass is complete, computed values are collected from the output layer.
3. Output values are compared with expected values from the training dataset and value of error using chosen metric is computed. There are different error metrics available, but the most commonly applied metric is mean squared error.
4. Depending on the value of error and sign of the error value, we would like to adjust weights of neurons in the network in such a way that the value of the error decreases. Importantly, the amount by how much we would like to reduce an error or so-called *step size* should be chosen very carefully. Choosing a large step size results in a trained model with low

predictive ability. On the other hand, if the size is too small, the network would take a long time to learn.

5. After deciding in which direction and by which amount the error value should be reduced, the next step is to update weights of neurons throughout the entire network correspondingly so that if the same data sample is fed into the network again, the resulting error value would be smaller compared to the error value from the previous forward-pass. The process of updating neuron weights to reflect desired change on resulting error value is called *backpropagation*.

Let's consider an example to understand how backpropagation works in practice. Suppose we have a two-dimensional artificial neuron that computes a function $f = \sigma(ax+by+c)$. An example of such neuron is shown on Figure 2.8. From Figure 2.8 we can observe that the Σ gate computes

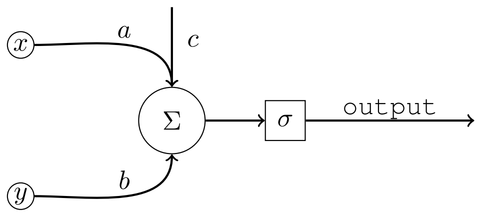


Figure 2.8: 2-dimensional artificial neuron

$ax + by + c$ and σ gate computes $\sigma(ax + by + c)$.
Let's set weights a, b and c to initial values of 0.5, 1.5, -1.0 respectively and let's assign $x = 4.0$ and $y = -2.0$. Substituting these values into f gives:

$$f = \sigma(ax + by + c)$$
$$f = \sigma(0.5 * 4.0 + 1.5 * (-2.0) - 1.0)$$
$$f = \sigma(-2.0)$$
$$f = 0.119$$

The neuron outputs a value of 0.119, however, our training data tells us that the expected output value for $x = 4.0$ and $y = -2.0$ should be 0.5. We would like to perform weight adjustment so that the output value of the neuron is closer to 0.5 given $x = 4.0$ and $y = -2.0$ as inputs. To perform adjustment of weights, the value of the gradient needs to be computed first with respect to weights a, b and c:

$$\frac{\partial(ax + by + c)}{\partial a} = x$$
$$\frac{\partial(ax + by + c)}{\partial b} = y$$
$$\frac{\partial(ax + by + c)}{\partial c} = 1.0$$
$$\frac{\partial\sigma(s)}{\partial s} = (1 - \sigma(s)) * \sigma(s)$$

To find out the gradient of $\sigma(ax + by + c)$ w.r.t to a, b and c, the differentiation chain rule has to be applied:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} * \frac{\partial y}{\partial x} \quad \texttt{(Chain Rule)}$$

$$\frac{\partial\sigma(ax+by+c)}{\partial a} = \frac{\partial\sigma(ax+by+c)}{\partial(ax+by+c)} * \frac{\partial(ax+by+c)}{\partial a} = (1-\sigma(ax+by+c)) * \sigma(ax+by+c) * x$$
$$\frac{\partial\sigma(ax+by+c)}{\partial b} = \frac{\partial\sigma(ax+by+c)}{\partial(ax+by+c)} * \frac{\partial(ax+by+c)}{\partial b} = (1-\sigma(ax+by+c)) * \sigma(ax+by+c) * y$$
$$\frac{\partial\sigma(ax+by+c)}{\partial c} = \frac{\partial\sigma(ax+by+c)}{\partial(ax+by+c)} * \frac{\partial(ax+by+c)}{\partial c} = (1-\sigma(ax+by+c)) * \sigma(ax+by+c) * 1.0$$

Substituting values for variables:

$$\frac{\partial\sigma(ax+by+c)}{\partial a} = (1-\sigma(-2.0)) * \sigma(-2.0) * 4.0 = 0.420$$
$$\frac{\partial\sigma(ax+by+c)}{\partial b} = (1-\sigma(-2.0)) * \sigma(-2.0) * (-2.0) = -0.210$$
$$\frac{\partial\sigma(ax+by+c)}{\partial c} = (1-\sigma(-2.0)) * \sigma(-2.0) * 1.0 = 0.105$$

Setting step size to 0.1 and updating weight of a:

$$\begin{aligned} &a \mathrel{+}= step * gradient \\ &a \mathrel{+}= 0.1 * 0.420 \\ &a = 0.5420 \end{aligned}$$

Updating weight of b:

$$\begin{aligned} &b \mathrel{+}= step * gradient \\ &b \mathrel{+}= 0.1 * (-0.210) \\ &b = 1.4790 \end{aligned}$$

Updating weight of c:

$$\begin{aligned} &c \mathrel{+}= step * gradient \\ &c \mathrel{+}= 0.1 * 0.105 \\ &c = -0.9895 \end{aligned}$$

Let's make another forward-pass using $x = 4.0$ and $y = -2.0$ as an input values through the network with updated weights to make sure that the output value is closer to the expected value of 0.5:

$$\begin{aligned} &f = \sigma(ax+by+c) \\ &f = \sigma(0.5420 * 4.0 + 1.4790 * (-2.0) - 0.9895) \\ &f = \sigma(-1.7795) \\ &f = 0.144 > 0.119 \end{aligned}$$

We have demonstrated how gradients propagate backwards through the network to adjust values of neurons' weights. As a general case, backpropagation can be applied on the network scale recursively updating weights of neurons throughout the neural network.

2.2.1.3 Neural Network Architectures

The neurons of a neural network form a directed acyclic graph where the edges represent the flow of information. A common architecture of neural networks is a layer-wise structure where

neurons are grouped in layers with an output of one neuron layer acting as an input to the next neuron layer. The most common layer type is a fully-connected layer: neurons within a single layer share no connections, however, two adjacent neuron layers form a fully connected bipartite graph.
A 2-layer sample neural network is shown on Figure 2.9. Even though it may seem counterintu-

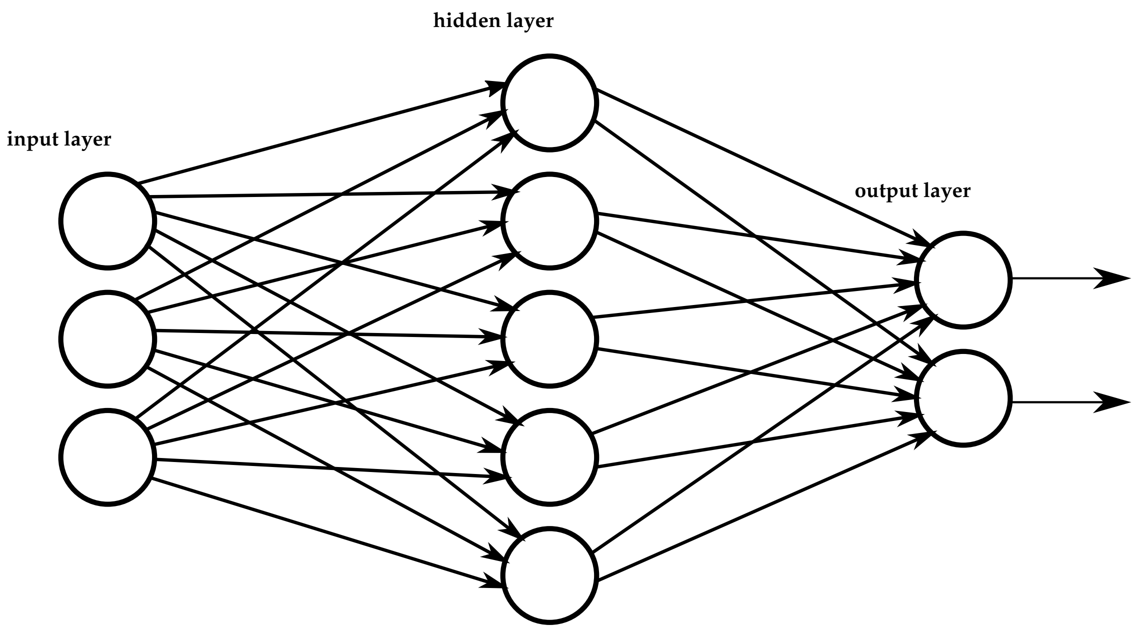


Figure 2.9: Neural Network Architecture

itive as the network depicted on Figure 2.9 has three layers, the standard naming convention for neural networks assumes that the input layer is not counted. N-layer neural network has single input layer, $N-1$ hidden layers and single output layer.
The *input layer* is an entry point of any neural network where the input values from training or real data are being fed to. Consider training a neural network for image classification, pixel intensity values of images from the training dataset will be fed to the input layer.
Hidden layers are transforming input data into the output data by applying transformations to the input values at every layer. For instance, if building a simple face detector, we can setup a 3-layer neural network where each hidden layer will be responsible to detect certain feature of a human face. First hidden layer can represent a nose detector whereas second hidden layer can represent a mouth detector. Such face detector would be much more effective compared to a face detector that passes weighted pixel intensity value vectors from input layer directly to the output layer attempting to learn how to detect faces using single layer of weight matrices .
The goal of the *output layer* is to represent the output of the neural network which comes in form of real-valued numbers. A neural network classifier that, given an image, is trained to determine whether the input image depicts cat, dog or none of the above, would have an output layer consisting of three neurons where each neuron would output a score how likely the input image is to belong to one of three aforementioned classes.

2.2.1.4 Limitations of traditional neural networks

Traditional neural networks are widely applied in the field of image recognition, can be trained to solve classification problems, however, they rely on two basic assumptions:

- Input samples are independent of each other
- Input and output are represented by fixed-size vectors

These assumptions significantly reduce the scope of problems that ordinary neural networks can solve. For example, if we want to build a neural network that predicts next word in a sentence, we cannot assume that previous words in that sentence are independent of each other. Therefore, we cannot apply traditional neural networks to solve the problem of code completion: context information from the source code such as variable declarations, previous use of the variable and preceding lines of code are highly relevant for accurate method name completion.

2.2.2 Recurrent Neural Networks

Traditional neural networks accept a fixed-size vector as an input and produce a fixed-size vector as an output which significantly limits their application scope. In contrast, recurrent neural networks, known as RNNs, allow to operate on sequences of vectors for both input and output. Figure 2.10 illustrates examples of possible RNN organizations. The leftmost network structure in

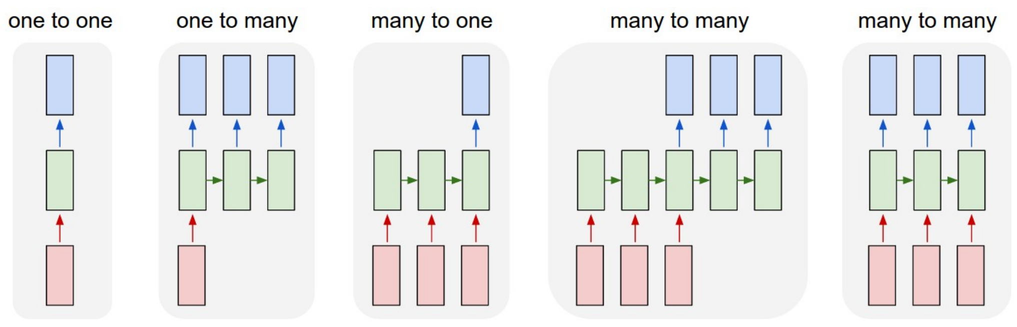


Figure 2.10: Examples of RNN structures [rnn]

Figure 2.10 represents a traditional neural network which accepts a single, fixed-size input vector and produces a single, fixed-size output vector. The other 4 network structures depicted in Figure 2.10 represent different RNN structures. For instance, *one-to-many* RNN can be useful to build an application that generates a textual image description by taking a single image as an input and generating a multi-sentence description as an output. *Many-to-many* RNNs are widely used for sequence-to-sequence translation tasks.
Consider a time step, t, and an input vector that is fed into RNN during t denoted as x. During the forward-pass at t, the RNNs takes, in fact, two inputs: the first input is an input vector representing new data, x, whereas second source of input is a hidden state vector which the RNN has learned during the previous $t-1$ steps. Figure 2.11 illustrates the internal organization of RNN in detail.

RNN logic during forward-pass at time t is presented below:

$$\begin{aligned} &x - \texttt{ input vector at time } t \\ &h - \texttt{ hidden state vector} \\ &y - \texttt{ output vector at time } t \end{aligned}$$

$$\begin{aligned} h_t &= tahn(W_h * h_{t-1} + W_x * x_t) \\ y_t &= W_y * h_t \end{aligned}$$

An RNN hidden layer unit, shown on Figure 2.11, uses *tanh* activation function that has been discussed in detail in Section 2.2.1.1.
To improve the learning ability of a neural network, the typical deep learning approach is to increase the number and the size of hidden neuron layers. To increase the learning capability of an RNN, one could stack up multiple RNNs on top of each other: one RNN receives input vectors and it's output vectors are the input to the next RNN. Using that way, we could connect any number of RNNs between each other.
Like traditional neural networks, RNNs also learn using backpropagation. Weights of neurons in the network update as gradients propagate from the topmost RNN layer, which produces output vectors, through hidden RNN layers towards the RNN input layer.

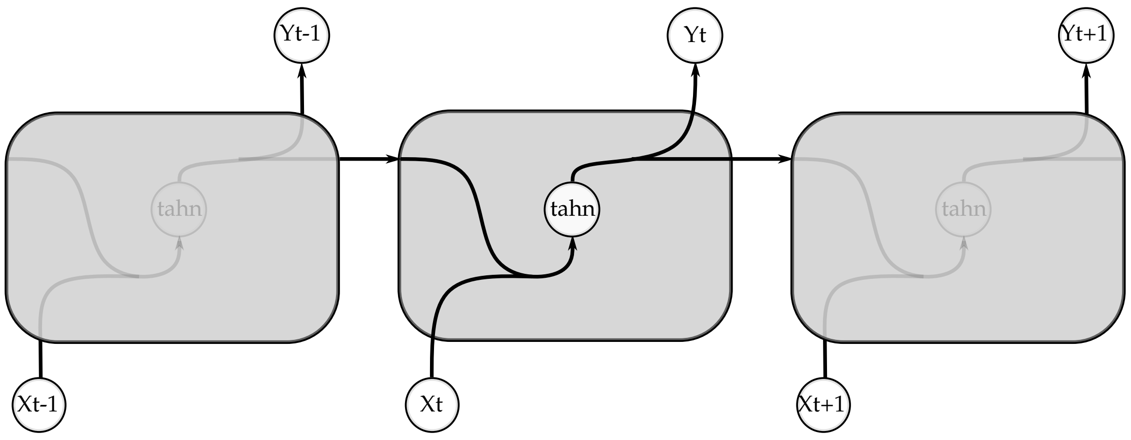


Figure 2.11: RNN internal organization

Despite a high degree of flexibility that RNNs provide, allowing to process input sequences of arbitrary size, [BSF94] demonstrated that RNNs using tanh hidden layer units fail to capture long-term dependencies. For instance, the RNN based on tanh hidden layer units could easily predict that blank space in sentence "My name is ..." should be filled with one of names that occurred in the training dataset. However, given a chunk of text "Most of my life I have spent in Russia. I have been actively studying language and culture of that beautiful country. After years of practice, I can speak ... fluently.", the RNN is likely to fail to predict word "Russian" for the blank space, because relevant context which is "Russia" in this case has occurred many words ago. Theoretically, one could tweak the RNN hyper-parameters in order for it to be able to make correct prediction for specific scenario demonstrated above, however, it is not a practical approach for most applications.
In Section 2.2.3, we are going to discuss special kind of RNNs, called *Long Short Term Memory*

networks (LSTM), that have been specifically designed to learn long-term dependencies.

2.2.3 LSTM

2.2.3.1 Introduction into LSTM

Long Short Term Memory networks, also known as LSTM, were introduced by S. Hochreiter and J. Schmidhuber in 1997 [HS97]. LSTM networks were developed to avoid long-term dependency problem that traditional RNNs suffer from: recurrent nets fail to learn if the gap between relevant information and the place where that information is needed becomes large. LSTM structure is very similar to RNN structure, however, the key difference, which allows LSTM to handle long-term dependencies, is an organization of a hidden layer units which have a much more sophisticated structure in LSTM compared to RNN. Figure 2.12 shows the internal organization of the LSTM hidden layer unit:

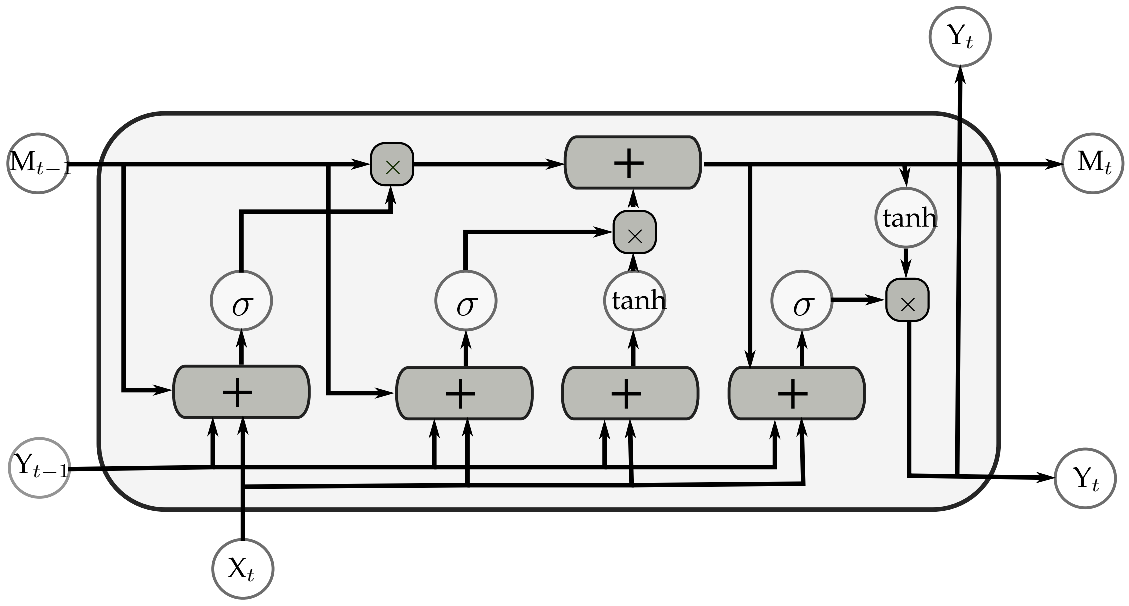


Figure 2.12: Structure of an LSTM hidden layer unit

2.2.3.2 LSTM structure

To assist the understanding of the diagrams presented later in this section, relevant notations are shown on Figure 2.13. The LSTM hidden layer unit shown on Figure 2.12 takes three input vectors and produces two output vectors. The input vector Y_{t-1} represents the output of previous LSTM unit. X_t represents the input vector at the current time instance t. M_{t-1} represents the internal state of previous LSTM unit. Output vectors, denoted as Y_t and M_t, represent LSTM cell output and LSTM cell state respectively. On Figure 2.12, there is an arrow at the top of the cell that takes M_{t-1} as an input on the left side of the diagram and outputs M_t on the right side of the diagram. Later on, we will refer to this arrow as *cell state pathway*. The cell state is transmitted along the cell state pathway and gets modified at two gates, the multiply gate and the addition gate, until it is outputted as M_t. The *cell state pathway* is highlighted on Figure 2.14. The multiply

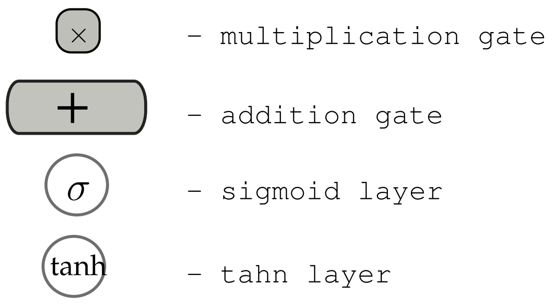


Figure 2.13: Graphical notation for LSTM diagrams

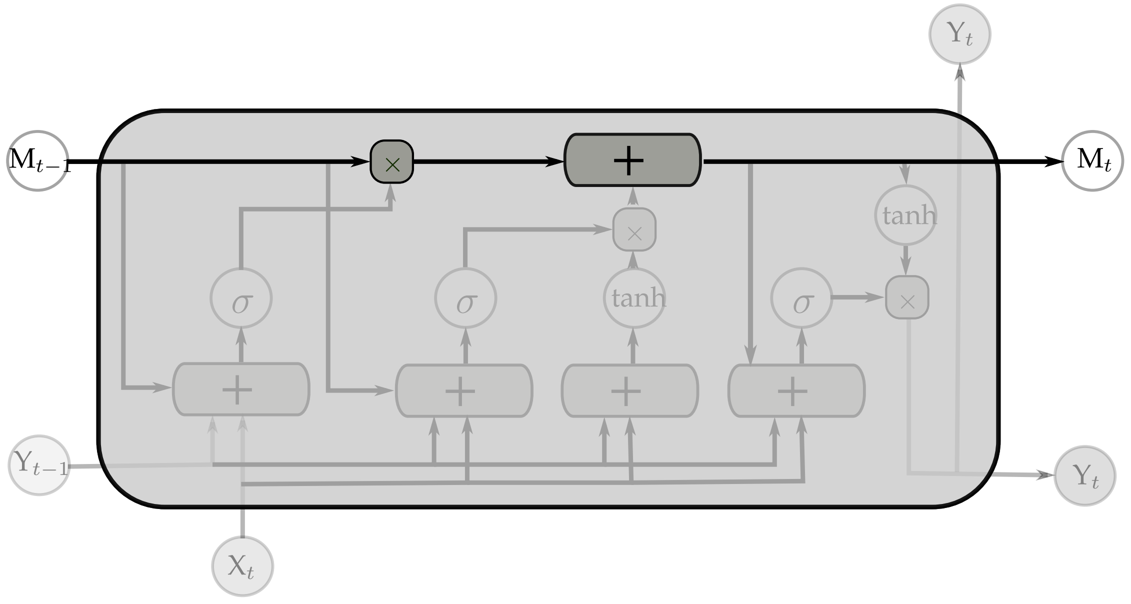


Figure 2.14: LSTM cell state pathway

gate receives two vectors as an input: M_{t-1} and F which is an output vector of the forget gate shown on the Figure 2.15. The forget gate accepts Y_{t-1}, X_t and M_{t-1} vectors as an input and is represented by a single layer neural network. The forget gate is responsible to decide how much of the previous cell state, M_{t-1}, to keep and how much to forget. The sigmoid activation function used in the forget gate produces an output vector containing values between zero and one. If the output vector, F, contains values that are close to zero, the previous cell state, M_{t-1}, will be mostly discarded because values of M_{t-1} will be saturated by F during vector multiplication at the multiply gate. In contrast, previous cell state, M_{t-1}, would be kept if values in F are close to one. A typical application of forget gate is when we are confident that the previously learned information is independent of the current input. For example, if we train an LSTM on a dataset containing independent text articles and, during the training, the LSTM has reached the end of an article and is about to start processing another article, it is useful to be able to forget previous state.

The equation representing the operation of "forget" gate is shown below:

$$W_f - \texttt{ weight matrix of the "forget" gate}$$
$$b_f - \texttt{ bias value of the "forget" gate}$$

$$F = \sigma(W_f \times (M_{t-1} + X_t + Y_{t-1}) + b_f)$$

The addition gate receives M_{t-1} and N' vectors as an input and outputs the updated cell state

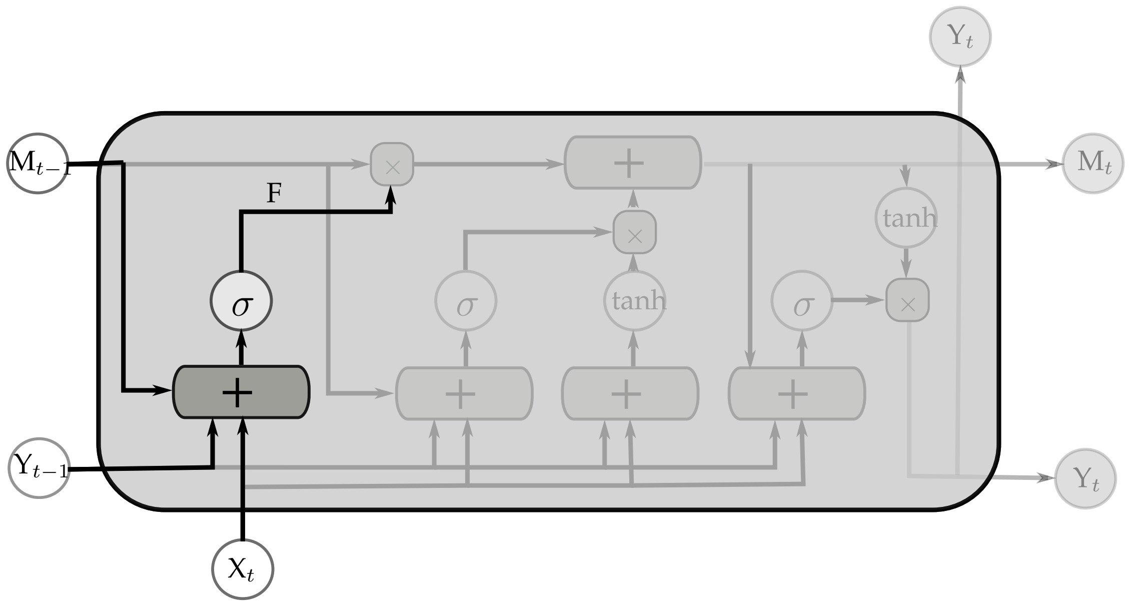


Figure 2.15: LSTM "forget" gate structure

vector, M_t. N' is the output vector of the "state update" gate which is depicted on Figure 2.16. The "state update" gate is responsible for adding new state information to the old cell state, M_{t-1}. The "state update" gate is composed of two layers, a sigmoid layer and a tanh layer. The sigmoid layer has the same structure as the "forget" gate which we have discussed previously. This layer is responsible for filtering which candidate values from the newly generated state, denoted as N, can pass onto the multiply gate and become a part of the updated cell state, M_t. The tanh layer is responsible to generate a list of candidate values to be included in the updated cell state. N', the dot product of the output vector from the sigmoid layer and the output vector from the tanh layer, is the vector representing a filtered list of candidate values to be included into the updated cell state. The equation below represents the computation performed by the "state update" gate:

$W_{fs}-$ `weight matrix of the sigmoid layer inside of the "state update" gate`
$b_{fs}-$ `bias value of the sigmoid layer inside of the "state update" gate`
W_s- `weight matrix of the tanh layer inside of the "state update" gate`
b_s- `bias value of the tanh layer inside of the "state update" gate`

$$S = \sigma(W_{fs} \times (M_{t-1} + X_t + Y_{t-1}) + b_{fs})$$
$$N = tanh(W_s \times (X_t + Y_{t-1}) + b_s)$$
$$N' = S \times N$$

The rightmost gate inside the LSTM unit depicted on Figure 2.17 represents the "output" gate.

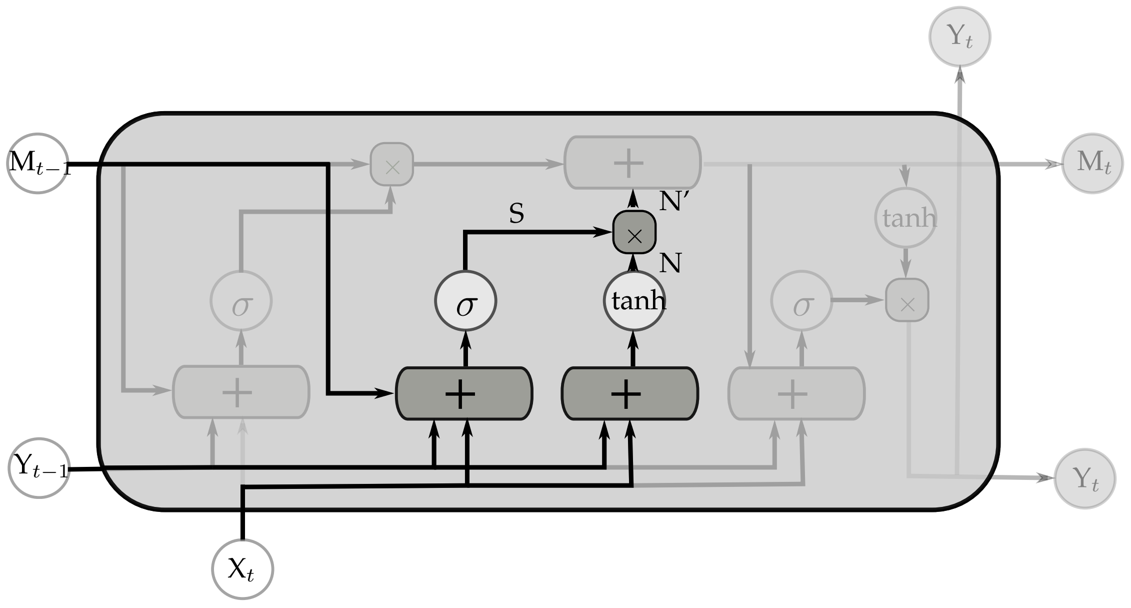


Figure 2.16: LSTM "state update" gate structure

The "output" gate is responsible to generate an output vector, y_t, which is an input to the next LSTM unit. The output vector is a filtered version of M_t because, typically, only few elements of the entire state information vector should be output. Recall our previous example of training LSTM on the chunk of text articles. The cell state may store contextual information from past sentences in the text chunk, but the LSTM is expected to output only a single word.
The sigmoid layer decides which parts of the cell state should be included in y_t. The tanh layer "squashes" M_t values between $[-1, 1]$ so that the output vector contains only normalized values. As a result of dot product between the sigmoid and tanh layer output vectors, y_t contains only filtered, normalized values. The mathematical expression computed by the "output" gate is pre-

sented below:

$$W_o - \texttt{ weight matrix of the "output" gate}$$
$$b_o - \texttt{ bias value of the "output" gate}$$

$$y_t = tanh(M_t) * \sigma(W_o \times (M_t + X_t + y_{t-1}) + b_o)$$

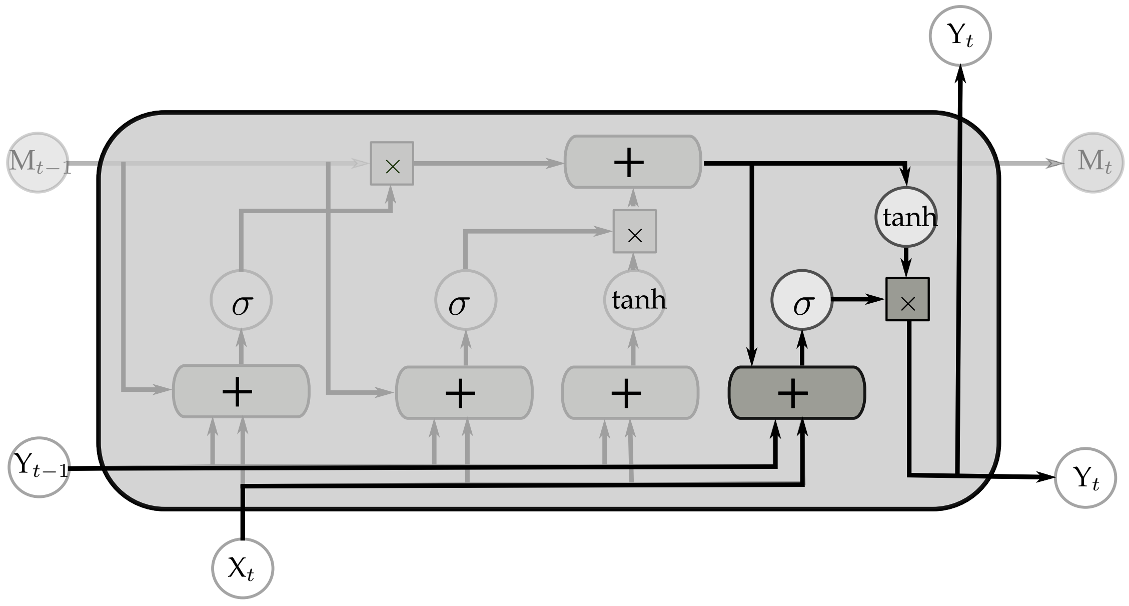


Figure 2.17: LSTM "output" gate structure

2.2.3.3 Neural Machine Translation

Neural Machine Translation, NMT, is a learning approach based on neural networks for automated translation tasks. NMT techniques have been shown to outperform existing machine translation algorithms on sequence-to-sequence learning problems [WSC+16] [SVL14].
The problem of proposing method name completions can be seen as a sequence-to-sequence translation task: the input sequence is a list of tokens obtained from the source code that contains useful contextual information and the method name to be predicted is the output sequence consisting of a single word.
Often, in sequence-to-sequence translation tasks, such as speech recognition or language translation, the dimensionality of the input and output vectors is not known in advance. To apply neural networks on the set of problems where the dimensionality of input and output sequences is variable, the *encoder-decoder* neural network architecture was proposed by [CVMG+14].
The basic outline of the *encoder-decoder* architecture is explained below:

- The encoder is represented by the RNN which reads every symbol of the input sequence step-by-step. Once the `end-of-sentence` symbol is reached, the encoder stops reading the data. The hidden state of the RNN, which represents the encoder, is a fixed-size vector representing the input sequence. Thus, the encoder allows mapping variable-length input sequences to the fixed-size representation.

- The decoder is another RNN that maps the fixed representation created by the encoder to the output sequence. Unlike traditional RNN, the decoder output depends on the previously outputted words and the context vector. The context vector is obtained by applying non-linearity to the sequence of hidden states.
- Encoder and decoder are, typically, trained together in order to maximize the probability of correctly predicting an output sequence given a source sentence.

The common choice for a hidden layer unit when using the *encoder-decoder* architecture is LSTM. The LSTM unit was discussed in detail in Section 2.2.3.
The performance of the *encoder-decoder* approach was demonstrated to significantly deteriorate as the length of input sequence increases [CVMBB14]. The reason is that the encoder layer needs to compress the entire input sequence into the fixed-size internal representation before starting to produce an output sequence using decoder.
To overcome this limitation, the attention-based RNN model was proposed [BCB14]. The key difference between the attention-based model and the encoder–decoder approach is that the attention-based model, instead of encoding a whole input sentence into a fixed-size representation, encodes every word from the input sequence into the separate vector. When producing an output sequence, the decoder searches the list of vectors generated during the encoding stage, where each vector corresponds to a single word in an input sequence. The decoder picks the subset of vectors from the list in order to generate an output sequence based on the relevance of each vector for producing an output sequence. It is advantageous to apply an attention-based model to the sequence-to-sequence translation problems where certain parts of the input sequence carry more information that can be used when generating an output sequence.

Chapter 3

Approach

This chapter presents the novel method for proposing method name completion suggestions developed throughout this project. Section 3.1 will describe the data gathering procedure. The models that we have used to preprocess the collected data are thoroughly explained in Section 3.2. The process of training the neural network model is described in Section 3.3. Section 3.4 presents the prototype code completion plugin that we have developed to demonstrate the viability of the proposed approach.

All source code files and scripts that we discuss in this Section are available at `http://tiny.uzh.ch/Kg`.

3.1 Data Acquisition

To tackle the problem of data collection, we have applied the structural approach proposed by [YWL06]. The suggested data preparation scheme consists of three phases: *data pre-analysis*, in which target data is identified and collected; *data preprocessing*, in which previously collected data is examined and analyzed and in which some data may be tailored to specific application via transformation or restructuration; and *data post-analysis*, in which some data is validated and re-adjusted. In this section, we are going to focus only on the first phase of the integrated data preparation scheme, namely, *data pre-analysis*. The following subsections will be structured according with the discussed data preparation scheme.

3.1.1 Data Requirement Analysis

The requirement analysis during the *data preprocessing* phase proposed by [YWL06] includes following steps:

- **What information we would like to have**: The information required, in order to train a neural network capable of predicting accurate method name suggestions, should feature multiple components. Firstly, an exhaustive list of method names available both, in core Python libraries and third-party libraries, should be gathered. Secondly, we should collect all context information available in the source code file which can be potentially relevant for the method name prediction task. In case of the code completion problem, the relevant information is a part of the source code file which precedes the position in the file where the prediction is to be inserted;

- **Which data is required for a specific task**: to obtain the information described in the previous requirement, a large amount of Python source code files needs to be gathered. The collected Python source code files should contain no duplicated data in order to have highly diversified training dataset. To capture various naming conventions and usage patterns of third-party libraries, the source code files should be obtained from different projects written by different developers;

- **Where can the data be found**: there are many code-hosting web-based services such as GitHub, BitBucket, GitLab and many more, which host thousands of open-source Python repositories.

- **Which format is the data in**: When cloning a code repository from one of the aforementioned web-based code-hosting services, you download an entire project folder. Apart from the source code files, the project folder often contains configuration files, auxiliary scripts, text files with setup instructions and many other files, which are irrelevant for our problem.

3.1.2 Data Collection

During the data acquisition phase, we have downloaded two datasets. The first dataset contains $1,000$ Python source code repositories collected from GitHub code-hosting web service. Section 3.1.2.1 describes the methodology and software tools we have applied to extract code repositories from GitHub. The second dataset contains $96,091$ Python package sources crawled from Python Package Index, known as PyPi. The detailed description of how we approached the crawling of PyPi data is presented in Section 3.1.2.2.

3.1.2.1 GitHub Data Collection

GitHub, the web-based development platform for collaborations on software projects, provides a public Application Programming Interface, API [gitb]. The GitHub API allows to browse and search hosted source code repositories using HTTP REST calls.
To extract $1,000$ most starred Python repositories, we have used the following API call:

$$api.github.com/search/\overset{\substack{\text{links to}\\ \text{repositories}\\ |}}{\boxed{repositories}}?q=\overset{\substack{\text{return only Python}\\ \text{repositories}\\ |}}{\boxed{language:python}}\ \&\ \overset{\substack{\text{most starred repositories}\\ \text{returned first}\\ |}}{\boxed{sort=stars\&order=desc}}$$

The response, returned by the API, is in JSON format and contains the information about requested repositories. The extract from a sample JSON response returned by the GitHub API is presented below:

```
{
    ...,
    "created_at": "2015-03-18T18:22:31Z",
    "updated_at": "2017-09-08T13:12:47Z",
    "pushed_at": "2017-09-06T20:59:08Z",
    "git_url": "git://github.com/google/yapf.git",
    "ssh_url": "git@github.com:google/yapf.git",
    "clone_url": "https://github.com/google/yapf.git",
    "svn_url": "https://github.com/google/yapf",
    "homepage": "",
    "size": 1481,
    "stargazers_count": 5483,
    "watchers_count": 5483,
    "language": "Python",
    ...
}
```

Listing 3.1: Extract from GitHub API JSON response

A GitHub repository can be downloaded by *cloning* it using it's *clone URL*. The *clone_url* of sample GitHub repository is highlighted on Listing 3.1.2.1 [gita].
We developed a simple utility script used to crawl GitHub repositories. In order to be able to clone Git repository on a local machine using *clone URL*, we have used third-party Python library, GitPython, which allows to interact with Git repositories from Python [gitc].
Despite the fact that GitHub is the one of the largest code-hosting web platforms providing convenient API to access hosted source code repositories, the API usage is subject to limitations. GitHub Search API returns at most $1,000$ search results for any API call, thus, significantly limiting the number of repositories that we could extract.

3.1.2.2 PyPi Data Collection

As presented in Section 3.1.2.1, we have managed to extract only $1,000$ source code repositories from GitHub due to GitHub Search API limitations. One of the research questions we are aiming to answer is how does the training dataset size affect the quality of method name suggestions. Therefore, alternative sources of data are required to prepare a larger dataset.
The Python Package Index, known as PyPi, is a repository of software for the Python programming language [pyp]. On $19/05/2017$, it featured $112,054$ Python packages. PyPi doesn't offer an API, however, it provides a simple search index which lists links to the archives of all hosted packages on a single web page. In order to download the Python packages available at PyPi search index, we have developed a custom web crawler based on Scrapy, an open-source web crawling framework [scra].
A *spider* is a class which defines custom crawling behavior. Scrapy uses user-defined spiders to decide which websites to scrape, which links to follow, how to scrape the data from the website and many more. The overview of the operation principle of the spider is shown below:

1. Scrapy makes an initial HTTP request to the PyPi simple search index URL defined by `start_urls` variable inside the spider. The default callback function when making an

initial HTTP request is *parse(self, response)*.

2. When a response from PyPi is received, *parse* callback function is invoked with *response* parameter representing the received HTTP response.

3. Our implementation of the *parse* function extracts values from HTML *href* attributes using *selectors* [scrb]. The HTML *href* attribute specifies the URL of the web page the link goes to. After collecting URLs listed on the PyPi search index web page, the spider makes a single HTTP request to each of the collected URLs setting *get_package_url(self, response)* function as a callback. Each of those URLs represent a link towards the web page which contains links for downloading different versions of the certain Python package.

4. *get_package_url* function extracts links only to those packages which are in `.zip`, `.tar.gz` or `.tar.bz2`. The package web page often contains auxiliary files, such as readme files or setup scripts, that are irrelevant to our research. Then, the link to the latest package version is identified and URL of that link is added to the `links.txt` file.

5. The output of the spider is `links.txt` file which stores a list of URL where each URL is pointing to the location of certain Python package archive.

6. To download the packages on our local machine using the list of links from the previous step, we use *wget* command-line tool utility

7. In the extracted dataset, we maintain the folder structure of the original project, however, only Python files with `.py` extension are kept.

We have managed to extract $96,091$ Python packages containing $1,534,528$ Python source files with `.py` extension. The total size of the dataset is 14GB.

3.1.3 Data Variable Selection

The *data variable selection* stage of the integrated data preparation scheme is concerned about selecting variables for modeling. The selection process will be discussed in detail in Section 3.2.

3.2 Preprocessing

In Section 3.1.1, we have stated that in order to train a neural network model capable of predicting accurate method name suggestions, we need the following information:

- **What to predict**: we are aiming to provide method name completion suggestions, therefore, the exhaustive list of method names available in Python core and third-party libraries is required.

- **How to predict**: relevant context information from the source code file should be used to make a prediction.

In this section, we will often refer to the position in a source code file where a method name suggestion would be inserted, if chosen. Therefore, we introduce the following definition:

Definition. *Completion position* is an index in a source code file where the method name completion suggestion, if chosen, would be inserted.

The first assumption that we make, when considering which data from the source code file is useful for purposes of predicting method name completion suggestions, is that the *relevant data* is always before the *completion position*. On Listing 3.3 the green box highlights the part of the source code file which we consider as *relevant*. The assumption may seem restrictive, but there are two

```
1 import os
2
3         def some_function():
4                 my_string = "Car"
5                 my_string.<completion_suggestion>
6                 my_string = "C:/Users/ThesisUser/"
7                 my_string = os.path.join(my_string,"MasterThesis")
```

Listing 3.3: An example code snippet

fundamental reasons behind it:

- **The code after the completion position might not be available**: the *completion position* in the code snippet on Listing 3.3 is on the line 5 and there are two other code lines below. Consider the scenario when a developer would be writing the presented piece of code and lines 6,7 wouldn't be there yet, however, the developer would expect a list of completion suggestions to be presented when writing the code in line 5.
- **The target variable type may change**: as demonstrated on Listing 3.3, the type of the variable `my_string` has changed on the line 6, right after the line where the code completion system was invoked. From now on, it is an irrelevant piece of information to the code completion assistant because the completion system was invoked before the type change has occurred.

This section presents three different source code preprocessors that extract relevant information for method name prediction from the datasets we have obtained using tools discussed in Section 3.1.2. The main goal of the proposed preprocessors is to answer two questions that we have previously set: *what to predict* and *how to predict*.

3.2.1 AST Visitor

Each of the source code preprocessors presented in the next sections performs two tasks. Firstly, every preprocessor extracts the context from the source code that is, according to the model implemented by the preprocessor, relevant for the method name prediction problem. Secondly, the preprocessor extracts names of the method calls that are used in the given source code file. While the realisation of the first task, namely, the extraction of the relevant context from the input file, is based on the preprocessor type, the implementation of the second task is shared across all the preprocessor types.
To acquire the list of method names used in the given source code file, a preprocessor builds an abstract syntax tree representation, AST, of the source code file. The concept of abstract syntax tree was discussed in Section 2.1.2. To build an AST representation of the source code file, the built-in Python `ast` library is used [pyt].
The `ast` module offers `NodeVisitor` class that, once an AST representation was built, walks the AST and calls a visitor function for every node found. We have subclassed the base `NodeVisitor` implementation and overriden `visit_Call` method which is invoked every time an AST node

of type `Call` is visited. Additionally, we introduced `call_nodes` class attribute that stores a triple for every node of type `Call` visited. The triple has the following format: `(caller_name, callee_name, ast_node)`. The `caller_name` is a piece of code, typically, a variable, which invoked the method call. The `callee_name` is the name of the method call which was invoked by the `caller_name`. The `ast_node` is an instance of the AST `Node` class of type `Call` where the method call invocation was found.

3.2.2 Basic Preprocessor

The underlying assumption behind the basic preprocessing model is that the part of the code line preceding the completion position is the only relevant piece of information for the method name prediction problem. Listing 3.4 illustrates presented assumption. Consider the developer just

```
import os

        def some_function():
                my_string = "C:/Users/ThesisUser/"
                os.path.<completion_suggestion>
```

Listing 3.4: Basic source code preprocessor model

typed `os.path.` and expects to receive a list of method name completion suggestions. The basic preprocessor extracts the part of the code line preceding the completion position as illustrated on Listing 3.4 and forwards it to the next processing step which is discussed in later sections.
The outline of the operation for the basic preprocessor is presented below:

1. The preprocessor takes a path to the directory as an input and scans a directory for Python files with `.py` extension.
2. For each of the acquired `.py` files, if any, the preprocessor builds an abstract syntax tree, AST, representation using Python built-in `ast` library.
3. The preprocessor uses `CallVisitor` class described in Section 3.2.1 to visit all nodes of type `Call` in the generated AST and to build the list of triples storing the information about the Call node.
4. Once the `CallVisitor` object finished iterating over the AST nodes, the acquired list of triples is passed onto the next function for further processing.
5. During the next processing stage, the list of triples is iterated over. At each iteration, the line number and the line of code corresponding to that line number, where the respective AST `Call` node was found, is obtained. Then, the part of the code line, which precedes the completion position, is extracted. Next, the extracted line is tokenized using NLTK `word_tokenize` function [nlt].
6. The output of the tokeniser, a list of tokens, is filtered removing `self` keyword in Python language indicating an instance variable. After the filtering step, the list of tokens is joined in a single line using space delimiter.
7. Finally, the line joined from the filtered list of tokens during previous step and the name of the respective method call are stored to the local data structures which are, once the input file finished processing, written into the output files on a disk.

3.2.3 Advanced Model Preprocessor

The advanced preprocessor is an evolution of the basic preprocessor that we discussed in Section 3.2.2. Apart from extracting the part of the code preceding the completion position, the advanced preprocessor also searches for all occurrences of the caller within a scope according to the *LEGB* rule. The only limitation of the advanced preprocessor is that it doesn't search for the caller occurrences at the global scope referenced as *G* in the *LEGB* rule. The *LEGB* rule and the concept of *scope* in the Python programming language were discussed in Section 2.1.1.
Listing 3.5 illustrates the operational principle of the advanced preprocessor. In order to be able to

```
import os

    def some_function():
        some_number = 5
        my_string = "Some content here"
        other_string = my_string.lower()
        my_string.<completion_suggestion>
```

Listing 3.5: An example of the advanced preprocessor operation

track the caller occurrences within it's scope, the advanced preprocessor implements two visitor classes, namely, `ClassNodeVisitor` and `FunctionNodeVisitor`.
The `ClassNodeVisitor` class, given an AST representation of a source code file, extracts and stores all AST nodes of type `ClassDef` which represent the definition of the class in AST notation. The `FunctionNodeVisitor` class keeps the track of AST nodes of type `FunctionDef`, that represent the function definition in AST syntax, in the given file, however, avoids entering an AST branch that is rooted at the node of type `ClassDef`. Thus, only the `FunctionDef` nodes outside of class definitions are tracked.

After discussing implementation details of the visitor classes used in the advanced preprocessor, the algorithm of the advanced preprocessor is presented:

1. The preprocessor takes a path to the directory as an input and scans a directory for Python files with `.py` extension.

2. For each of the acquired `.py` files, if any, the preprocessor builds an abstract syntax tree, AST, representation using Python built-in `ast` library.

3. The preprocessor uses an instance of the `ClassNodeVisitor` class to walk the AST in order to collect all AST nodes of type `ClassDef`.

4. Once `ClassNodeVisitor` finished walking the AST, the acquired list of AST `ClassDef` nodes is passed onto the next processing phase.

5. During the next processing stage, for every `ClassDef` node in the list collected by the `ClassNodeVisitor`, an instance of `CallVisitor` is created. The `CallVisitor` iterates over an AST rooted at the `ClassDef` node searching for `Call` nodes.

6. For every `Call` node collected during step 5, there a corresponding `(caller_name, callee_name, ast_node)` triple. The preprocessor script searches for all occurrences of the `caller_name` within enclosing `ClassDef` node. Every line containing the object referenced by the `caller_name` within `ClassDef` node is treated as relevant and added to the output file.

7. Once the relevant context from the class definition nodes is extracted, the list of `FunctionDef` nodes collected by the `FunctionNodeVisitor` goes into the processing pipeline. For every `FunctionDef` node, an instance of `CallVisitor` is created. The `CallVisitor` iterates over an AST rooted at the `FunctionDef` node looking for `Call` nodes.

8. This processing step is identical to the processing step 6. The only difference is that the enclosing, top-level node is `FunctionDef` node.

To gain a better understanding of the operation principle behind the advanced preprocessor script, the code snippet on Listing 3.6 is presented. The green box highlights the code that is processed by the `ClassNodeVisitor`. The red box indicates the code that is handled by the `FunctionNodeVisitor`.

```python
import os

class Foo(object):

        def __init__(self):
                self.foos = []

        def add_foo(self, foo):
                self.foos.append(foo)

class AnotherFoo(object):

        def _init_(self):
                self.description = ""

        def set_description(self, desc):
                self.description = desc

def foo():
        thesis_path = "C:/Foo/MasterThesis"
        source_code_path = thesis_path.lower()
        full_path = os.path.join(thesis_path, source_code_path)

def another_foo():
        list = [5,6]
        list.remove(5)
        list.append(5)
        list = list.extend([2,3])
```

Listing 3.6: The operational flow of the advanced preprocessor script

3.2.4 N-Chars Model Preprocessor

The N-chars preprocessor operates on the character level opposed to the preprocessors presented in Section 3.2.2 and Section 3.2.3. The reason for that is to introduce another level of granularity

for the neural network model. Consider variables named *my_list* and *his_list*. From the perspective of a word-level model, these variables are two different tokens. However, for a character-level model, these variables appear as *m y _ l i s t* and *o t h e r _ l i s t* input sentences. The character-level model would be able to detect that both input sequences share some portion of tokens and, as a consequence, when a new sequence like *n e v e r _ s e e n _ l i s t* would be encountered, the model would treat it as similar and is more likely to generate sensible outputs.
The N-chars preprocessor model assumes that the relevant context for the method name prediction problem is a collection of N or fewer characters before the completion position where N is an adjustable parameter. In our research, we set $N = 1000$. It is important to mention that the n-chars preprocessor counts every character before the completion position regardless whether it is a delimiter symbol, a user-defined comment or a newline character.
The code snippet on Listing 3.7 presents an example, which outlines the part of the source code file, the N-chars preprocessor treats as the relevant context for $N = 10$: As demonstrated on List-

```
import os

        def some_function():
        some_string = "blablabla"
        str = some_string + "xy"
        str.<completion_suggestion>
```

Listing 3.7: An example of the N-chars preprocessor operation

ing 3.7, the N-chars preprocessor extracts ten characters before the completion position: `'" x y " \n \t s t r .'` where `\n` and `\t` denote newline and tab characters respectively.
Key highlights of the N-chars preprocessor operation are presented below:

1. The preprocessor takes a path to the directory as an input and scans a directory for Python files with `.py` extension.

2. For each of the acquired `.py` files, if any, the preprocessor builds an AST representation using Python built-in `ast` library.

3. The preprocessor uses `CallVisitor` class described in Section 3.2.1 to visit all nodes of type `Call` in the generated AST and build the list of triples storing the information about found Call nodes.

4. For every `Call` node collected during step 3, there is a corresponding `(caller_name, callee_name, ast_node)` triple. Next, the preprocessor extracts N characters starting one index before the position, where the `callee_name` occurs in the source code file, moving towards the beginning of the file. If the amount of characters in the file before the `callee_name` is less than N, the preprocessor takes whatever amount is available. In addition to that, the preprocessor does not break source code tokens in between. Consider following scenario: the N-chars preprocessor with $N = 20$ has already acquired fifteen characters and the next source token is `string` which is six characters long. The preprocessor can foresee that by including all characters from the `string` token, the total number of acquired characters would exceed N. To avoid exceeding the limit, it would ignore the `string` token and stop execution collecting fifteen characters in total.

5. Once finished extracting N or less characters, the preprocessor joins the collected characters into the single string and replaces all occurrences of tab and newline characters by Unicode

symbols that are not recognized by the Python compiler. This is done in order to ensure that the Unicode symbol representing space or tab character does not occur in the dataset. Finally, every character in the resulting string is separated by space so that the output string, when appears in the output file, is a sequence of space separated characters.

3.2.5 Discussion

In this section, we presented three different models of the source code preprocessors that extract relevant information for the method name completion problem from the input Python program files. Each preprocessor makes different assumptions about which information from a given code file is relevant for the problem. Before looking at the results, we can hypothesize how the models are going to perform relative to each other.

The advanced preprocessor is very likely to outperform the basic preprocessor because it extracts additional information such as occurrences of the caller name within a scope. The extra contextual information about the caller variable can give strong hints about the type of the caller variable. However, given a piece of code like `os.path.<completion_suggestion>`, both models are expected to behave similarly.

The N-chars preprocessor, in majority of the cases, captures the same context as the advanced preprocessor, especially if N is set to a large value. In addition to that, the N-chars model acquires other pieces of the program file: their relevance for the method name prediction problem is questionable. On one hand, the N-chars model could capture user-defined comments and declarations of other variables that might be completely unrelated for the method name to be predicted. On the other hand, the model is likely to acquire file imports, the name of the enclosing function and other parts of the source code that may provide strong cues which method name to predict.

Since the N-chars model is provided with plain source code and does not receive any preprocessing hints, we expect it to perform worse than the advanced model, but it may be that the hidden layers of a neural model are able to capture all relevant clues on their own.

3.3 Training

This section presents an overview of available neural machine translation kits and describes the procedure we used to train a neural network model capable of predicting method name suggestions.

3.3.1 Overview of Software Tools

In Section 2.2.3.3, we covered state-of-art techniques in a field of neural machine translation, NMT, and stated that the problem of predicting method name completions can be seen as a sequence-to-word translation task.
There are many open-source NMT toolkits such as *seq2seq*, *GroundHog* that were developed for research purposes [gro] [seq]. These toolkits support particular research projects and are not designed for public use offering little or no documentation which makes them difficult to adopt. However, there are two open-source NMT implementations that are fairly mature and have a solid documentation.
The NMT toolkit, called *Nematus*, was proposed by the Natural Language Processing Group at

the University of Edinburgh [SFC+17]. Nematus supports state-of-art neural network architectures that have been proposed in the field of NMT such as the attention-based model and the encoder-decoder architecture that we discussed in Section 2.2.3.3. In addition to that, Nematus offers large number of options and configuration parameters for training a neural network. These options include choosing the type of hidden layer unit, LSTM or GRU, setting the number and the size of neural network layers, multi-GPU training and many others.
Alternative solution is an Open Machine Translation toolkit developed by the Harvard Natural Language Processing group with a support of SYSTRAN. Similar to Nematus, OpenNMT supports state-of-art neural network architectures and offers an extensive list of configuration options. On top of that, OpenNMT has an extensive documentation and an active user community. There is also an official OpenNMT forum where OpenNMT developers and maintainers discuss open issues with the community on the regular basis. Furthermore, OpenNMT offers various utility tools, such as C++ Translate and REST Translation Server, that allow to integrate trained neural network models into the existing product. The technical report describing OpenNMT features can be found at [KKD+].
In our research project, we used OpenNMT toolkit mainly because it offers mature documentation and provides utility tools for model deployment.

3.3.2 Training Process

The process of training a neural network model using OpenNMT toolkit is two-stage.
First part, the preprocessing stage, is concerned with building the word and feature vocabularies from the given input files. Each word in the vocabulary file is assigned an index which is used later during the training phase. The input to the OpenNMT preprocessor are two file pairs, source and target training files and source and target evaluation files, where files within a pair must be aligned with each other. The OpenNMT framework uses evaluation files to detect convergence of training. In our case, the source file, produced by one of the preprocessor scripts described in Section 3.2, contains the information which we assume is relevant for the method name prediction problem. The target file contains a list of corresponding method names. To generate a pair of evaluation files, we developed a little utility script that, given a pair of files (source and target) and number of lines to include in evaluation files, generates source and target evaluation files. To ensure the data in the evaluation files is uniformly distributed, the utility script uses randomly generated indexes to select which lines from the input files should go into the evaluation files.
Extracts from aligned source and target files generated by the advanced preprocessor are presented on Listing 3.8 and Listing 3.9 respectively. The OpenNMT framework offers a vast num-

```
lines = [] lines
string = "Car" split_string = string
file = open ( "f.txt", "r" ) lines = file
```

Listing 3.8: An extract from sample source file

```
append
split
readlines
```

Listing 3.9: An extract from sample target file

ber of configuration options to parametrize a neural network architecture. These options include different encoder and decoder types, various types of hidden layer units. In our work, we chose to use a unidirectional RNN encoder implementation and a decoder that uses an attention model which are default parameters proposed by the OpenNMT toolkit.
In Section 3.1.2.1 and Section 3.1.2.2, we explain how we trained a neural network model on each of the collected datasets, namely, GitHub dataset and PyPi dataset. The approach that we used to

collect these datasets is described in Section 3.1.

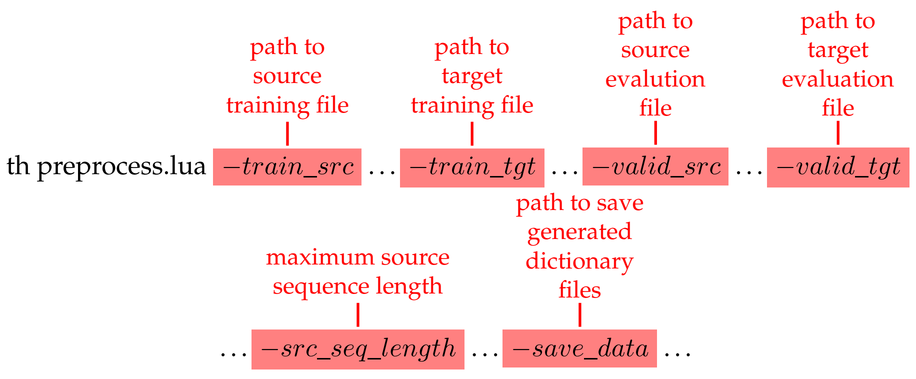
3.3.2.1 Training on GitHub dataset

The data collected from the GitHub code-hosting web service contains $1,000$ Python repositories that include $157,772$ Python source code files in total. We have processed the data with each of the preprocessor implementations presented in Section 3.2 and obtained three different training datasets.

Type of preprocessor	Size of source training file (in MB)
Basic	51.8
Advanced	$1,931$
N-chars	$9,917$

Table 3.1: GitHub dataset characteristics

From Table 3.1 we observe that the amount of information which is treated as a relevant context significantly differs for every preprocessor model.
The next step is to use a preprocessor built into the OpenNMT toolkit in order to obtain source and target dictionary files. We used the OpenNMT preprocessor with the following parameters:



```
th preprocess.lua -train_src ... -train_tgt ... -valid_src ... -valid_tgt
... -src_seq_length ... -save_data ...
```

path to source training file
path to target training file
path to source evalution file
path to target evaluation file
maximum source sequence length
path to save generated dictionary files

Most of the preprocessor parameters are paths to file system locations where training and evaluation files are stored. Unlike others, the `src_seq_length` configuration option represents the maximum allowed length of an input sequence with a default value set to 50. If a sequence contains more words than permitted by `src_seq_length`, the preprocessor does not include it into the training dataset. When preprocessing the data files generated by the N-chars preprocessor script, the most of the input sequences from these files were ignored. This is because each input sequence is a bag of $1,000$ or less characters which the OpenNMT preprocessor treats as separate tokens, ignoring all sequences that have more than 50 characters. To avoid that, we set `src_seq_length=` 1000 when processing data files generated by the N-chars preprocessor.
Once OpenNMT finishes preprocessing stage, the next step is training. To train a neural network model, we used the OpenNMT `train.lua` script with following parameters:

$$\text{th train.lua } \overset{\text{path to the training data}}{-data}\ \overset{\text{number of layers}}{-layers} \ldots \overset{\text{number of neurons per layer}}{-rnn\_size} \ldots \overset{\text{which GPU(s) to use for training}}{-gpuid} \ldots \overset{\text{path to save trained model}}{-save\_model} \ldots$$

We set neural network parameters, namely, `layers` and `rnn_size`, to 2 and 600 respectively. The research literature suggests that deep neural network models i.e. neural networks having multiple hidden layers apart from input and output layers, outperform two-layer networks [SVL14]. Unfortunately, we are bound by the hardware limitations when choosing neural network parameters. The amount of memory on the graphical processing unit, GPU, provided for our project is too small to fit larger neural network into the memory.

3.3.3 Training on PyPi dataset

We managed to download $96,091$ Python repositories that contain $1,534,528$ Python source code files in total using the crawler discussed in Section 3.1.2.2. A training dataset was prepared by every preprocessor model that we presented in Section 3.2. As a result, three training were generated. The figures reported in Table 3.1 and Table 3.2 follow similar trend: the simple preprocessor

Type of preprocessor	Size of source training file (in MB)
Basic	402
Advanced	$10,582$
N-chars	$68,200$

Table 3.2: PyPi dataset characteristics

extracts very little context from the given code files compared to the N-chars preprocessor.
Due to hardware limitations, we were unable to carry out the training process using the dataset collected from PyPi. The reason is that parameters representing dimensionality of a neural network could not be increased. A two-layered neural network with each layer consisting of 600 neurons that we used to train on the datasets generated from GitHub data is insufficient to capture properties of this dataset.

3.4 Code Completion Plugin

This section describes the implementation details of the code completion plugin that uses a neural network model trained according to the approach described in Section 3.3.

3.4.1 Overview of Integrated Development Environments

There are many integrated development environments, IDEs, on the market, both free and paid, that support Python programming language. For example, *PyCharm* is a popular commercial Python IDE by JetBrains offering a lot of productivity features such as intelligent code completion, refactoring assistant and others [pyc]. *Eric*, *Wing*, *Netbeans* are all examples of free, open-source

integrated development environments that also offer an extensive list of productivity features including code completion assistant [eri] [win] [net].
The choice of the IDE, for which we were to develop a code completion plugin, was dictated by the presence of API functionality that is necessary to detect when the code completion plugin should be invoked. In most IDEs, a code completion feature is built into the core libraries, making it impossible to provide the desired API functionality. The only software development environment that offers this functionality is NetBeans IDE. Therefore, NetBeans platform served us as a basis for a code completion plugin implementation. To implement a module for NetBeans platform, the Java programming language is to used.

3.4.2 Implementation

3.4.2.1 NetBeans Completion API

The NetBeans integrated development environment offers an `Editor Code Completion API` allowing to integrate custom code completion assistants into the IDE [edi]. There are two interfaces, `CompletionItem` and `CompletionProvider`, that must be implemented by the software component aiming to provide code completion suggestions.
`CompletionProvider` interface offers following methods:

– `createTask`: when user requests a list of completion suggestions, the corresponding task is created and executed. During the task execution, a list of instances of classes implementing the `CompletionItem` interface should be generated where each instance represents a single completion suggestion. The task can be synchronous or asynchronous.

– `getAutoQueryTypes`: determines whether code completion window is brought up automatically or by user action.

An instance of class conforming to the `CompletionItem` interface represents single completion suggestion. Highlight of key methods available in `CompletionItem` interface is shown below:

– `defaultAction`: specifies the action to execute when user presses Enter key or double-clicks mouse cursor on the item

– `getSortPriority`: returns the priority of the item. The higher the priority, the closer to the top of suggestion list the item appears.

– `getSortText`: returns a text which used when sorting items alphabetically.

– `getInsertPrefix`: returns a text used for finding a longest common prefix

– `render`: render the item into graphics context

– `getPreferredWidth`: returns visual width of the item

The flow chart diagram on Figure 3.1 demonstrates the typical usage scenario of `Editor Completion API`.

3.4.2.2 Neural Network Model Integration

To interface with the neural network model that translates relevant context from the source code file into method name suggestions, we used an utility tool provided in the OpenNMT toolkit called *REST Translation Server*.
A HTTP request is sent to a translation server running on fixed IP address and port specifying

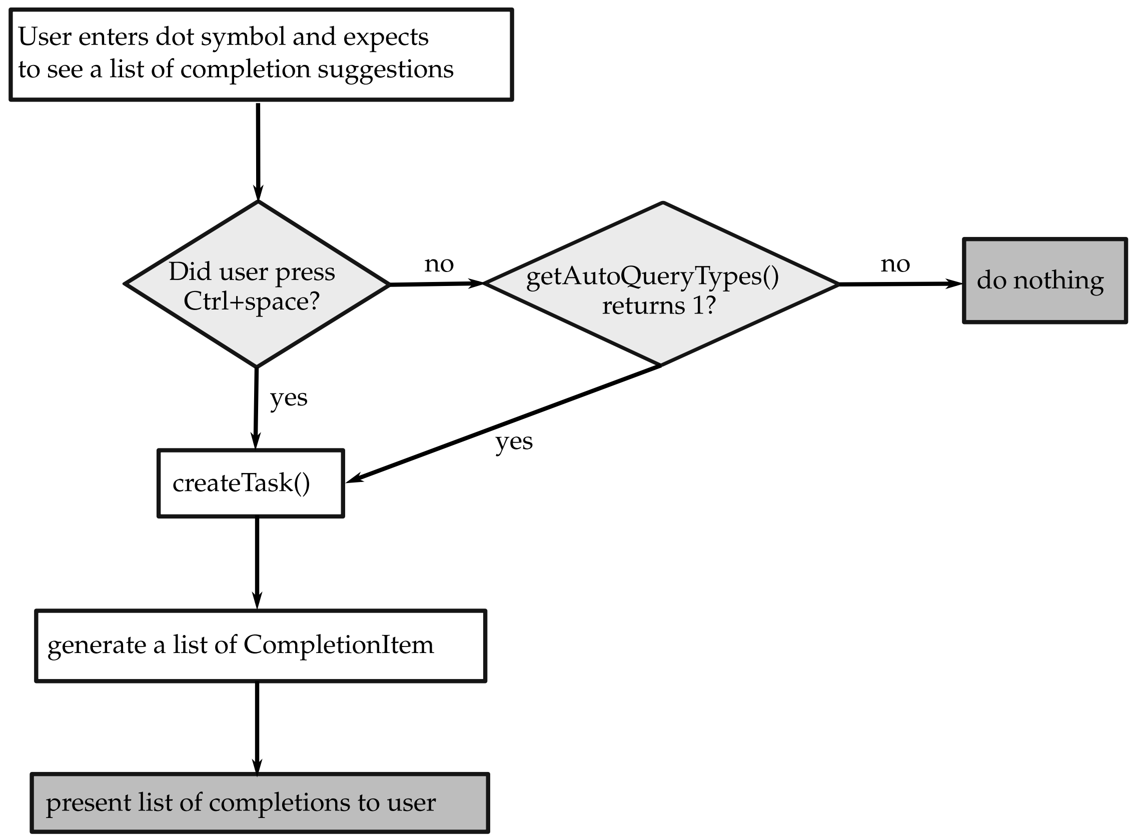

User enters dot symbol and expects to see a list of completion suggestions
Did user press Ctrl+space?
no
getAutoQueryTypes() returns 1?
no
do nothing
yes
yes
createTask()
generate a list of CompletionItem
present list of completions to user

Figure 3.1: Typical usage scenario of `Editor Completion API`

what is the data to be translated. The translation server responds with a JSON containing translation results. The principle of operation of the OpenNMT translation server is demonstrated on Figure 3.2.

3.4.2.3 Architecture

In this section, we present an overall outline of the code completion plugin operation.
`PythonCompletionProvider` is a class which implements `CompletionProvider` interface and interconnects system components such as module responsible for communication with the OpenNMT translation server or the module which extracts relevant information from the source code file to be later sent for translation.
The implementation of `createTask` method is presented below: Note that request to the translation server is not sent every time `createTask` method is called. Suppose a developer invoked the completion assistant on an object and, once a list of suggestions is presented, he types `'a'` expecting to see only suggestions beginning from `'a'`. The NetBeans engine would invoke `createTask` method twice: once when the developer initially invoked the code completion assistant and once when the developer typed `'a'` expecting to see only suggestions beginning with `'a'`. To avoid making duplicate translation requests, we introduced intermediate cache for storing translation results. When a completion assistant is initially invoked on an object, the translation request is sent to the REST server, returned results are stored in the cache. Later, if a filtered

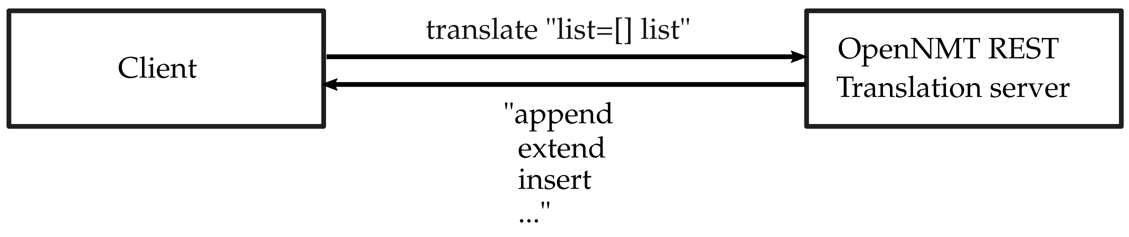


Figure 3.2: OpenNMT REST Translation Server

version of method name list is needed, the suggestions matching the filter criteria are obtained from the cache.

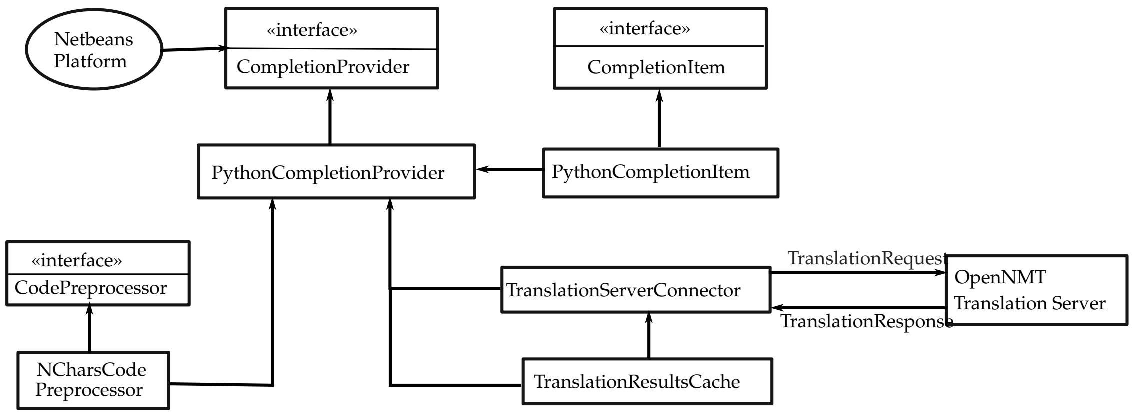


Figure 3.3: Class diagram of the code completion assistant

```java
@Override
public CompletionTask createTask(int queryType, JTextComponent jtc) {
   if (queryType != CompletionProvider.COMPLETION_QUERY_TYPE) {
         return null;
   }

   return new AsyncCompletionTask(new AsyncCompletionQuery() {

      @Override
      protected void query(CompletionResultSet resultSet, Document doc, int caretOffset){
         try {
            List<TranslationResult> translationResults;
            String relevantContext = doc.getText(0, caretOffset);
            int dotIndex = relevantContext.lastIndexOf(".");
            if (doc.getText(caretOffset-1, 1).compareTo(".") == 0) {
               String processedCode = preprocessor.processCode(relevantCode);
               translationResults = serverConnector.translate(
               new TranslationRequest(processedCode));
            } else {
               String prefix = doc.getText(dotIndex+1, caretOffset - dotIndex - 1);
               translationResults = cache.getStartsWith(prefix, 0);
            }
            for (TranslationResult translationResult: translationResults) {
               String suggestion = translationResult.getMethodNameSuggestion();
               /* filter out <unk> results */
               if (suggestion.compareTo("<unk>") != 0) {
                  resultSet.addItem(
                  new PythonCompletionItem(translationResult, dotIndex));
               }
           }
         } catch (BadLocationException ble) {
            Logger.getLogger(PythonCompletionProvider.class.getName()).
            log(Level.SEVERE, null, ble);
         }
         resultSet.finish();
      }
   }, jtc);
}
```

Listing 3.10: The `createTask` method implementation

Chapter 4

Evaluation

In this chapter, we perform a quantative and qualitative comparison of neural network models. They were trained using the datasets produced by the preprocessor implementations discussed in Section 3.2. In addition to that, we evaluate the code completion plugin presented in Section 3.4 against state-of-art code completion engines.

4.1 Evaluation Approach

The evaluation process is divided up into three stages.
Firstly, we compare how neural network models, trained on different datasets, perform against each other during the training process using the *perplexity* evaluation metric. Secondly, we evaluate the neural network models based on the relevance of completion suggestions they generate. Apart from using model assessment based on theoretical evaluation metric such as perplexity, it is important to evaluate how code completion models perform on real-world test cases. Since each model makes different assumptions about code parts that are relevant for the method name suggestion task, we hypothesize that every model will exhibit distinct behaviour when applied on a real-world problem. To evaluate proposed the code completion engine on actual data, we propose four constraining criteria:

– **Amount of relevant context**: we would like to analyze how much code i.e. *relevant context*, every model needs in order to make sensible completion suggestions. If little context is needed, the model should be capable of suggesting suitable method names even when the program file has only couple of code lines.

– **Proximity of relevant context**: this criterion is concerned with the distance from a completion position to the code parts which provide information about the method name to be predicted. While this criteria is not helpful to assess the performance of the simple preprocessor model which uses only the part of a code line before the completion position, the N-chars preprocessor model, which considers N or fewer characters before the completion position, is highly dependent on the distance of the relevant content within the code file relative to the completion position.

– **Variable naming**: this criterion is used to assess how much does an inadequate variable naming affect the quality of method name suggestions. For example, variable with a name `string` referencing an object of type `list` may confuse a context-based completion engine.

– **Prevalence of the variable type**: variables of built-in Python types such as `string` or `list` occur within our training datasets much more frequently compared to variable types that are introduced by third-party libraries. Therefore, we would like to estimate the effectiveness of our model in predicting method names for infrequent variable types.

To assess described criteria, we prepared twenty test cases which are code snippets extracted from actual Python software projets. Listing 4.1 demonstrates one of the evaluation cases which is designed to test the quality of method name suggestions on a varible type from `BaseHTTPServer` library given fair amount of context and adequate variable naming. With the use of above-

```python
import time
import BaseHTTPServer

HOST_NAME = 'example.net' # !!!REMEMBER TO CHANGE THIS!!!
PORT_NUMBER = 80 # Maybe set this to 9000.

class MyHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    def do_HEAD(s):
        s.send_response(200)
        s.send_header("Content-type", "text/html")
        s.end_headers()
    def do_GET(s):
        """Respond to a GET request."""
        s.send_response(200)
        s.send_header("Content-type", "text/html")
        s.end_headers()
        s.wfile.write("<html><head><title>Title goes here.</title></head>")
        s.wfile.write("<body><p>This is a test.</p>")
        # If someone went to "http://something.somewhere.net/foo/bar/",
        # then s.path equals "/foo/bar/".
        s.wfile.write("<p>You accessed path: %s</p>" % s.path)
        s.wfile.write("</body></html>")

if __name__ == '__main__':
    server_class = BaseHTTPServer.HTTPServer
    httpd = server_class((HOST_NAME, PORT_NUMBER), MyHandler)
    print time.asctime(), "Server Starts - %s:%s" % (HOST_NAME, PORT_NUMBER)
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        pass
    httpd.server_close() <completion>
```

Listing 4.1: An example of evaluation test case (`<completion>` marks the method name expected to be predicted)

mentioned test cases, we compare the proposed approach for predicting method names with state-of-art techiques that are currently applied in modern IDEs. The detailed procedure and obtained results are presented in Section 4.3.4.
Furthermore, we demonstrate that our approach can be generalized to other dynamic progamming languages by training a method completion model on the Javascript, another widely used programming language with dynamic type system.

Finally, we discuss threats to validity of our experiments in Section 4.5.

4.2 Model Performance

In this section, we discuss how each of the models performed during the training process.

4.2.1 Perplexity metric

To measure how models perform during the training process, we apply *perplexity* evaluation metric. The *perplexity* is a way to evaluate a language model which can be explained as an *average number of choices per word*. The higher the perplexity value, the more choices a language model has when predicting the next word.
The OpenNMT framework that we use for training neural network models calculates *perplexity* as follows:

$$\texttt{perplexity} = e^{-\frac{1}{N}\sum_{i=1}^{N}\ln q(x_i)}$$

where N is the total number of test samples and $q(x_i)$ represents how well the probability model q predicts a test sample x_i. When the model q predicts samples with the highest level of confidence i.e. $q(x_i) = 1\ \forall i$, the value of perplexity is 1. In contrast, when confidence of the model predicting samples is at the lowest point i.e. $q(x_i) = 0\ \forall i$, the value of perplexity is $+\infty$.
The OpenNMT toolkit trains a neural network model over a number of epochs reporting the value of perplexity after every epoch. The perplexity of the model is calculated using a validation dataset that contains previously unseen data.

4.2.2 Results

We trained each of our models for thirteen epochs capturing the value of validation perplexity after every epoch for each model. The results presented on Figure 4.1 demonstrate how perplexity

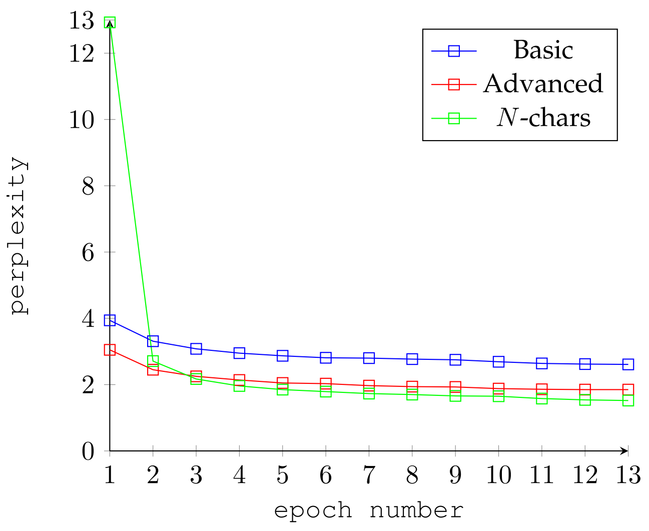


Figure 4.1: Perplexity for each model during the training

value has evolved. The basic model has a nearly flat learning curve starting with perplexity value of 3.94 at epoch 1 and finishing with the value of 2.61 at epoch 13. We hypothesize that the basic model exhibits slow learning pace because it uses too little information for method name prediction. As we discussed in Section 3.2.5, the advanced model is expected to outperform the basic model and the results confirm our prediction. The advanced model demonstrates final value of perplexity of 1.85 at epoch 13. Despite the poor performance in the beginning, the N-chars model was capable of achieving final perplexity of 1.52 at epoch 13, which is the lowest value across all the models.

Therefore, it appears that the model which requires least amount of processing i.e. N-chars demonstrates best performance.

4.3 Qualitative evaluation

To analyze performance of the models when applied on the actual problem and assess how it aligns with the theoretical results, this section presents a qualitative evaluation of the models on twenty real-world test cases.

4.3.1 Evaluation Metric

To quantatively assess the proposed method for suggesting completions, we use an accuracy metric that is commonly applied in the field of code completion [RVY14] [RL08] [NHC+16]. If a target method name is within top N suggestions where N is a variable parameter, we say that a completion engine fulfiled the task and give it a score of one. The target name that appears in the list of suggestions but isn't in top N proposals is considered irrelevant and the code completion assistant receives a score of zero. The final score is calculated as a sum of scores on individual test cases and used as a basis for calculating the accuracy metric. For example, given $N = 10$, a code completion assistant included the target method name in top ten suggestions five times when evaluated on twenty test cases. The system achieved a score of five and the accuracy of the system is $\frac{5}{20} = 25\%$.
A second type of assessment technique that we use is a *precision* performance metric. *Precision* is the ratio of relevant elements to the total number of presented elements. This metric is rarely used to evaluate code completion engines. The reason is that the majority of code completion assistants are designed for statically typed programming languages as we discussed in Section 1.2. In programming languages with static type system, the information about variable type is available as code is being edited, thus, the complete list of methods for the given object type is accessible at any moment. In contrast, the type information in dynamically typed programming languages is available only at runtime. Therefore, our code completion engine might misinterpret the context and propose irrelevant suggestions as a consequence of being unable to access the information about object types.
We calculate the value of precision metric as follows: for each test case, the value of precision is a ratio of the number of suggested methods that can be used on a given variable type to the total size of suggestions list. To assess overall precision for a given set of test cases, mean value of individual results is computed.

4.3.2 Comparison of Preprocessor Models on Actual Code Examples

Even though the N-chars model has proven to be superior to other models from theoretical prospective, we perform a quantitative comparison of the models on real-world test cases. The reason is that we believe that the characteristics of the context i.e. source code file, highly affect the quality of results predicted by the model.
We prepared twenty test cases which are actual code extracts from real Python repositories that did not appear neither in the training dataset nor in the validation dataset. These test cases were carefully picked in order to test how characteristics of the source code, presented in Section 4.1, affect the method name suggestions. There are four characteristics that we aim to test, namely, *amount of relevant context*, *proximity of relevant context*, *variable naming* and *prevalence of the variable type*. For each characteristic, there are two possible values, for example, *variable naming* = {adequate, inadequate} or *prelevance of variable type* = {often used, rarely used}. Every combination of characteristics and their values has a corresponding test case resulting in sixteen test cases in total. In addition to that, we have prepared another four code snippets which were designed to assess our models in edge case situations. A replication package containing all test cases can be found at `http://tiny.uzh.ch/Kg`.
We use two evaluation metrics to quantatively assess the performance of the models, namely, a target method name in *top-N* suggestions and *precision*, described in Section 4.3.1. The target method name in *top-N* suggestions metric is used with three different values of N: 3, 5 and 10.

Table 4.1 shows evaluation results of our models on sixteen test cases described above:

	top-3(%)	top-5(%)	top-10(%)	precision(%)
Basic	50.0	56.25	62.5	55.0
Advanced	56.25	62.5	68.75	63.75
N-chars	62.5	75.0	81.25	63.75

Table 4.1: Evaluation results of our models on sixteen real-world test cases

The actual figures presented in Table 4.1 partially confim the hypothesis that we stated in Section 3.2.5 and align with the theoretical results demonstrated in Section 4.2.2: the advanced model outperforms the simple processor strategy.

Interestingly, it appears that the N-chars preprocessor strategy, which we expected to demonstrate rather average performance, reported the lowest perplexity value. Since the input to the model is just a plain source code which may contain comments or empty lines, the model has to learn how to weigh an input i.e. which parts of the input sequence are relevant for the prediction and which should be discarded.

During the testing procedure, we observed that the basic model is very sensitive to the inadequate variable naming, however, on a source code file with informative variable names, the quality of method name suggestions is in line with the advanced model.

The advanced model demonstrates superior performance when the distance between the completion position and the relevant context is large. Yet, it cannot always interpret the context surrounding the completion position correctly and predict the method name appropriate for a given situation. The model often offers a target method name in the lower part of the suggestion list. The possible reason is that the model has enough knowledge about the caller variable in order to predict the list of correct method names, however, due to the inability to assess the nearby context, the target method name often appears at the bottom of the suggestion list and, therefore, the model receives a score of 0.

The N-chars model demonstrated strong positive performance across the most of the test cases. By analyzing the context above the completion position, the model can often correctly predict not only the list of method suggestions, but also the order of presentation. However, the main limitation of the model appears to be the size of N i.e. how many characters does it look behind. Often, variable declarations are at the top of a function body, whilst method invocations on those variables are at the bottom of the function. In a relatively long program file, the N-chars model with $N = 1,000$ would be unable to lookbehind enough to capture the relevant context resulting in irrelevant method name suggestions.

In addition to the described above, we noticed that all models perform significantly better when predicting completion suggestions on built-in Python types and modules. This phenomenon is clearly related to the popularity of built-in libraries that can solve the most of common tasks.

4.3.3 Comparison of Preprocessor Models on Confusing Code Examples

We also assess the behaviour of the proposed approaches on a set of test cases that were specifically designed to confuse our models.

4.3.3.0.1 Import Renaming Consider the import statement at the top of Listing 4.2 which demonstrates the usage of a syntatic feature in Python called `aliasing` which allows to refer to imported modules using any name. Interestingly, this specific case uses `sys`, which is a widely

```
import os as sys

def get_correct_path(path_extension):
        path_1 = "C:/MasterThesis/project"
        path_2 = "project/scripts/"
        return sys.path.join(path_1, path_2) <completion>
```

Listing 4.2: Special test case 1 (`<completion>` marks the method name expected to be predicted)

used module to interact with the parameters of Python interpreter, as an alias for `os`, another popular Python library. Table 4.2 demonstrates the performance of our models in described scenario:

	top-3(%)	top-5(%)	top-10(%)	precision(%)
Basic	100.0	100.0	100.0	50.0
Advanced	100.0	100.0	100.0	40.0
N-chars	0.0	0.0	0.0	10.0

Table 4.2: Special test cases evaluation results 1

Despite showing a strong performance on the conventional test cases, the N-chars model have failed to propose meaningful suggestions in this scenario. For some reason, it interpreted from the context, that the type of *sys.path* is *list*, and proposed corresponding method names. The 10% accuracy value is a lucky co-incidence: *os* and *list* share methods with same names e.g. *remove*. The simple model was able to figure out that *sys* is an alias and proposed *join* as a top suggestion. The advanced model, despite reporting nearly same score as the basic model, was actually confused by the example and included in the prediction method names for three types, namely, *os*, *list* and *str* luckily including *join* in top-3 methods.

4.3.3.0.2 Change of Variable Type The example program presented on Listing 4.3 is aimed to test whether our models weighs the relevant context based on proximity from the completion position. The evaluation results are presented in Table 4.3:

```
my_list = [4]
my_list.append(5)
my_list.extend(1)
my_list.remove(4)
print("List length: {0}".format(len(list)))
my_list = "CAPS LOCKED SENTENCE"
my_list.lower() <completion>
```

Listing 4.3: Special test case 2 (`<completion>` marks the method name expected to be predicted)

	top-3(%)	top-5(%)	top-10(%)	precision(%)
Basic	0.0	0.0	0.0	50.0
Advanced	0.0	0.0	0.0	20.0
N-chars	0.0	0.0	0.0	0.0

Table 4.3: Special test cases evaluation results 2

None of the models managed to predict the target method name even within top 10 suggestions. Clearly, the models do not put more importance on those parts of relevant context that are closer to the completion position. The attention-based model described in Section 2.2.3.3 could be a potential remedy. We can specify which parts of the input sequence carry more importance, therefore, the model will learn that the most recent use of the variable is the most crucial.

4.3.3.0.3 Instance Variables With the next test case presented on Listing 4.4, we aim to evaluate the quality of method name suggestions for instance variables within a class definition. The test case was designed in such a way that the variable name i.e. `my_foos`, does not provide any hints about the type of underlying object. Table 4.4 presents evaluation results on the test case

```
class Foo(object):

        TOO_MANY_FOOS = 10

        def __init__(self):
                self.my_foos = []
                self.description = ""

        def add_foo(foo):
                if "nice" in foo.description:
                        if len(self.my_foos) > TOO_MANY_FOOS:
                                pass
                        else:
                                self.my_foos.append(foo) <completion>
```

Listing 4.4: Special test case 3 (`<completion>` marks the method name expected to be predicted)

described above:

	top-3(%)	top-5(%)	top-10(%)	precision(%)
Basic	100.0	100.0	100.0	30.0
Advanced	100.0	100.0	100.0	90.0
N-chars	100.0	100.0	100.0	60.0

Table 4.4: Special test cases evaluation results 3

The figures demonstrated in Table 4.4 speak for themselves: all three models were able to recognize the type of *my_foos* instance variable and suggest relevant method name completions. However, we cannot confidently claim that these results will generalize to other code completion scenarios involving instance variables: the presented example is elementary and was included in order to demonstrate that our models can also generate method name predictions for instance variables.

4.3.3.0.4 Inadequate Variable Naming The final test case presented on Listing 4.5 is designed to assess how our models deal with an extreme case of inadequate variable naming. This example demonstrates two variables, *my_list* and *other_list*. Each of them represents an underlying object of type *set*. Variable names were picked in such a way that they give our models false hints about their types. Furthermore, *list* and *set* built-in types share many method names in common such as *pop* and *remove*.
Results are presented in Table 4.5:

```
my_list = {1,2,3}
print("Length of list: {0}".format(len(my_list)))
my_list.remove(2)
my_list.pop()
print("Length of list: {0}".format(len(my_list)))
other_list = {3,4}
is_contained = my_list.issubset(other_list) <completion>
```

Listing 4.5: Special test case 4 (`<completion>` marks the method name expected to be predicted)

	top-3(%)	top-5(%)	top-10(%)	precision(%)
Basic	0.0	0.0	0.0	10.0
Advanced	0.0	0.0	0.0	50.0
N-chars	0.0	0.0	0.0	50.0

Table 4.5: Special test cases evaluation results 4

Reported precision values show that the advanced and N-char models actually captured that there was a variable of type *set* within a program file. In fact, the list of suggesstions generated by both models consists of method names applicable to *list* and *set* object types only. However, the target method name, *issubset*, was not in top-N suggestions for any of the models. The most likely reason is that *issubset* did not appear repeatedly within the training dataset. Therefore, it is not likely to be amongst top predictions.

4.3.4 Evaluation against existing state-of-art tools

In this section, we evaluate the N-chars model, which was shown to outperform other presented models, against industry-leading Python code completion assistants.
PyCharm is a commercial Python IDE developed by JetBrains which offers intelligent code completion system [pyc]. Jedi, a state-of-art code completion and static code analysis library for Python programming languages, has been integrated in many popular text editors, such as Vim, Sublime Text or Atom [jed]. Finally, we evaluate our system against the Kite, cloud-powered software productivity tool for Python which offers smart code completion amongst other features [kit]. We apply the same evaluation approach and performance metrics as demonstrated in Section 4.3.2.

We perform evaluation on the general test set which consists of sixteen test cases. The analysis of four special test cases is omitted because they were specifically designed to demonstrate on which real-world cases neural networks may struggle.
Table 4.6 presents evaluation results of the N-chars code completion model against PyCharm, Jedi and Kite on sixteen different test cases.

	top-3(%)	top-5(%)	top-10(%)	precision(%)
Our model	62.5	75.0	81.25	63.75
PyCharm	25.0	31.25	31.25	54.375
Jedi	18.75	25.0	31.25	65.0
Kite	37.5	37.5	56.25	87.5

Table 4.6: Evaluating our approach against industry-leading code completion engines on general test cases

The results presented in Table 4.6 clearly demonstrate that our model is considerably superior to other presented code completion assistants. During the testing we observed that the analyzed code completion engines, producing the list of alphabetically ordered suggestions, included the target method name in the bottom part of the list. As the result in spite of being among suggested options, target method name, was not taken into account for the top-N metric value calculation in such cases. Another tendency that we spotted during the testing process is that analyzed completion engines often do not generate any method name suggestions at all. For example, one of the test cases involved predicting a method name on the variable of built-in Python type *dict*. All of analyzed completion engines failed to generate a suggestion list. However, when these code completion engines do generate suggestions, they often have a precision value of 100%.
We have demonstrated that our system has the potential to outperform existing industry leading code completion engines. Even though the evaluation dataset was rather small and potentially not representative (see Section 4.5 for further discussion), we are confident in the approach, since the perplexity values on the entire validation data indicate high performance. Furthermore, we demonstrate in next section that our model can be generalized to other dynamically typed programming languages such as Javascript. In contrast, discussed code completion engines rely on static language analysis tools and, therefore, cannot be easily adopted for application to other programming languages.

4.4 Generalizability of Approach

The industry-leading completion assistants that were discussed in Section 4.3.4 heavily rely on static code analysis tools when suggesting method name completions. An implementation of these tools for a given programming language is specific to the semantic and syntactic structure of the language. Therefore, discussed code completion engines cannot be easily adapted to support other dynamic programming languages.
We hypothesize that the proposed approach to the method name completion problem can be easily generalized to other programming languages. To test our hypothesis, we apply the approach presented in Chapter 3 to train a code completion model on $1,000$ Javascript software repositories collected from GitHub. To prepare the corpus for training, we adapted the N-chars preprocessor script so that the preprocessor knows how to generate an AST for a given Javascript program.

The final value of perplexity reported at the end of the training is aligned with results presented in Section 4.2. Figure 4.2 demonstrates how perplexity evolved:

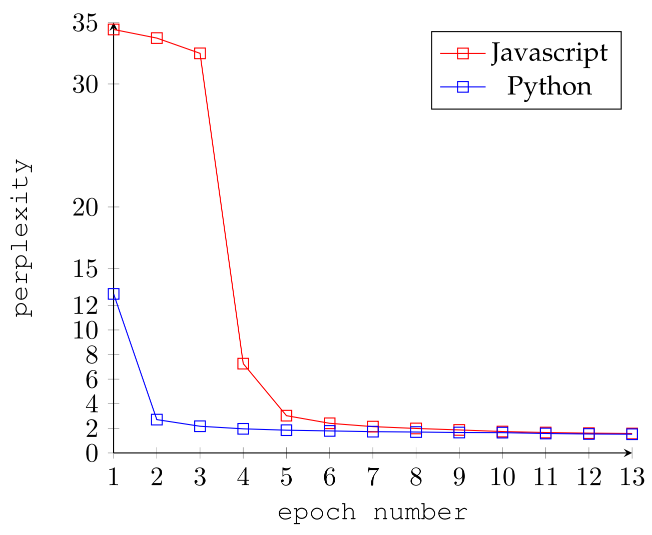


Figure 4.2: Perplexity graph when training N-chars model on Python and Javascript repositories

The graph presented on Figure 4.2 shows that, for first 3 epochs, the Javascript model was struggling to learn, however, from epoch 7, the value of perplexity is aligned with the Python model. The potential reason for it is that the Javascript syntax allows use of *closures* i.e. inner functions that have access to the variables within an enclosing function [jav]. The code written inside a closure is often irrelevant for the method call that follows after the closure. This language-specific feature may not be immediately captured during the training.

We present two test cases on Listing 4.6 and Listing 4.7 to demonstrate that the Javascript model generates method name suggestions suitable for a given context.

```
var http = require('http');
http.createServer(function (req, res) {
   res.writeHead(200,{'Content-Type':
      'text/html'});
   res.write(req.url);
   res.end();
}).listen(8080); <completion>
```

```
listen
on
end
pipe
addListener
address
installHandlers
attach
writeHead
createServer
```

Listing 4.6: Javascript test case 1 (`<completion>` marks the method name expected to be predicted)

In this section, we demonstrated that our approach can be used to train a method name completion engine for Javascript. Although it is uncertain whether similar results can be achieved for other programming languages with dynamic type systems, positive evaluation results suggest

```
exports.toCamelCase = function(str, upper) {
   str = str.toLowerCase().
      replace(/(?:(^.)|(\s+.)|(-.))/g,
         function(match) {
      return match.charAt(match.length - 1).
         toUpperCase();
   });
   if (upper) {
      return str;
   }
   return str.charAt(0).toLowerCase()
      + str.substr(1); <completion>
};
```

```
slice
substr
substring
charAt
toUpperCase
replace
charCodeAt
search
toString
join
```

Listing 4.7: Javascript test case 2 (`<completion>` marks the method name expected to be predicted)

that our approach has some potential.

4.5 Threats to Validity

In the following section, we present risks and threats to validity that we identified.
Although the dataset collected from GitHub is large and contains diverse projects, our findings may not generalize for other corpus. To mitigate the risk we demonstrated how our approach can scale to another dynamically typed programming language in Section 4.4. However, the dataset containing Javascript repositories was also extracted from the same code hosting facility i.e. GitHub. The results may also be affected because all software repositories in our training corpus are open-source, therefore, we cannot claim that our approach can be generalized to industrial projects.
The number of evaluation cases may also pose a threat. Our models were evaluated only on the set of twenty test cases that covers only a fraction of real-world tasks that code completion engines may face.

Chapter 5

Future Work

This chapter summarises a select number of further relevant ideas and objectives along with brief explanations of how these can be achieved.

5.1 Larger dataset

In our work, we have used a dataset consisting of a thousand software repositories collected from the *GitHub* code-hosting service. This data may be insufficient for evaluation purposes (see Section 4.5) as results based on a single source are likely to be biased. To mitigigate this risk, we have collected a larger dataset from *PyPi*, (see Section 3.1.2.2) which we were not able to process due to hardware limitations and constraints imposed by the OpenNMT framework (see Section 3.3.3). Training our code completion models on the *PyPi* dataset using more powerful hardware or a less resource-demanding framework may produce more accurate results.

5.2 Neural Network Architectures

We presented several types of neural machine translation systems and their architectures (see Section 2.2.3.3). The parametrisation space is large and has not been explored sufficiently. Further experimentation with different neural network architectures, including combinations of various types of encoders and decoders could have a positive impact on the overall performance. In addition, we can vary the parameters of the neural network such as number of layers and size of each layers in terms of neurons in order to achieve better performance.

5.3 Preprocessor models

Certain amounts of preprocessing may aid the overall process of code completion. We presented three different preprocesor models that extract relevant context for code completion from a source code file (see Section 3.2). As this has shown positive results (see Section 4.3.2), it may be helpful to revise these models and introduce additional features. For example, replace import aliases in Python with actual module names throughout the file before the preprocessing begins.
Another extension could be to combine the advanced model with the N-char model by extracting N chars preeceding the place where prediction should go along with all occurrences of the method invoker variable found throughout the code. There is a lot of room for improvement and experimentation with preprocessor models.

5.4 Generalisation to other programming languages

We demonstrated that our approach can be generalized to other dynamically typed programming languages such as Javascript (see Section 4.4). To prove the point of generalizability even further, it is needed to obtain the datasets consisting of software repositores for other dynamically typed programming languages such as Ruby or PHP and test our assumption of generalisability even further by training a method completion model on these datasets and comparing the results against the state-of-the-art static analysis technique.
If the system can be shown to scale well to other porgramming languages where types are dynamic, a software tool could be developed, which could train on any input given. That is, given a number of repositories and a scheme for method extraction, for example, an AST parser, a ready-to-go method completition plugin could be produced as output.

Chapter 6

Conclusion

To summarise the contribution of this project, we have proposed and implemented a novel approach for code completion in programming languages with dynamic type systems. Our approach is based on recurrent neural networks and makes no assumptions about the syntactic and semantric structure of the underlying programming language. Therefore, it is able to generalise well to other dynamically typed programming languages.

As a basis for training of our neural-based method name completion model, we have developed three different preprocessor strategies. A preprocessor strategy is a set of rules responsible to determine which parts of the program file are relevent for a method name at certain position to be completed. For example, one of our preprocessor strategies, N-chars , uses N characters before the method-invoking object in order to estimate the method name to be completed.

The evaluation of our model has yielded positive results. We demonstrated that our models can actually offer sensible method name suggestions in typical codefiles given ordinary but also edge-case code, for example, given inadequate variable naming or heavy reliance on third-party libraries.

The proposed method is promising, but is far from perfect, nevertheless. Training a neural network is often associated with multiple challenges. These include data acquisition, demanding training resource requirements in terms of both hardware and the large hyperparametrisation space. Exploration is extremely time consuming as testing new ideas and configurations requires training models from scratch. Given sufficient time and hardware resources, significant progress can be made in this area.

Bibliography

[ARSH14] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Daqing Hou. CSCC: Simple, efficient, context sensitive code completion. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 71–80. IEEE, 2014.

[art] A diagram of an artificial neural network. `https://tex.stackexchange.com/questions/132444/diagram-of-an-artificial-neural-network`. Accessed: 2017-06-11.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[BSF94] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[cod] Eclipse Code Recommenders. `http://www.eclipse.org/recommenders/`. Accessed: 2017-08-21.

[CVMBB14] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[edi] NetBeans Editor Code Completion API Documentation. `http://bits.netbeans.org/dev/javadoc/org-netbeans-modules-editor-completion/overview-summary.html`. Accessed: 2017-08-05.

[eri] The Eric Python IDE. `https://eric-ide.python-projects.org/`. Accessed: 2017-08-21.

[gita] Cloning a repository. https://help.github.com/articles/cloning-a-repository/. Accessed: 2017-08-01.

[gitb] GitHub API v3. https://developer.github.com/v3/. Accessed: 2017-06-22.

[gitc] GitPython is a python library used to interact with Git repositories. https://github.com/gitpython-developers/GitPython. Accessed: 2017-05-10.

[gro] GroundHog: library for implementing RNNs with Theano. https://github.com/lisa-groundhog/GroundHog. Accessed: 2017-09-03.

[GZZK16] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[HD17] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773. ACM, 2017.

[HP11] Daqing Hou and David M Pletcher. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 233–242. IEEE, 2011.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. LSTM can solve hard long time lag problems. In *Advances in neural information processing systems*, pages 473–479, 1997.

[HWM09] Sangmok Han, David R Wallace, and Robert C Miller. Code completion from abbreviated input. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 332–343. IEEE, 2009.

[jav] Understand JavaScript Closures With Ease. http://javascriptissexy.com/understand-javascript-closures-with-ease/. Accessed: 2017-07-14.

[jed] Jedi, auto-completion/static analysis library for Python. https://github.com/davidhalter/jedi. Accessed: 2017-08-21.

[kit] Kite: the smart copilot for programmets. https://kite.com/. Accessed: 2017-09-06.

[KKD$^+$] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush. OpenNMT: Open-Source Toolkit for Neural Machine Translation. *ArXiv e-prints*.

[LHKM13] Yun Young Lee, Sam Harwell, Sarfraz Khurshid, and Darko Marinov. Temporal code completion and navigation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1181–1184. IEEE Press, 2013.

[Lut08] Mark Lutz. Learning python, 3rd edition. In Tatiana Apandi, editor, *Learning Python, 3rd Edition*, chapter 16, pages 310–314. O'Reilly Media, 2008.

[MKF06] Gail C Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Eclipse IDE? *IEEE software*, 23(4):76–83, 2006.

[MP43] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[net] NetBeans IDE: fits the pieces together. https://netbeans.org/. Accessed: 2017-08-21.

[neu] Convolutional Neural Networks for Visual Recognition. `http://cs231n.github.io/neural-networks-1/`. Accessed: 2017-08-31.

[NHC+16] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522. ACM, 2016.

[nlt] NLTK 3.2.4 documentation: nltk.tokenize package. `http://www.nltk.org/api/nltk.tokenize.html`. Accessed: 2017-07-05.

[NNN+12] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, pages 69–79. IEEE Press, 2012.

[OYLM12] Cyrus Omar, YoungSeok Yoon, Thomas D LaToza, and Brad A Myers. Active code completion. In *Proceedings of the 34th International Conference on Software Engineering*, pages 859–869. IEEE Press, 2012.

[PLM15] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with Bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):3, 2015.

[pyc] PyCharm: Python IDE for Professional Developers by JetBrains. `https://www.jetbrains.com/pycharm/`. Accessed: 2017-08-21.

[pyp] PyPI - the Python Package Index. `https://pypi.python.org/pypi`. Accessed: 2017-05-19.

[pyt] The Python Standard Library Documentation: ast — Abstract Syntax Trees. `https://docs.python.org/3/library/ast.html`. Accessed: 2017-06-19.

[RBV16] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 731–747. ACM, 2016.

[RL08] Romain Robbes and Michele Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 317–326. IEEE Computer Society, 2008.

[rnn] Unreasonable Effectiveness of Recurrent Neural Networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`. Accessed: 2017-04-15.

[RVY14] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *ACM SIGPLAN Notices*, volume 49, pages 419–428. ACM, 2014.

[scra] Scrapy - An open source and collaborative framework for extracting the data you need from websites. `https://scrapy.org/`. Accessed: 2017-06-11.

[scrb] Scrapy Selectors. `https://doc.scrapy.org/en/latest/topics/selectors.html`. Accessed: 2017-06-11.

[seq] seq2seq: a general-purpose encoder-decoder framework for Tensorflow. `https://github.com/google/seq2seq`. Accessed: 2017-09-03.

[SFC$^+$17] Rico Sennrich, Orhan Firat, Kyunghyun Cho, Alexandra Birch, Barry Haddow, Julian Hitschler, Marcin Junczys-Dowmunt, Samuel Läubli, Antonio Valerio Miceli Barone, Jozef Mokry, and Maria Nadejde. Nematus: a Toolkit for Neural Machine Translation. In *Proceedings of the Software Demonstrations of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, pages 65–68, Valencia, Spain, April 2017. Association for Computational Linguistics.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[syn] Neural Networks: The synapse. `https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html`. Accessed: 2017-05-28.

[TSD14] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280. ACM, 2014.

[win] Wing Python IDE: the intelligent development environment for Python programmers. `https://wingware.com/`. Accessed: 2017-08-21.

[WSC$^+$16] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[XZC$^+$16] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 607–618. ACM, 2016.

[YWL06] Lean Yu, Shouyang Wang, and Kin Keung Lai. An integrated data preparation scheme for neural network data analysis. *IEEE Transactions on Knowledge and Data Engineering*, 18(2):217–230, 2006.