Deep Learning with Neural Attention for Code Suggestion

Avishkar Bhoopchand

MSc Computational Statistics and Machine Learning

supervised by

Dr. Sebastian Riedel

Mr. Tim Rocktäschel

Dr. Earl Barr

September 2016

Abstract

The Code Suggestion features of modern IDEs are invaluable productivity tools for programmers. While they tend to be reasonably functional in statically-typed languages, they offer little assistance when it comes to writing idiomatic code, lack functionality in dynamic programming languages and focus on single tokens rather than sequences. Recent work has shown that NLP language models can be successfully used to improve code suggestion systems by learning from repositories of natural code. The aim of this project was to investigate whether the performance of LSTM neural language models on code suggestion could be improved through the addition of neural attention mechanisms, designed to improve learning of long-range dependencies. It was found, on a new large Python corpus developed for this work, that Python code exhibits very long-range dependencies relating to references to previously declared identifiers. While standard neural language models were found to be very effective at suggesting syntax and structure, they struggled with these identifier references. The addition of attention mechanisms significantly improved the models ability to suggest identifier references. Standard attention models were found to outperform LSTMs in perplexity and in suggestion accuracy by up to 4 points, and 6 percentage points respectively. A novel model that allowed for multiple filtered attention mechanisms was also developed to more efficiently deal with the long-range dependencies in source code. The novel model surpassed the performance of a standard attention model with the same amount of memory by 2.5 perplexity points and 3 percentage points in suggestion accuracy. It also outperformed an attention model with more than double the attention memory while also being more interpretable.

Acknowledgements

Thank you to my supervisors, Sebastian, Tim and Earl for their excellent advice and for providing me with the opportunity to conduct this research. To my wife, Asmitha, who moved half-way across the world with me and provided tremendous support throughout the year. Finally, to my parents, the Chevening scholarship programme, University College London and Allan Gray for their support.

The code developed for this project is available on Github at `https://github.com/uclmr/pythonlm`.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the problem of code suggestion, explains why it is amenable to machine learning and then states the aims and key contributions of the dissertation.

1.1 Code Suggestion

As part of their task of developing software, programmers are required to combine the rich functionality offered by numerous frameworks and libraries with their own project's code. These projects may themselves be very large, potentially having evolved over many years with the contributions of numerous other programmers. The numbers of modules, classes and functions that programmers have access to can easily reach into the thousands and they often need to be combined in specific patterns or sequences in order to achieve the desired functionality. In order to help deal with the inherent complexity of software development, programmers tend to rely heavily on the tools provided by their IDEs. One of the most useful tools, provided in some shape or form by every modern IDE, is the code completion or code suggestion system. Code suggestion refers to the recommendation of one or more tokens[1] to extend a given sequence of complete tokens. Code completion on the other hand refers to the slightly different problem of recommending a completion to a partial token given a sequence of complete tokens and a partial token.

Existing code suggestion and completion engines are built using hand-coded

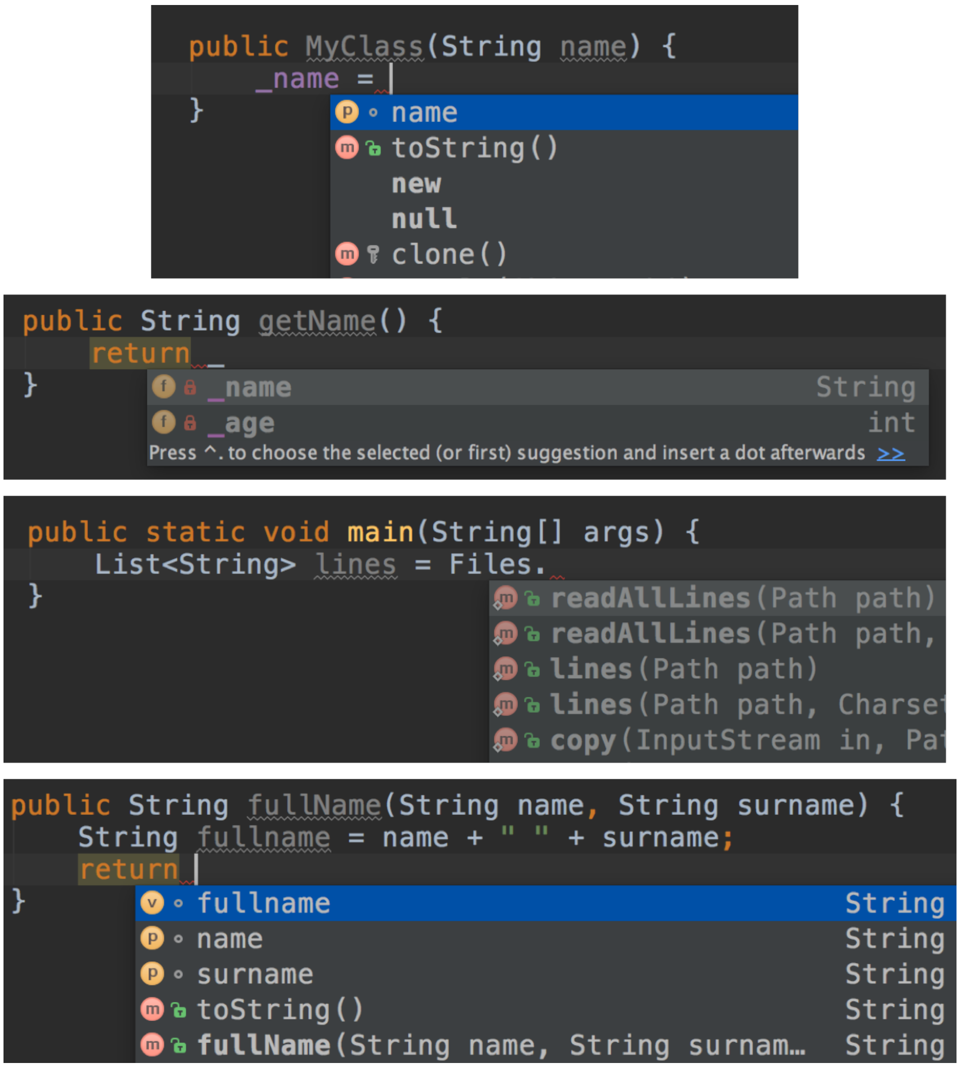
[1]A token is a sequence of characters that constitute a single logical or lexical unit

heuristics. Nevertheless, they have evolved over many years to become quite effective, particularly at code completion. This is especially true when applied to statically-typed languages where the type information can be leveraged to narrow the list of recommendations. This is not quite the case in dynamically typed languages such as Python. Figures 1.1 and 1.2 highlight some of these differences in capabilities using examples of suggestions made by the IntelliJ IDE for Java code and suggestions made by the same IDE on similar Python code respectively. It seems that the difference in capability is both due to a lack of type information in dynamic languages as well as possibly less time and effort spent on the engines for dynamic languages (as evidenced particularly by the first Python example, where it would be relatively easy to think of a heuristic to make a sensible recommendation). Existing solutions also tend to recommend single tokens rather than entire code snippets. Many IDEs have so called "live-template" features to partially address this, but these templates are fairly rigid, determined manually by the IDE creators or by developers themselves and may not necessarily reflect common practice or idiomatic code.

1.2 A machine learning perspective

A machine learning based alternative to the hard-coded heuristics was first proposed by Hindle et al [1] in 2012, who noted that "code, despite being written in an artificial language [...] is a natural product of human effort." By exploiting the predictable statistical properties they found in Java source code, they successfully applied machine learning techniques from the Natural Language Processing (NLP) community to build a code suggestion system. Such systems are capable of automatically learning language syntax, API methods and common patterns based on the analysis of source code written by actual programmers. The systems therefore work entirely with *idiomatic* source code, that is, "code written in a manner that other experienced developers find natural" [2].

Subsequent to the work of Hindle et al., who used fairly simple n-gram language models, the NLP community has moved on to so-called deep learning mod-



```
public MyClass(String name) {
    _name = 
}
            name
            toString()
            new
            null
            clone()
```

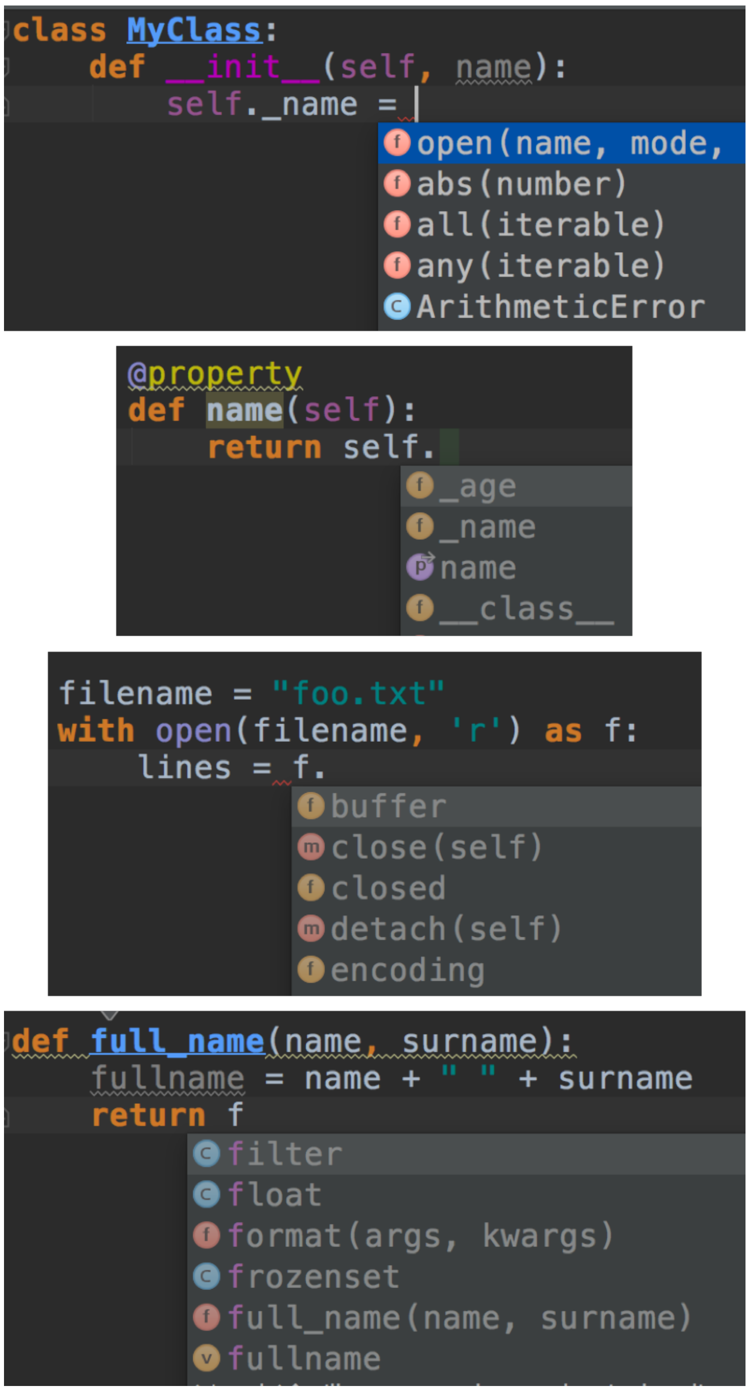
```
public String getName() {
    return _
}
        _name                                    String
        _age                                        int
Press ^. to choose the selected (or first) suggestion and insert a dot afterwards >>
```

```
public static void main(String[] args) {
    List<String> lines = Files.
}
                           readAllLines(Path path)
                           readAllLines(Path path,
                           lines(Path path)
                           lines(Path path, Charse
                           copy(InputStream in, Pa
```

```
public String fullName(String name, String surname) {
    String fullname = name + " " + surname;
    return 
}
        fullname                                String
        name                                    String
        surname                                 String
        toString()                              String
        fullName(String name, String surnam…    String
```

Figure 1.1: Java code completion recommendations from the IntelliJ IDE

els, spurred on by their tremendous success in language modelling with neural language models [3, 4, 5], machine translation [6] and speech recognition [7]. These models are based on artificial neural networks that were first introduced in the late ’50s [8], but have enjoyed a recent resurgence due to the availability of large datasets along with the computing power necessary to exploit them in the form of GPUs. More recently, attention mechanisms, which are a form of memory, have been added to the deep learning mix in NLP. These mechanisms equip models with an improved ability to deal with long-range dependencies, which occur when there are interactions that are many steps apart in an input sequence. It was hypothesised by Dam et al. [9] that programming language source code exhibits such long-range



```
class MyClass:
    def __init__(self, name):
        self._name = 
```

```
@property
def name(self):
    return self.
```

```
filename = "foo.txt"
with open(filename, 'r') as f:
    lines = f.
```

```
def full_name(name, surname):
    fullname = name + " " + surname
    return f
```

Figure 1.2: Python code completion recommendations from the IntelliJ IDE

dependencies.

The task of code suggestion is particularly ripe for exploitation by machine learning because of the availability of huge datasets on public source code repositories like Github. Many of the popular open-source projects on Github have contributions from a large numbers of experienced developers. The scrutiny in the form of pull requests and reviews and level of testing that occurs on many of these projects suggests that they contain very high quality code.

In short, the availability of large volumes of high-quality code, coupled with the inherent predictability of natural source code lends itself very well to modern deep learning techniques.

1.3 Aims, Contributions and Key Results

The primary aim of this work was to investigate whether neural attention mechanisms could be leveraged to deal with the long term dependencies in source code, thereby improving the performance of neural language models on the task of code suggestion. This aim can be broken down into the following key research questions.

- How effective are current deep neural language models on the task of code suggestion when applied to a dynamic programming language?
- Do neural attention mechanisms improve the ability of language models to deal with the long-range dependencies in source code?
- Can the long-range dependencies in source code be identified?
- Can attention mechanisms be adapted to better suit the particular dependencies in source code?

In order to answer these questions, a corpus of source code written in a dynamic programming language was required. As none were available from previous work, a contribution of this project was the creation of a new Python source code corpus. This corpus will be made available for future work and is discussed further in chapter 4.

An analysis of the corpus revealed that the majority of long-range dependencies in source code come from identifier references (tokens that refer to a previously declared identifier). It was hypothesised that these are difficult for existing language models to deal with and that focussing on improving a model's ability to deal with identifier references would provide the largest return in terms of suggestion accuracy. In this light, a new attention mechanism that is particularly well suited to the long range dependencies in source code was developed and is discussed in chapter 5.

Key findings of this work were that deep neural language models were very well suited to modelling programming languages and significantly outperformed the older models used in previous work. In particular, they were very effective at learning source code syntax and structure but struggled with long-range dependencies found in identifier references. The addition of standard attention mechanisms for language modelling significantly improved the performance of the models and led to a 6 percentage point improvement in suggestion accuracy. Finally, by focussing on the long-range dependencies found in source code, a novel attention model developed in this work outperformed the standard attention model, including one with more than double the memory available to it. The new model was also easier to interpret.

1.4 Outline

Chapter 2 describes the background in language modelling and deep learning necessary to understand the contributions of this dissertation. Chapter 3 provides a literature review of previous related work done in code suggestion using machine learning. Chapter 4 describes the Python corpus developed for this work. Chapter 5 goes on to extend current state-of-the-art attention mechanisms to suit the application to code suggestion. This new approach is evaluated alongside standard attention mechanisms as well as various baselines used in previous work in Chapter 6 before concluding.

Chapter 2

Background

This chapter describes the background knowledge in both n-gram and neural language modelling necessary to understand its application to code suggestion. Section 2.1 provides a short overview of n-gram language models, while section 2.2 moves on to deep learning, neural language models and the practical techniques used to train these powerful models on large datasets.

A language model, from the field of Natural Language Processing (NLP), is used to measure the probability of observing some sequence of tokens in a language. For example, for the sequence $S = a_1, a_2, ..., a_n$, the joint probability of S is given by equation 2.1, where the individual terms are estimated from a corpus of written or spoken language.

$$P(S) = P(a_1) \cdot \prod_{t=2}^{n} P(a_t | a_{t-1}, ..., a_1) \tag{2.1}$$

2.1 N-gram Language Models

A common form of language model which is both easy to train and to use is the n-gram model [10]. This model makes the (unrealistic) assumption that language follows the Markov property. The probability of occurrence of a particular token is conditionally independent of all preceding tokens, except for a fixed prefix of length n-1. For example, using a 4-gram model, the terms that make up the joint probability of observing S, are approximated as follows: $P(a_t | a_{t-1}, a_{t-2}, ..., a_1) \approx$

$P(a_t|a_{t-1}, a_{t-2}, a_{t-3})$.

The simplest, maximum likelihood estimates, of the probabilities in an n-gram model are obtained by counting observed token sequences. For example, the estimate of the probability $P(a_4|a_3, a_2, a_1)$ is $P(a_4|a_3, a_2, a_1) = \frac{\#(a_1\ a_2\ a_3\ a_4)}{\#(a_1\ a_2\ a_3)}$ where $\#(a_1, .., a_n)$ refers to the number of times the n-gram (sequence of n tokens) occured in the training corpus.

In practice, the procedure is complicated by the fact that certain token sequences may not occur in the model's training corpus, but may reasonably occur in some other corpus against which the model is evaluated. Smoothing is used to address this by combining Maximum A Posteriori estimation (incorporation of a prior distribution) and interpolation with lower-order estimates [11]. Popular techniques include the simple but effective Laplace [12] or "add-one" smoothing, which uses a Dirichlet prior that effectively adds one to every count. A more advanced technique called Kneser-Ney [13] smoothing, which uses interpolation, is also popular, along with a practically useful modification called Modified Kneser-Ney, introduced by Chen and Goodman [14].

2.2 Deep Learning and Neural language models

Recently, the best performing language models in NLP have been based on so called deep neural networks, in particular the Recurrent Neural Network (RNN). A neural language model directly estimates each term in the probability distribution of equation 2.1, without imposing a Markovian assumption. The models are able to capture long-term dependencies in token sequences through the use of memory. In order to understand these models, the rest of this section covers the development of RNN language models, starting with the most basic form of neural network and building up to the current state-of-the-art language models with neural attention mechanisms.

2.2.1 Neural networks

Neural networks have been in existance since the '50s when Rosenblatt introduced the first Perceptron model [8], loosely modelled on neurons found in the mammalian

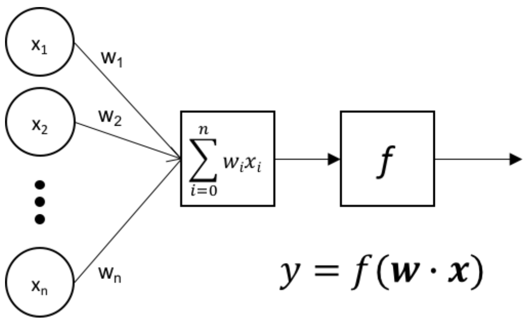


Figure 2.1: A basic Perceptron unit, also represented in vector form where $\mathbf{x} = [x_1...x_n]$ and $\mathbf{w} = [w_1...w_n]$

brain. As seen in Figure 2.1, the Perceptron can be modelled as a simple unit that takes in n distinct inputs (or an n-dimensional vector), computes a weighted sum of the inputs and applies an *activation function* (f) to the result to obtain the unit's output [15]. By optimising the weights $w_1, ..., w_n$, the unit can be used to learn a linear function that separates its input data into two classes. While the traditional activation function was a threshold function, modern implementations use smooth, non-linear, differentiable activation functions such as the logistic sigmoid (equation 2.2), hyperbolic tangent or rectifier (equation 2.3). [1]

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

$$\text{rectifier}(x) = \max(0, x) \tag{2.3}$$

Multiple Perceptron units can be combined to form a Multi-Layer Perceptron (MLP) also known as a feed-forward neural network. These consist of layers of units with the outputs of each unit connected to the input of every unit in the next layer. An example of a 3-layer feed-forward network, used for classification of a 4-dimensional input ($\mathbf{x}$) into one of two categories ($\mathbf{y}$), is illustrated in Figure 2.2. A more succinct vector representation is illustrated on the right of the figure, which will be the preferred form of illustration going forward. Finally, a matrix-vector

[1]The rectifier function presented here is not differentiable at exactly 0, but this fact can often be safely ignored in practice and it has been found to be effective in many applications. [16]

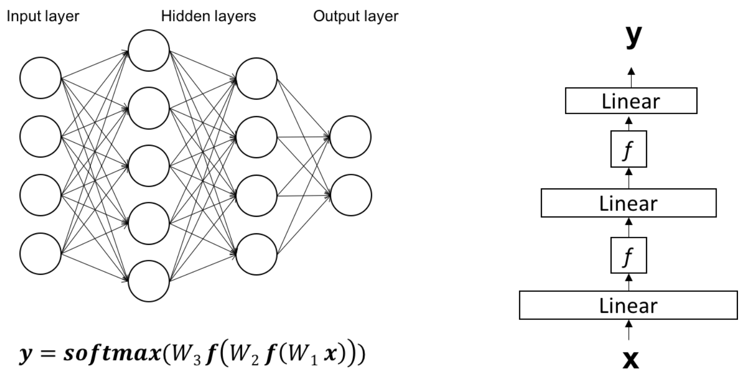


Figure 2.2: A 3-layer feed-forward neural network.

equation is presented in which the weights between the input and first hidden layer form a 5x4 parameter matrix W_1. Similarly, the weights between the hidden layers form the 4x5 matrix W_2 and the final set of weights, the 2x4 matrix W_3. The function **f** represents the activation function used in the units, applied in a point-wise fashion to its vector input.

Training neural networks involves simultaneous optimisation of the numerous weight parameters in order to minimise some loss function. The loss function quantifies the cumulative error made by the network, or the difference between the network's final output and the correct target for each training example in a supervised learning context.

A first-order, iterative optimisation algorithm called Gradient Descent is most commonly used to find a local minimum of the loss function. The algorithm updates each weight parameter in the direction of steepest descent along the loss function, represented by the negative partial derivative of the loss function with respect to that weight.

The partial derivatives themselves are calculated using the Backpropagation algorithm [17]. As seen in Figure 2.2, a neural network can be represented as a set of nested functions. The Backpropagation algorithm is then essentially the repeated application of the chain rule to compute the derivative of the final loss function with respect to each weight.

Gradient Descent requires the calculation of the loss over the entire training set

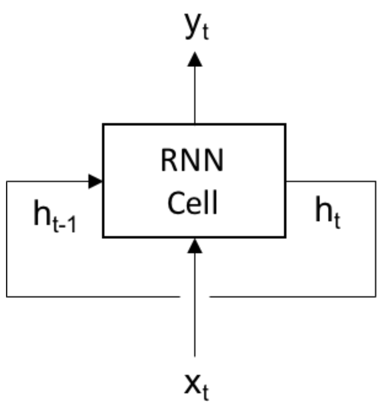


Figure 2.3: A basic RNN depicting the recurrent connection of an RNN cell

before updating any weights. For this reason, Stochastic Gradient Descent (SGD) is preferred in practice, where the loss for one, or a small mini-batch of (random) training examples are calculated and used to update the weights. Since the updates in SGD are estimates, they are scaled down by a learning rate, which has been found to reduce the chance of overshooting local minimums of the loss function [18].

2.2.2 Recurrent neural networks

Recurrent Neural Networks (RNNs) are neural networks that contain cyclic connections between units. These models are useful when the input data has an inherent sequential nature, such that the output of a particular unit may be influenced by the computation done on previous inputs in the sequence. They are therefore particularly useful in modelling the sequential nature of natural language. One way to think about RNNs is that they contain some internal memory representing the information captured so far as they process an input sequence. This idea is made more explicit when RNNs are combined with Long Short-Term Memory cells (LSTMs). A simple RNN in vector form is depicted in Figure 2.3. The input vector at step t in the input sequence is denoted $\mathbf{x_t}$, the output vector $\mathbf{y_t}$ and the *recurrent state* or "memory" vector which feeds back into the network and is used in the calculation of the next step is denoted $\mathbf{h_t}$ ($\mathbf{h_t}$ is also often referred to as the *hidden state*)

The calculations applied inside the RNN cell to the input and previous hidden state in order to derive the current output and hidden state are shown in equations 2.4

and 2.5. The matrices W_i, W_h and W_o represent a collection of weights, analogous to the W matrices of Figure 2.2 and $\mathbf{f}$ is once again a piecewise non-linear activation function. Note that the calculation of the output $\mathbf{y_t}$ at any given step t depends on the recurrent state $\mathbf{h_{t-1}}$. The recurrent state therefore captures relevant information contained earlier in the input sequence and can therefore be thought of as a simple form of memory.

$$\mathbf{h_t} = \mathbf{f}(W_i \cdot \mathbf{x_t} + W_h \cdot \mathbf{h_{t-1}}) \tag{2.4}$$

$$\mathbf{y_t} = \mathbf{f}(W_o \cdot \mathbf{h_t}) \tag{2.5}$$

SGD remains the most common method used to train RNNs in conjunction with a variation of the backpropagation algorithm called Backpropagation Through Time (BPTT) [19]. The network is unrolled to a fixed length, either the maximum length of possible input sequences, or some smaller fixed length in the case of Truncated BPTT. An example of an RNN unrolled 3 steps is depicted in Figure 2.4. Unrolling allows gradient information to propagate backwards through the network. For example, the gradient of the loss at step $t+1$ can propagate back to step t, $t-1$ and further.

Figure 2.4 also shows how RNNs can be stacked in layers to form a *deep* network. The output of one layer becomes the input to the next layer. Later layers tend to learn more abstract representations of the input sequence or may operate on different time scales [20].

RNNs can in theory learn long-range dependencies in sequences, which are interactions between inputs that are many steps apart. In practice, however, it often turns out that they struggle to do so because of the vanishing gradients problem [21]. This is a result of the common non-linear activation functions having gradients with absolute value less than one. The use of backpropagation to compute gradients in an n-step network, results in up to n multiplications of these small gradient values for the error propagation from the last to the first step. The error signal, represented by the gradient, therefore decreases exponentially with n, resulting in very small

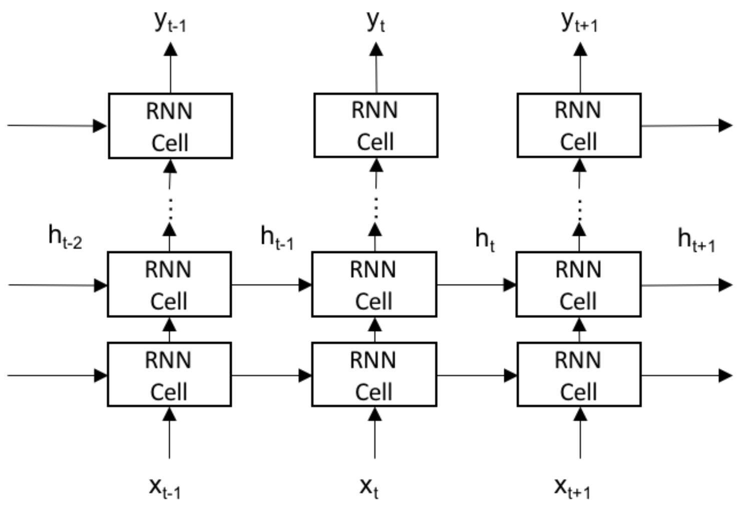


Figure 2.4: 3 steps of an unrolled RNN on which the BPTT algorithm can be run

(often negligible) updates to the parameters in earlier steps.

2.2.3 Long Short-Term Memory (LSTM) cells

LSTM cells were introduced by Hochreiter and Schmidhuber [22] in order to alleviate the vanishing gradients problem suffered by standard RNNs. These cells introduce gating mechanisms that control access to an internal memory representation. By doing so, they are able to learn long range dependencies by protecting important parts of their memory from being overwritten. The gates also help to preserve the error signal, allowing it to propagate much further back in time. The equations governing the output and hidden state vectors of an LSTM are shown in equations 2.6 to 2.11 and the important components of the LSTM are illustrated in Figure 2.5.

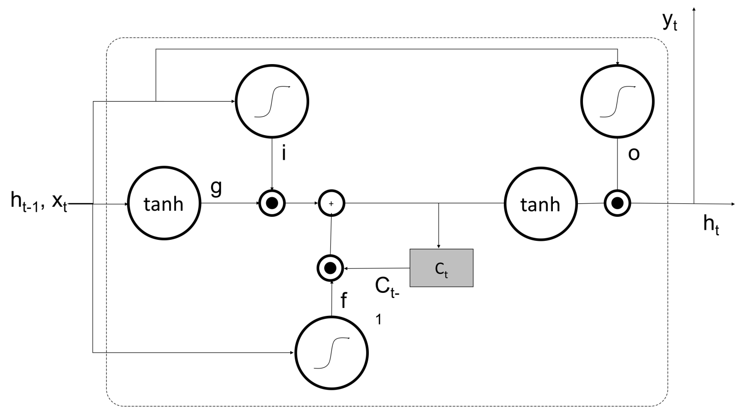


Figure 2.5: An LSTM cell showing the flow of data to the cell memory and out of the cell, controlled by the 3 gates *i*, *f* and *o*

$$\mathbf{i} = \sigma(\mathbf{x_t}W_x^i + \mathbf{h_{t-1}}W_h^i) \tag{2.6}$$

$$\mathbf{f} = \sigma(\mathbf{x_t}W_x^f + \mathbf{h_{t-1}}W_h^f) \tag{2.7}$$

$$\mathbf{o} = \sigma(\mathbf{x_t}W_x^f + \mathbf{h_{t-1}}W_h^f) \tag{2.8}$$

$$\mathbf{g} = \tanh(\mathbf{x_t}W_x^g + \mathbf{h_{t-1}}W_h^g) \tag{2.9}$$

$$\mathbf{c_t} = \mathbf{c_{t-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \tag{2.10}$$

$$\mathbf{h_t} = \tanh(\mathbf{c_t}) \odot \mathbf{o} \tag{2.11}$$

The 3 gates **i**, **f** and **o** are are calculated based on the current input and the previous recurrent state, using the same equation with different parameter matrices W. The sigmoid function σ ensures that the gates are vectors with elements between 0 and 1 (and are therefore smooth rather than digital gates). By taking the element-wise product ($\odot$) of **i** with the transformed input **g**, the input gate, **i**, effectively controls how much of the current input is used to update the cell memory ($\mathbf{c_t}$). The forget gate controls how much of the previous cell memory ($\mathbf{c_{t-1}}$) should be forgotten (decayed towards 0) and the output gate controls how much of the transformed cell memory ($\tanh(\mathbf{c_t})$) should be output.

2.2.4 RNNs for Neural Language Models

A modern neural language model can be built by combing RNNs with LSTM cells. The input sequences consist of tokens (which could be sentences of words, words or sentences of characters or the formally defined tokens of a programming language). Since RNN cells take numerical vectors as inputs, the unique tokens in the language, referred to as *token types* which make up the *vocabulary*, V, need to be encoded somehow. One approach is to use the *one-hot* encoding scheme. Each token type is assigned a unique, sequential, integer ID, τ and encoded as a $|V|$ dimensional vector with 1 in position τ and 0 elsewhere. The dimension of these vectors grows with the vocabulary size and can therefore be extremely large. Their sparsity also makes them inefficient.

A more efficient approach is to use a dense vector encoding, also known as an *embedding*. Each token type τ is mapped to a vector $\mathbf{x}_\tau \in \mathbb{R}^s$ where s refers to the embedding size. These dense representations are considered model parameters and can be trained. It was noted by Mikolov et al. [23], that learning these embeddings results in "high-quality distributed vector representations that capture a large number of precise syntactic and semantic word relationships". In other words, the model learns to automatically organise concepts, and capture the relationships between words, by placing syntactically and semantically related words close together in the embedding vector space. As a side note, this idea was extracted and used in a model called word2vec that has the sole purpose of building word embeddings. Mikolov et al. [23] found that certain semantic and syntactic patterns could be reproduced using vector arithmetic on their English word embeddings. For example, subtracting the vector representation of "man" from "brother" and adding "woman" would produce a vector closest to the embedding of "sister".

At each step, while processing the embedded input sequence, the RNN outputs a fixed size vector in $\mathbb{R}^s$. These vectors are each projected to $\mathbb{R}^{|V|}$ after which a softmax [24] (Equation 2.12) can be applied to derive a probability distribution over the token types in the vocabulary. During training, these distributions are compared to the actual next token that occurred in the training set and the cross-entropy [25] loss is calculated as a summary of the success or failure of the model to predict the next token. Equation 2.13 [26] shows the cross entropy loss for a single step using the model's predicted distribution **y** and the true target distribution $\mathbf{y}'$, which in this case is a one-hot encoding of the target token type. The symbols y'_k and y_k refer to the *k*th component of vectors $\mathbf{y}'$ and **y** respectively.

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{|V|} e^{x_j}} \tag{2.12}$$

$$\varepsilon(\mathbf{y}, \mathbf{y}') = -\sum_{k=1}^{|V|} y'_k \log y_k \tag{2.13}$$

An illustration of a typical neural language model, where the RNN has been unrolled for 3 steps, is provided in Figure 2.6. The input token type ids are labelled τ_1 to τ_3 and have corresponding trainable *s*-dimensional embedding vectors x_1 to x_3. Each LSTM cell receives the vector embedding of the current timestep as well as the recurrent state of the previous timestep as input. The initial recurrent state is set either to a vector of zeros or to the final state vector of a previous truncated sequence. The output distribution at each step, obtained by applying the softmax function to a linear projection of each output vector, is compared to the one-hot encoding of the target next token.

The neural language model presented in this section forms the basis of the neural models implemented for code suggestion in this work.

2.2.5 Practical Issues Related to Training

A number of practical issues arise when training RNNs on large, noisy datasets. The first potential issue is that of exploding gradients, essentially the opposite of the vanishing gradients problem discussed earlier, but not accounted for by LSTMs. A heuristic approach of gradient clipping, introduced by Pascanu et al. [27], is commonly employed to deal with this. Gradient vectors are rescaled whenever their (Euclidean) norms exceed some threshold. For example, the gradient vector **g** is

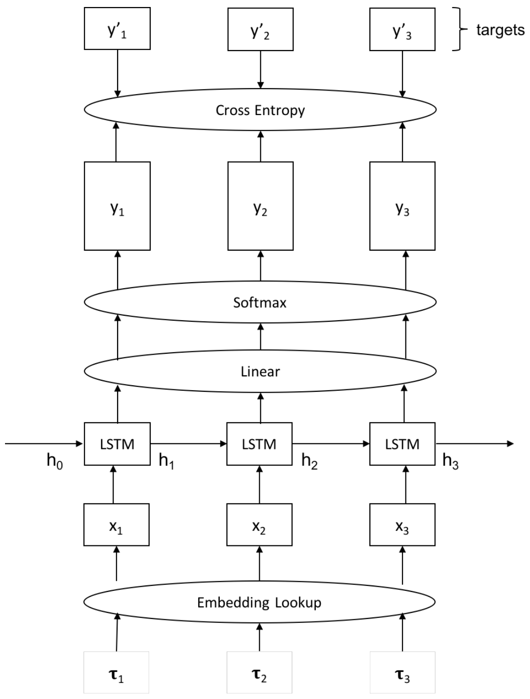


Figure 2.6: An illustration of a neural language model unrolled for 3 steps.

replaced with $\frac{threshold}{\|\mathbf{g}\|}\mathbf{g}$ whenever $\|\mathbf{g}\|$ is larger than the threshold.

Another issue commonly arising with complex machine learning models, particularly those as powerful as deep neural networks, is that of overfitting. Overfitting occurs when the model starts to fit the noise in its training data rather than capturing the true underlying signal. It is easily detected by noticing decreasing loss on the training set coupled with increasing loss on a held-out test or validation set. Overfitting is traditionally dealt with by regularisation, a useful form of which called *dropout* is common in deep neural networks. The key idea is that units are randomly dropped from the network during training, which was shown by Srivastava et al. [28] to "significantly reduce overfitting and give major improvements over other regularization methods". Dropout during training, effectively results in sampling from a number of "thinned" networks. When making inferences at test time, the use of the un-thinned network is akin to more traditional ensemble methods.

Finally, when dealing with large vocabularies, it is clear from equation 2.12, that the calculation of softmax can become expensive due to the sum over the entire vocabulary in the denominator. Hierarchical softmax [29] addresses this by structuring the softmax as a binary tree computation, thus requiring $O(\log|V|)$, rather than $O(|V|)$ operations to compute the probability of a single token. Sampled softmax [30], another alternative that can be used during training, re-expresses the denominator of equation 2.12 as an expectation. The expectation can then be estimated using importance sampling where the proposal distribution is the log-uniform distribution for a vocabulary that follows a Zipfian distribution (it is shown in chapter 4 that the vocabulary associated with Python source code does indeed follow a Zipfian distribution).

2.2.6 Attention

Attention mechanisms are a recent trend in NLP, having been successfully applied to sequence-to-sequence tasks such as machine translation [31], question-answering [32] and syntactic parsing [33]. Sequence-to-sequence models "consist of two [RNNs]: an encoder that maps an input sequence of words into a dense vector representation, and a decoder that conditioned on that vector representation generates an output sequence" [34]. Attention mechanisms were introduced to overcome the bottleneck of the single vector representation of the input sequence by allowing the decoder to refer back to, or attend to, output vectors generated by the encoder for each part of the input sequence, at each step of output generation.

The use of attention implies that the vector representation between the encoder and decoder no longer needs to capture the entire semantics of the input sequence, but rather just a representation informing the second RNN which of the encoder's output vectors it needs to attend over [35].

Very recently, attention mechanisms were adapted for use in language modelling by Cheng et al. [36], who considered attention as equipping LSTM cells with an expanded, fixed-length memory tape rather than a single memory cell. They achieved promising results in the standard Penn Treebank benchmark. In a similar vain, Tran et al. [37] added a "memory block" to LSTMs for language modelling of English, German and Italian and outperformed both n-gram and standard neural baseline models.

As they relate to a key aim of this work, attention mechanisms for language modelling as well as an extension making them more suitable to the long-term dependencies in source code are developed and discussed further in chapter 5.

Chapter 3

Related Work

This chapter reviews and highlights previous related work done on the application of NLP language models to the task of code suggestion. These works are compared and contrasted to the approach taken in this dissertation.

3.1 N-gram models

The majority of previous work in code suggestion using language modelling, and more generally the analysis of formal programming languages with NLP language models, has focussed on the use of n-gram models. Many of these works were inspired by Hindle et al. [1] who argued that while in theory, programming languages are complex, flexible and powerful, in practice, the programs that real people write fall in a much smaller space. Real programs contain significant repetitiveness and have sufficient predictable statistical properties that can be captured by statistical language models. They tested this hypothesis by training n-gram language models on a corpus of Java and C code and discovered that the models captured significant local regularity within projects that was not the result of programming language syntax. Regularity within a project refers to the "vocabulary, and specific local patterns of iteration, field access, method calls, etc". They applied their language model to build a code suggestion system in Eclipse, which they found to perform better than the standard Eclipse code suggestion system.

While Hindle et al's work was pioneering on the task of code suggestion, the NLP models they used have now become outdated. The key differences in this work are the use of more modern language modelling techniques, a significantly larger corpus of source code and the application to a dynamic, rather than a static language. As was noted in the introduction, existing code suggestion systems in IDEs are already quite good for static languages. Dynamic languages could arguably benefit more from machine learning.

Similar work to that of Hindle et al included Raychev et al [38] who used 3-gram models along with a simple variant of an RNN language model to complete “holes” in program snippets. While still technically code suggestion, it is unclear what practical application this unusual variation of the task would have.

Other work involving the application of n-gram language models to programming languages, albeit on different tasks to code suggestion, included those of Allamanis et al. [39] who used n-gram models to build a tool to suggest revisions to code to improve its consistency with the conventions of the rest of the project. Campbell et al. [40] noted that syntax errors should be surprising to a language model trained on a corpus of compilable code (the language model should assign erroneous tokens a low probability). They used this idea to build an improved error-reporting system for Java.

Tu et al. [41] enhanced the standard n-gram model, which captures global regularity in software, by incorporating a cache. This cache allowed them to exploit the high level of locality inherent in software that results from the specialisation of modules. They empirically verified that code tends to take on special forms of repetitiveness in local contexts, for example through the declaration of new identifiers and patterns of identifier usage. They evaluated their model against 2 small Java and Python corpora consisting of 9 projects each and found that their model was able to capture local regularity. This resulted in an accuracy improvement of between 16% and 45% over standard n-gram models.

The introduction of attention mechanisms to language models compares to Tu et al’s idea of adding a component to the language model that was specifically designed to exploit the properties of source code. As will be seen in chapter 6, the addition of a standard attention mechanism improves upon a neural language model’s ability to model the locality in source code, particularly relating to identifiers. This idea is taken a step further with the introduction of a novel architecture that further focusses on the unique properties of source code.

3.2 Programming Language Grammar

Programming languages need to be understood by computers and are therefore inherently unambiguous. They have formal grammar specifications and well-defined, deterministic parsers that can convert a lexed sequence of language tokens into an Abstract Syntax Tree (AST). This inherent structure was exploited by both Maddison and Tarlow [42] and Allamanis and Sutton [2] who used Probabilstic Context Free Grammars (PCFGs) to build a generative model of source code and to extract idiomatic patterns (also known as templates) respectively. The PCFG model allowed them to generate ASTs directly rather than working at the lexical level.

In particular, Maddison and Tarlow [42] argued that while the syntax of programming languages can be represented by context-free grammars, conforming to the grammar does not ensure that a program can actually compile. This is due to additional rules, not represented in the grammar, such as the requirement that variables be declared before being used. To model these additional context-dependent rules, they augmented their PCFG with traversal variables that maintained context depending on the AST generated so far. The traversal variables modulated the distribution over child trees. Similar extensions to PCFGs were successfully applied to a Javascript corpus by Bielik et al. [43].

These works, which involved direct modelling of ASTs, represent an alternative, and possible complement, to the purely lexical approach followed here. Maddison and Tarlow used a form of log-bilinear model and Allamanis and Sutton used Bayesian methods to implement their PCFGs, rather than neural models. Some components necessary to build a neural architecture to do similar modelling do technically exist though. In particular, Tai et al [44] introduced a generalisation of LSTMS called Tree-LSTMs, a variation of which can be used to model ASTs. While these models would be extremely useful for building a program representation to use in program classification, for example, it is unclear how they can be adapted for use in a generative context. Another notable related architecture called Recurrent Neural Network Grammar (RNNG), which can be used in a generative context, was developed recently by Dyer et al. [45]. The RNNG model operates similarly to PCFGs, but the context-free assumption is relaxed by the use of RNNs which condition on the entire history. They integrated a transition-based parser directly into their neural architecture, enabling their model to generate syntactic parse trees of English and Chinese and outperformed sequential LSTMs in a language modelling task.

Similar approaches to those mentioned were initially considered for this work, but their complexity, relative lack of research and implementations, coupled with the intuition that there was still large room for improvement in lexical-level models through the use of neural attention, led to the decision to focus on the lexical-level approach for this work. Furthermore, Maddison and Tarlow showed that even PCFGs don't guarantee compilable code and additional mechanisms would be necessary to ensure this. The excellent results already obtained by lexical-level neural models leads to the question of whether modelling ASTs do provide practical gains that would outweigh their complexity. This would be an interesting avenue for future work.

3.3 Datasets

A feature common to the majority of previous work in the area is that of relatively small datasets. This was addressed by Allamanis and Sutton [46] who constructed a corpus of 352 million lines of Java and analysed it with n-gram language models. They found that by using a sufficiently large dataset, they were able to construct a single language model that was effective across multiple different project domains, unlike Hindle et al. who found it necessary to build models on a per-project basis. They verified that identifiers were the largest contributor to vocabulary size and that the vocabulary grew by on average, 56 new token types per 1 000 lines of code due to the introduction of new identifiers. Their observation is similar in spirit to Heap's law [47], which states that the vocabulary size of a corpora of natural language will grow in an unbounded fashion. By segregating groups of identifiers, they established that method names tended to be more predictable than type and variable names. Ultimately, their language model was used to build new data-driven code complexity metrics.

Inspired by Allamanis and Sutton's findings, this work also used a single model trained on a large Python corpus, rather than training separate models on different projects. Similar results relating to the vocabulary and identifier names were found and are discussed in chapter 4. Their finding is particularly promising for the prospects of developing the models of this work into a real code suggestion system built into an IDE. It suggests that such a system can be pre-trained and does not necessarily need to be updated or customised to every unique project. While the project-based approach may be feasible for simple n-gram models, neural models are significantly more difficult and time-consuming to train and it would be impractical to re-train or even update them on developers local machines in a real system.

3.4 Deep learning

Coinciding with the switch from n-gram to neural language models in the general NLP community, White et al. [48] compared various n-gram models with modern neural language models in the programming language context. The n-gram models included those with n ranging from 2 to 9 as well as the cache-based model of Tu et al. They performed their analysis on a large Java corpus collected from Github. Their models were evaluated using Perplexity, an "intrinsic evaluation metric that estimates the average number of tokens to choose from at each point in a sequence." They found that the neural language model beat their best n-gram baseline (an 8-gram interpolated model with 100-token unigram cache) with a perplexity of 10.17 compared to 12.22. In the code suggestion task, they found their neural language model to also outperform the best n-gram model in terms of top-k accuracy. The neural model, in particular, scored 12 percentage points higher in top-1 accuracy.

Other research focussing on the use of deep neural language models for source code included that of Das and Shah [49] who compared an RNN model with GRU cells (an alternative to LSTMs) to a feed-forward neural network with attention. They noted that they were surprised that the feed-forward network with attention performed better than the RNN, but their evaluation was based on a 2 very small corpora, consisting of a single C and Python project respectively. The use of LSTM cells was compared to standard RNN cells (as used by White et al.), by Dam et al. [9] on the same dataset used by Hindle et al. They found that LSTMs significantly outperformed standard RNNs, which they hypothesised was due to their improved ability to learn long-term dependencies found in source code.

White el al's work validated the premise that neural language models would also outperform previous n-gram models on source code as they had done in other areas of NLP. It is for this reason that new work, such as that presented here, can largely ignore n-gram models and go straight to the more advanced neural models. Nevertheless, in light of their explicit mention of not claiming that the same result would apply to other programming languages, a selection of n-gram models were still run to re-confirm that the same finding applies to Python source code in chapter 6. The neural models used here were also more advanced than those used by White et al, who used the standard RNN's described in section 2.2.2. As explained in chapter 2, standard RNNs suffer from issues such as the vanishing gradient problem which hampers their ability to deal with long-range dependencies. The LSTM model which was found by Dam et al. to outperform RNN models forms the neural baseline used in this work. These models are then extended with the attention mechanisms that Das and Shah found to significantly improve performance even on less sophisticated feed-forward networks.

Other interesting applications of deep neural language models involving source code have been used in contexts such as code summarisation using Convolutional Neural Networks and an attention mechanism [50]. Mou et al [51] incorporated information from the Abstract Syntax Tree to build a vector representation of programs that can be used for program classification. Finally, Ling et al. [52] used deep neural networks with multiple attention mechanisms, each specialised to a particular task, to automatically generate implementations of Magic the Gathering cards in Java and Python. They generated a probability distribution over these attention mechanisms and used the weighted average output of each mechanism when generating their output sequence. This idea served as a strong inspiration for the novel architecture introduced in chapter 5, which like Ling et al. weighs the output of multiple task-specific attention mechanisms, but does so in a language modelling rather than a sequence-to-sequence context and also does so on a token level rather than the character level used by Ling et al.

Chapter 4

Data

As mentioned in chapter 3, the majority of previous work in code suggestion focussed either on very small corpora, or on statically-typed languages, particularly Java. For this reason, a new Python corpus was collected for this work. According to the programming language popularity website Pypl [53], Python is the 2nd most popular language after Java. It is also the 3rd most common language in terms of number of repositories on the open-source code repository, Github, after Javascript and Java [54].

4.1 Data Collection

Github is the largest hosting platform for source code in the world, with more than 14 million users and over 35 million repositories [55]. Coupled with the fact that it offers numerous metrics on each repository it hosts and has an easy to use API for data retrieval, it was the logical choice to obtain a new source code corpus.

It is difficult to automatically determine what constitutes good quality code. For this reason, a number of heuristics were developed in order to increase the likelihood of obtaining good quality code as well as to restrict the corpus to a manageable size. To download the Python corpus, the Github API was queried for the top 1000 projects with more than 100 stars and sorted by the number of forks descending. Stars are similar to bookmarks by users, whereas forks are copies of a repository that allow users to freely experiment with changes without affecting the original repository [56]. The large thresholds applied to these metrics suggested a high degree of interest in the projects considered. It was assumed that projects with a high level of interest tend to be of good quality as they are likely used and analysed by many people. This methodology was similar to that used by Allamanis and Sutton [46] and Allamanis et al [39].

Once the list of 1000 repositories were obtained, they were cloned using a Python implementation of the `git` tool. The URL and exact commit SHA [1] of each repository was recorded and will be made available along with a script to clone the repositories. This will allow anyone interested in doing future work using the same corpus to easily reproduce it.

The constituent Python files from the downloaded repositories, were split at a project level into a training set, validation set and test set (the lists of which will again be made available). Hashes of files were compared across sets to ensure that there was no leakage of information from the training set into the validation or test sets.

4.2 Corpus Statistics

Basic statistics for the corpus are provided in table 4.1, which shows that this corpus is significantly larger than those used in previous code suggestion work. Figure 4.1 shows the distribution of token types in the training set on a log-log scale. It indicates, for example, that approximately 1 million token types occur only once in the entire training set and another 1 million token types occur between 1 and 10 times. It is clear that the distribution follows Zipf's law, which states that given some corpus of natural language, the frequency of any word is inversely proportional to its rank in the frequency table [57]. It implies roughly that the vast majority of the token types that make up the vocabulary, occur very infrequently in the corpus. A Zipfian distribution was also found in a large Java corpus in an analysis by Linstead et al. [58].

The majority of the long tale in the vocabulary of the corpus is made up of unique identifiers in the source code. These are the names given by programmers

[1]Git uses an SHA hash to uniquely identify commits to a repository. Their tooling allows one to restore a repository to the exact state it was in after a given commit

Table 4.1: Python corpus statistics

Dataset	Files	Lines	Tokens	Vocab Size
Train	118 298	26 868 583	88 935 698	2 323 819
Validation	26 466	5 804 826	18 147 341	
Test	43 062	8 398 100	30 178 356	
Total	187 826	41 071 509	137 261 395	2 323 819

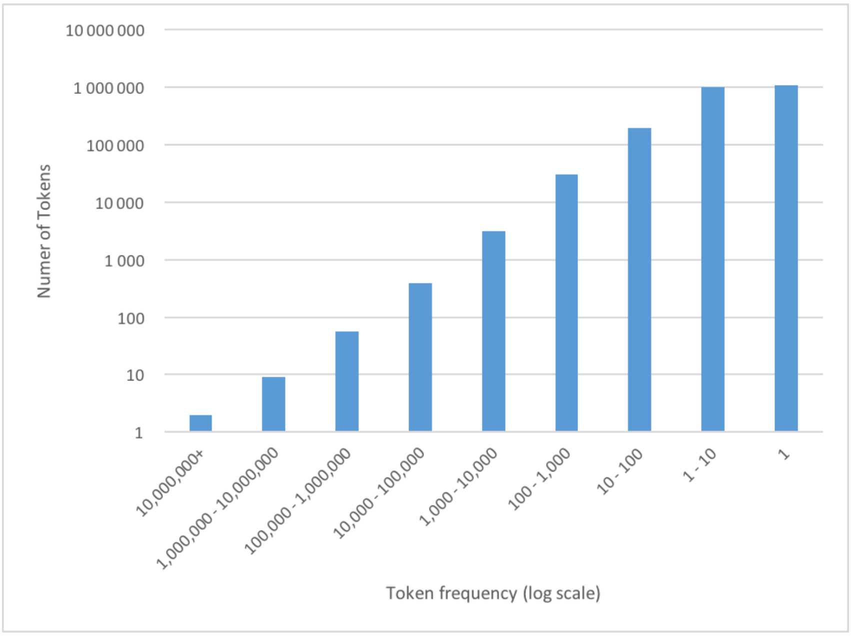


Figure 4.1: Vocabulary distribution of the Python corpus training set

to the classes, variables, arguments, attributes and functions in a program. Out of the vocabulary of 2.3 million token types, almost 1 million are identifier names. Identifiers therefore warrant focussed analysis.

Figure 4.2 shows a box plot of the distances (in number of tokens) between identifier declaration and usage, split by identifier group. The figure shows that the dataset clearly exhibits the long term dependencies which Dam et al. hypothesised to exist in source code. Not surprisingly, arguments and variables tend to be referenced a lot sooner than attributes, functions and classes which exhibit some very long term dependencies as shown by the 3rd quartile of distances between 500 and 1000. It would not be surprisingly, therefore, for language models to struggle to predict identifier usage due to these extremely long range dependencies.

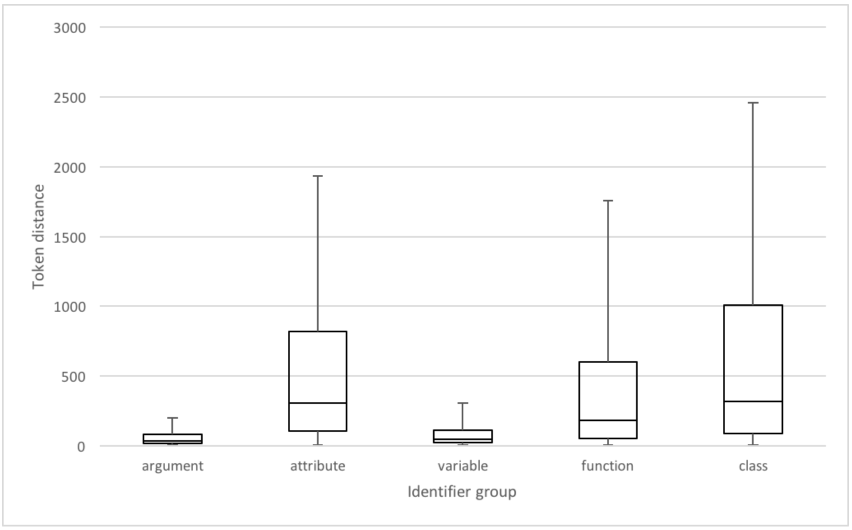
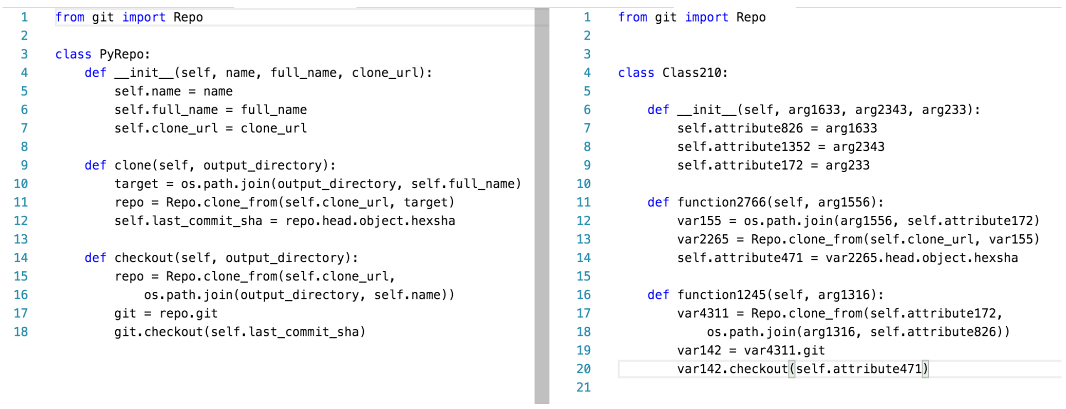


Figure 4.2: Token distances between identifier declaration and usage indicating long term dependencies in source code

4.3 Normalisation of Identifiers

The Python corpus discussed in this chapter contains a large vocabulary with a long tail of infrequent words. This presents a potential problem for language models which require sufficient data to form sensible embeddings of every token type contained in the vocabulary. Many of the tokens in the corpus occur only once and the majority of the long tail consists of identifier names.

For this reason, as well as to enable a sensible analysis of model performance on identifiers, a normalised dataset was derived. Each identifier name was replaced with a new name indicating the identifier group (class, variable, argument, attribute or function) and a number that makes the identifier unique within its enclosing scope. This pre-processing step was performed by parsing each source code file into an Abstract Syntax Tree (AST) from which the declaration points and scope of each identifier could be determined. Using this information, the ideas of Hermann et al. [59] were applied to generate a new name for each identifier by combining the identifier group name with a random number. Random numbers were chosen instead of sequential numbers to prevent models from learning any spurious pattern in the number sequence. The new names were replaced in the AST and the code regenerated. Note that only novel identifiers defined within the scope of a single file



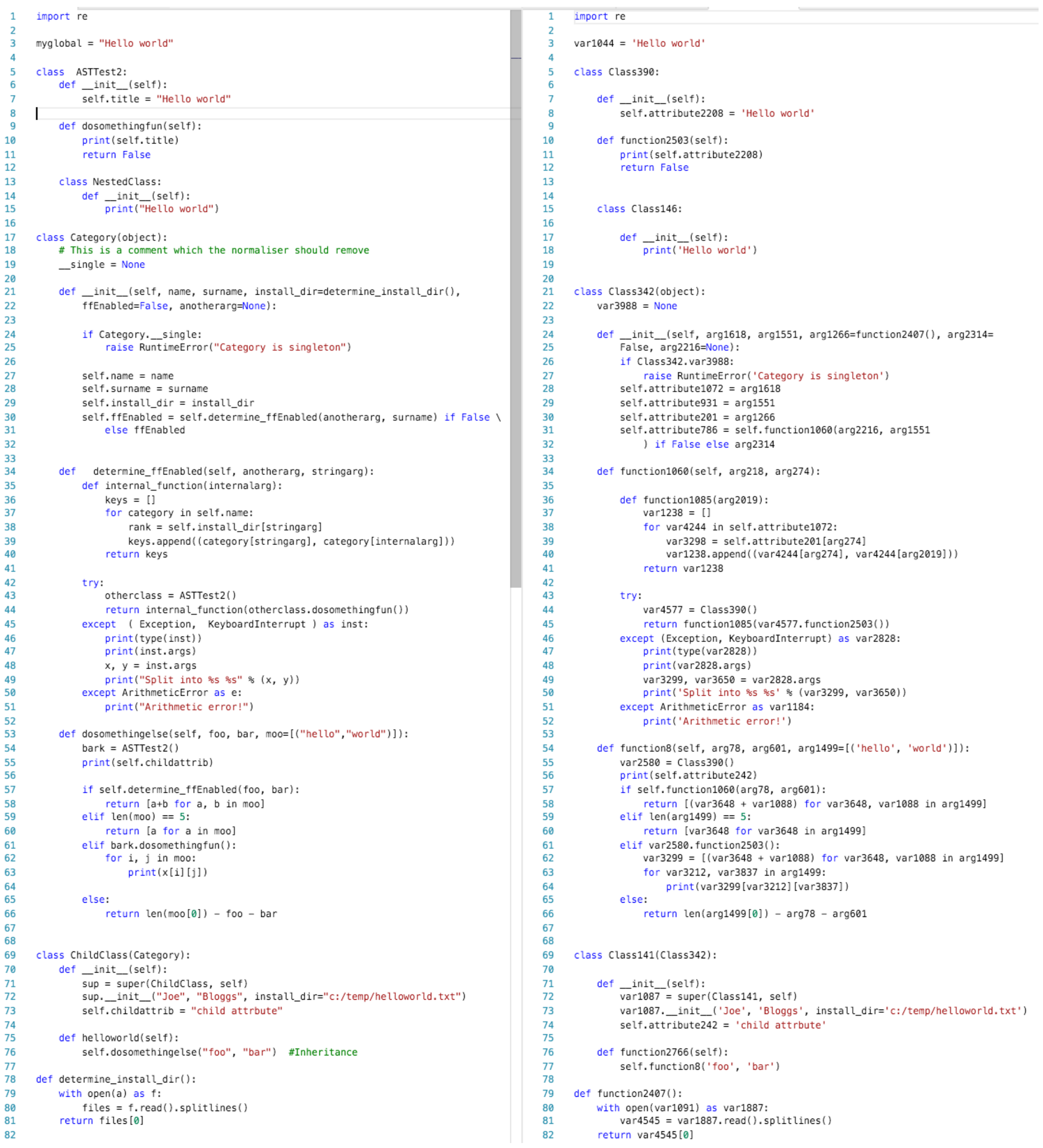
```python
from git import Repo

class PyRepo:
    def __init__(self, name, full_name, clone_url):
        self.name = name
        self.full_name = full_name
        self.clone_url = clone_url

    def clone(self, output_directory):
        target = os.path.join(output_directory, self.full_name)
        repo = Repo.clone_from(self.clone_url, target)
        self.last_commit_sha = repo.head.object.hexsha

    def checkout(self, output_directory):
        repo = Repo.clone_from(self.clone_url,
            os.path.join(output_directory, self.name))
        git = repo.git
        git.checkout(self.last_commit_sha)
```

```python
from git import Repo


class Class210:

    def __init__(self, arg1633, arg2343, arg233):
        self.attribute826 = arg1633
        self.attribute1352 = arg2343
        self.attribute172 = arg233

    def function2766(self, arg1556):
        var155 = os.path.join(arg1556, self.attribute172)
        var2265 = Repo.clone_from(self.clone_url, var155)
        self.attribute471 = var2265.head.object.hexsha

    def function1245(self, arg1316):
        var4311 = Repo.clone_from(self.attribute172,
            os.path.join(arg1316, self.attribute826))
        var142 = var4311.git
        var142.checkout(self.attribute471)

```

Figure 4.3: A small example of a normalised Python file, with the original on the left and normalised on the right

were affected by the normalisation process. Identifier references to external APIs and libraries were left untouched. The process of regenerating the code from the AST also had an additional benefit of ensuring consistent formatting across the corpus. A small example of a normalised Python file is presented in figure 4.3. An example of one of the test cases used in the development of the normalisation script is presented in Figure 4.4. The code is not actually useful, but it is valid Python and shows a number of different situations in which identifier names can occur as well as difficulties such as inheritance. Note that once again, the script used to normalise the corpus will be made available to future researchers.

The normalisation process resulted in a significant reduction in the vocabulary size to 987 151 token types (after also replacing numbers with a special <num> token). The normalisation has the added benefit of preventing models from predicting common names in places where a reference to a previously declared variable is called for, despite that name not being declared in the particular file. Finally, by preserving the identifier's group, neural models would still be able to sensibly differentiate between the different identifier groups in the embedding space if necessary.



```python
import re

myglobal = "Hello world"

class  ASTTest2:
    def __init__(self):
        self.title = "Hello world"

    def dosomethingfun(self):
        print(self.title)
        return False

    class NestedClass:
        def __init__(self):
            print("Hello world")

class Category(object):
    # This is a comment which the normaliser should remove
    __single = None

    def __init__(self, name, surname, install_dir=determine_install_dir(),
        ffEnabled=False, anotherarg=None):

        if Category.__single:
            raise RuntimeError("Category is singleton")

        self.name = name
        self.surname = surname
        self.install_dir = install_dir
        self.ffEnabled = self.determine_ffEnabled(anotherarg, surname) if False \
            else ffEnabled


    def   determine_ffEnabled(self, anotherarg, stringarg):
        def internal_function(internalarg):
            keys = []
            for category in self.name:
                rank = self.install_dir[stringarg]
                keys.append((category[stringarg], category[internalarg]))
            return keys

        try:
            otherclass = ASTTest2()
            return internal_function(otherclass.dosomethingfun())
        except  ( Exception,  KeyboardInterrupt ) as inst:
            print(type(inst))
            print(inst.args)
            x, y = inst.args
            print("Split into %s %s" % (x, y))
        except ArithmeticError as e:
            print("Arithmetic error!")

    def dosomethingelse(self, foo, bar, moo=[("hello","world")]):
        bark = ASTTest2()
        print(self.childattrib)

        if self.determine_ffEnabled(foo, bar):
            return [a+b for a, b in moo]
        elif len(moo) == 5:
            return [a for a in moo]
        elif bark.dosomethingfun():
            for i, j in moo:
                print(x[i][j])

        else:
            return len(moo[0]) - foo - bar


class ChildClass(Category):
    def __init__(self):
        sup = super(ChildClass, self)
        sup.__init__("Joe", "Bloggs", install_dir="c:/temp/helloworld.txt")
        self.childattrib = "child attrbute"

    def helloworld(self):
        self.dosomethingelse("foo", "bar")  #Inheritance

def determine_install_dir():
    with open(a) as f:
        files = f.read().splitlines()
    return files[0]

```

```python
import re

var1044 = 'Hello world'

class Class390:

    def __init__(self):
        self.attribute2208 = 'Hello world'

    def function2503(self):
        print(self.attribute2208)
        return False


    class Class146:

        def __init__(self):
            print('Hello world')


class Class342(object):
    var3988 = None

    def __init__(self, arg1618, arg1551, arg1266=function2407(), arg2314=
        False, arg2216=None):
        if Class342.var3988:
            raise RuntimeError('Category is singleton')
        self.attribute1072 = arg1618
        self.attribute931 = arg1551
        self.attribute201 = arg1266
        self.attribute786 = self.function1060(arg2216, arg1551
            ) if False else arg2314

    def function1060(self, arg218, arg274):

        def function1085(arg2019):
            var1238 = []
            for var4244 in self.attribute1072:
                var3298 = self.attribute201[arg274]
                var1238.append((var4244[arg274], var4244[arg2019]))
            return var1238

        try:
            var4577 = Class390()
            return function1085(var4577.function2503())
        except (Exception, KeyboardInterrupt) as var2828:
            print(type(var2828))
            print(var2828.args)
            var3299, var3650 = var2828.args
            print('Split into %s %s' % (var3299, var3650))
        except ArithmeticError as var1184:
            print('Arithmetic error!')

    def function8(self, arg78, arg601, arg1499=[('hello', 'world')]):
        var2580 = Class390()
        print(self.attribute242)
        if self.function1060(arg78, arg601):
            return [(var3648 + var1088) for var3648, var1088 in arg1499]
        elif len(arg1499) == 5:
            return [var3648 for var3648 in arg1499]
        elif var2580.function2503():
            var3299 = [(var3648 + var1088) for var3648, var1088 in arg1499]
            for var3212, var3837 in arg1499:
                print(var3299[var3212][var3837])
        else:
            return len(arg1499[0]) - arg78 - arg601


class Class141(Class342):

    def __init__(self):
        var1087 = super(Class141, self)
        var1087.__init__('Joe', 'Bloggs', install_dir='c:/temp/helloworld.txt')
        self.attribute242 = 'child attrbute'

    def function2766(self):
        self.function8('foo', 'bar')

def function2407():
    with open(var1091) as var1887:
        var4545 = var1887.read().splitlines()
    return var4545[0]
```

Figure 4.4: A test case used in the development of the normalisation process

Chapter 5

Neural Attention for Code Suggestion

This chapter discusses neural attention mechanisms for language modelling and develops a new variation which, by leveraging the unique long-range dependencies in source code, is particularly well suited to the task of code suggestion.

5.1 Attention for Language Modelling

Standard neural attention mechanisms for language modelling (such as those of Cheng et al. [36] and Tran et al. [37]) utilize a fixed-sized rolling window of k token embeddings that enter an *attention memory* (M_t). At each step in the input sequence, the current state of the network, represented by the LSTM's recurrent state $\mathbf{h}_t$, is used to determine a probability distribution ($\boldsymbol{\alpha}_t$) over the vectors in memory. A weighted average ($M\boldsymbol{\alpha}_t^T$) of the memory vectors is then added to the LSTM's output vector. In this way, the LSTM's single state vector does not bear the full burden of capturing all relevant information from the past. It can instead inform the attention mechanism which of the previous m outputs to look back on to obtain relevant information.

When modelling short sequences such as natural language sentences, it is practical to have a memory size that can capture the entire input sequence. However, as noted in chapter 4, programming languages contain certain very long range dependencies which can't practically be captured by such fixed sized rolling windows. For this reason, a new approach was considered that can explicitly capture these long term dependencies.

5.2 Attention for Code Suggestion

The key idea of the novel approach presented in this chapter was to allow for one or more separate, *filtered* attention mechanisms, each with its own memory and corresponding to some learned suggestion *task*. The suggestion tasks may have reduced vocabularies and can therefore specialise on different classes of suggestions. Instead of relying on fixed token windows, the tokens that enter each independent attention memory are indicated by an input flag. Since these flags may be sparse in the input sequence, attention mechanisms can be specialised to deal with specific forms of long-range dependencies that span larger distances than any practical fixed window.

At each prediction step, the model outputs a distribution over the tasks, one of them being to predict form the standard LSTM language model, which may be considered a fall-back. The final prediction distribution consists of the weighted average of the language model and each attention mechanism.

The specific task considered in this work, which may be considered a proof of concept, was to predict a reference to a previously declared identifier. The tokens corresponding to identifier declarations were flagged to enter the attention memory for this task. When predicting the next token in a sequence, the model was free to choose, based on the current state and input, whether to utilise the specialised identifier attention mechanism (similar to a symbol table) or to fall back to the more general language model. Other, more concrete developer use-cases may also be modelled as tasks, such as the completion of a function call.

5.3 Formal Specification

5.3.1 Attention mechanisms

Suppose that there are n attention tasks, each with a fixed memory size of k. For each step t and for each attention task i, an attention distribution $\boldsymbol{\alpha}_{ti}$ and attention vector $\mathbf{a}_{ti}$ is calculated according to equations 5.1 to 5.4. M_{ti} refers to the memory accumulated for task i up to point t in the sequence. W_i^V and W_i^h are parameters of size $\mathbb{R}^{s\ x\ s}$ and w_i is a parameter of size $\mathbb{R}^s$ where s refers to the size of the embedding vectors. Finally, note that $\mathbf{1}_k$ represents a k-dimensional vector of ones and $\otimes$ a point-wise product. Equations 5.1 to 5.4 are adapted from those used for sequence-to-sequence attention by Rocktäschel et al. [35], with the key difference being the addition of the i subscript to cater for multiple possible attention mechanisms.

The vectors entering the memory $(\mathbf{e}_{i1}, ..., \mathbf{e}_{ik})$ may be either the input embedding vector at a flagged input step or the corresponding output vector.

$$M_{ti} = [\mathbf{e}_{i1}\ \mathbf{e}_{i2}\ ...\ \mathbf{e}_{ik}] \qquad \in \mathbb{R}^{s\ x\ k} \tag{5.1}$$

$$G_{ti} = \tanh(W_i^M M_{ti} + W_i^h \mathbf{h}_t \otimes \mathbf{1}_k) \qquad \in \mathbb{R}^{s\ x\ k} \tag{5.2}$$

$$\boldsymbol{\alpha}_{ti} = \text{softmax}(w_i^T G_{ti}) \qquad \in \mathbb{R}^{1\ x\ k} \tag{5.3}$$

$$\mathbf{a}_{ti} = M_{ti} \boldsymbol{\alpha}_{ti}^T \qquad \in \mathbb{R}^{s} \tag{5.4}$$

5.3.2 Attention for Language Modelling

The standard attention mechanism used in language modelling, such as those of Cheng et al. [36] and Tran et al. [37], can be considered a special case of equations 5.1 to 5.4 [1] where the i subscript is dropped as there is only one such mechanism per time step. The vectors making up M_t are set to a fixed window of the last k LSTM output vectors. At each step, the attention vector $\mathbf{a}_t$ is combined with the output vector of the LSTM, $\mathbf{o}_t$ (the equations for which were in section 2.2.3) according to equation 5.5. Here W_O is a projection matrix of size $\mathbb{R}^{s\ x\ 2s}$ and the non-linear activation function $\mathbf{f}$ is optional, but was chosen to be tanh when implementing this model. The resulting final output vector $\mathbf{O}_t$ is projected to size $\mathbb{R}^{|V|}$ and a softmax is applied, as is standard in neural language models, to arrive at a probability distribution $\mathbf{y}_t$ over the the token types in the vocabulary. This process is presented in equations 5.6 and 5.7 where W_V and $\mathbf{b}_v$ are affine projection parameters of size $\mathbb{R}^{|V|\ x\ s}$ and $\mathbb{R}^{|V|}$ respectively.

[1] The model was in fact implemented as a special case in this work

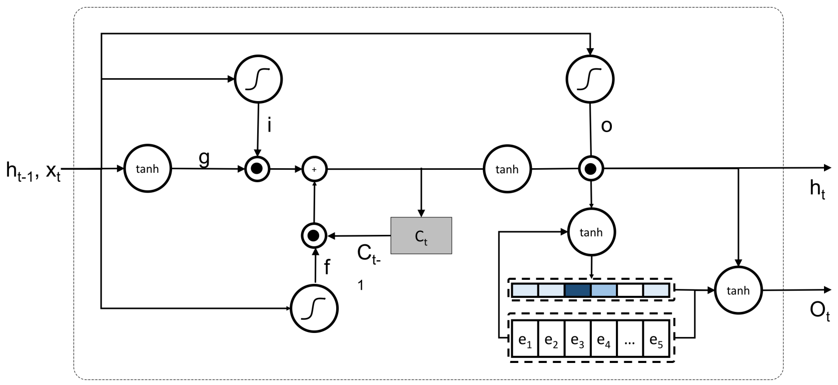


Figure 5.1: An LSTM cell with built-in attention mechanism

$$\mathbf{O}_t = \mathbf{f}(W_O \begin{bmatrix} \mathbf{o}_t \\ \mathbf{a}_t \end{bmatrix}) \qquad \in \mathbb{R}^s \qquad (5.5)$$

$$\mathbf{l}_t = W_V \mathbf{O}_t + \mathbf{b}_V \qquad \in \mathbb{R}^{|V|} \qquad (5.6)$$

$$\mathbf{y}_t = \text{softmax}(\mathbf{l}_t) \qquad \in \mathbb{R}^{|V|} \qquad (5.7)$$

The combination of the LSTM cell with an attention mechanism can be conveniently thought of as a single unit replacement for LSTM cells in a neural language model. An illustration of an LSTM with integrated attention is presented in figure 5.1

5.3.3 A Novel Approach

The novel approach introduced here allows for n attention mechanisms as described in section 5.3.1, with the addition of a memory component that tracks the vocabulary id of each item in memory, referred to as M'_{ti} and presented in equation 5.8. Items enter each attention memory only as a result of an input flag indicating that they should do so, rather than through a fixed window. Instead of being integrated into the LSTM cells, the attention mechanisms are considered separately. The standard LSTM output is projected to $\mathbb{R}^{|V|}$ and has a softmax applied to it to to obtain the standard neural language model probability distribution $\mathbf{y}_t$ over the vocabulary. A distribution $\boldsymbol{\lambda}_t$ over the n attention tasks and the language model itself, is then calculated using equation 5.10. Here W^{λ} and $\mathbf{b}^{\lambda}$ are trainable affine projection parameters of size $\mathbb{R}^{(n+1)\ x\ s}$ and $\mathbb{R}^{n+1}$ respectively. The state representation vector $\mathbf{h}_t^{\lambda}$, combines information from the recurrent state of the language model, the current input and the n attention mechanisms. It is calculated according to equation 5.9, where $W^{h^{\lambda}}$ is a projection matrix of size $\mathbb{R}^{s\ x\ (n+2)s}$.

$$M'_{ti} = [\text{id}_{ti1}\ \text{id}_{ti2}\ ...\ \text{id}_{tik}] \qquad \in \mathbb{N}^k \tag{5.8}$$

$$\mathbf{h}_t^{\lambda} = W^{h^{\lambda}} \begin{bmatrix} \mathbf{h}_t \\ \mathbf{x}_t \\ \mathbf{a}_{t1} \\ \vdots \\ \mathbf{a}_{tn} \end{bmatrix} \qquad \in \mathbb{R}^{size} \tag{5.9}$$

$$\boldsymbol{\lambda}_t = \text{softmax}(W^{\lambda}\mathbf{h}_t^{\lambda} + \mathbf{b}^{\lambda}) \qquad \in \mathbb{R}^{n+1} \tag{5.10}$$

The standard language model can be considered a special “suggestion task 0”. It assigns a probability $P(\tau|\text{task} = 0, \text{context}_t) = (\mathbf{y}_t)_\tau = (\text{softmax}(W_V\mathbf{o}_t + \mathbf{b}_V))_\tau$ to every token type with unique integer id τ in the vocabulary. Note that $(\mathbf{y}_t)_\tau$ refers to the τth component of the LSTM probability vector $\mathbf{y}_t$ and $\tau \in [1, |V|]$. Each of the n attention probability vectors $\boldsymbol{\alpha}_{t1}$ to $\boldsymbol{\alpha}_{tn}$ can also be thought of as a probability mass function where $P(\tau = (M'_{ti})_j|\text{task} = i, \text{context}_t) = (\boldsymbol{\alpha}_{ti})_j$ for $i \in [1,n]$ and $j \in [1,k]$ and is 0 for all other token types. Note that $(M'_{ti})_j$ is the jth component of the vector memory M' of the ith attention mechanism at step t.

The final output probability of the model, $\mathbf{Y}_t$ at step t is then calculated as the $\boldsymbol{\lambda}_t$ weighted average of the probabilities assigned by each of the tasks. Since $P(\text{task} = i|\text{context}_t) = (\boldsymbol{\lambda}_t)_i$, the final probability assigned to token type τ is deter-

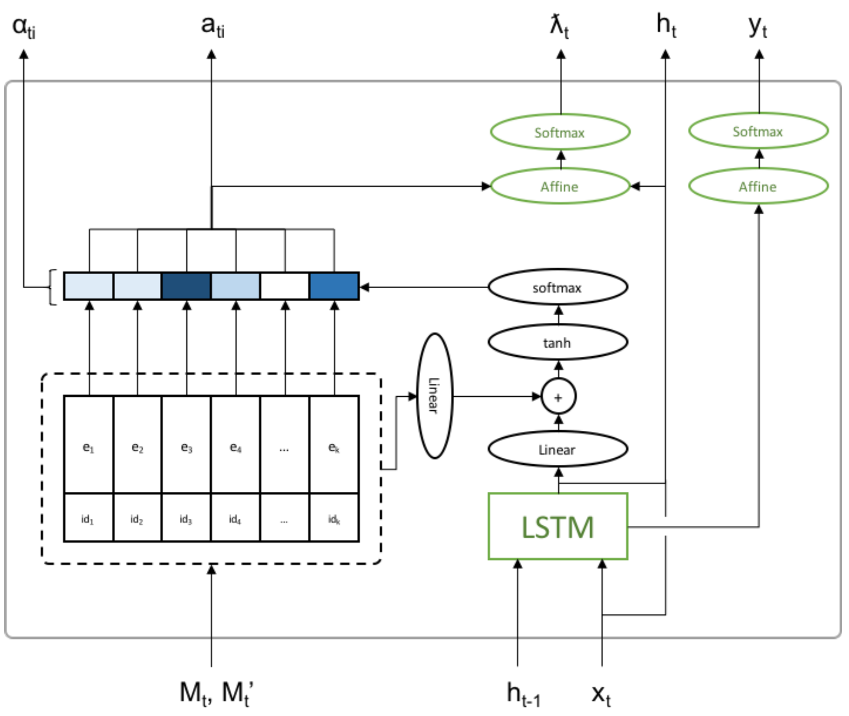


Figure 5.2: Illustration of the internal workings of the attention mechanisms

mined by the law of total probability as per equation 5.11. (Note again that the 0th task is considered to be the standard language model's prediction).

$$P(\tau|\text{context}_t) = (\mathbf{Y}_t)_\tau = \sum_{i=0}^{n} (\boldsymbol{\lambda}_t)_i P(\tau|\text{task} = i, \text{context}_t) \qquad id_\tau \in \{1,\ ...,\ |V|\} \tag{5.11}$$

The internal workings of a single task attention mechanism are illustrated in figure 5.2. The figure depicts a single attention task, however for multiple tasks, everything except the components illustrated in green will be replicated. Note that unlike the standard attention for language modelling, the attention mechanisms must now be considered separately from the LSTM, as the results are only combined in the very last stage of the language model process. The manner in which a task attention mechanism fits together with the language model is illustrated in figure 5.3.

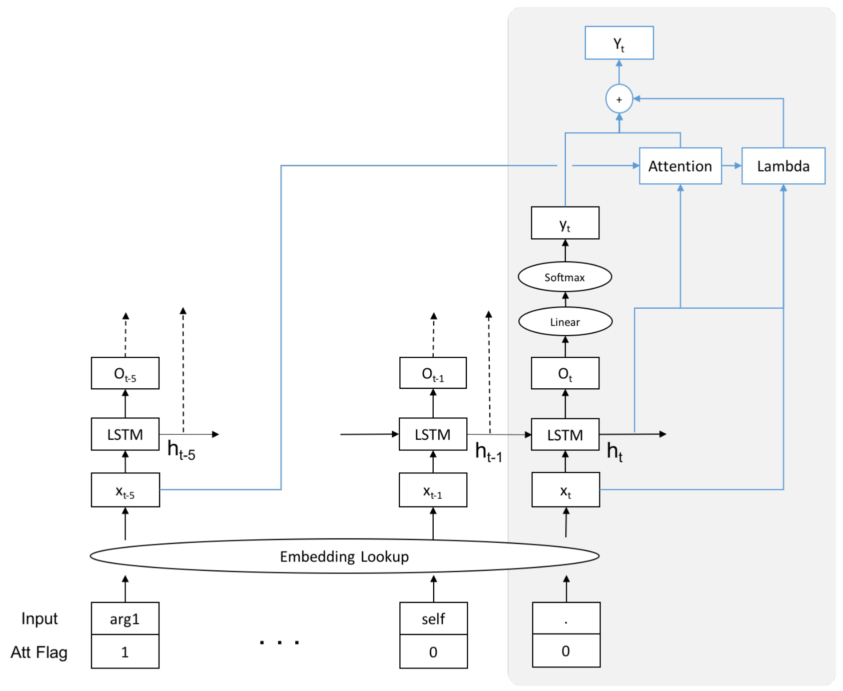


Figure 5.3: Illustration of the integration between a single task attention mechanism and a standard neural language model

5.4 Initial Attempts

The model of section 5.3.3 was initially designed to include the filtered attention mechanisms as part of the LSTM cell as it is in the standard attention model. The lambda weightings as calculated in equation 5.10 were then applied at the point of combining the LSTM output vector $\mathbf{o}_t$ with the attention output vectors $\mathbf{a}_{t1},...,\mathbf{a}_{tn}$ per equation 5.12. As in the standard attention model, this combined output vector then underwent the usual projection and softmax (equation 5.13) to derive the final output probability vector.

$$O'_t = \begin{bmatrix} \mathbf{o}_t & \mathbf{a}_{t1} & \cdots & \mathbf{a}_{tn} \end{bmatrix} \qquad \in \mathbb{R}^{s\ x\ (n+1)}$$

$$\mathbf{O}_t = O'_t \boldsymbol{\lambda}_t \qquad \in \mathbb{R}^{s} \tag{5.12}$$

$$\mathbf{y}_t = \text{softmax}(W_V \mathbf{O}_t + \mathbf{b}_V) \tag{5.13}$$

A number of variations of the model were implemented and tested with the identifier tagging scheme. All of these variations resulted in either the model learning to assign the entire λ weight to the language model and ignore the attention mechanism, or resulted in no improvement in perplexity over the standard LSTM language model. A short discussion of these results is provided in section 6.4.

For the embedding vectors that fed into the memory M, the use of input embeddings and output embeddings were tried. In addition to this, it was hypothesised that because the purpose of the output embedding is to encode the next token in the sequence, it may make more sense to split each LSTM output vector into two parts, one used for the standard language model process and the other used to drive the attention mechanisms. Different variations of the lambda state vector were also tried, including setting $\mathbf{h}_t^\lambda$ directly to $\mathbf{h}_t$, $\mathbf{x}_t$ or a projection of $\mathbf{a}_{t1},...,\mathbf{a}_{tn}$ and every combination thereof. Finally, different non-linear activation functions were included in the calculation of λ itself as well as a fixed uniform lambda to force the model to use the attention vector at every time step.

These many variations were tried due to the strong intuition that the model should have worked and should have been able to learn when to use the attention mechanism. It was thought that some detail in the equations may have been what prevented it from doing so. The revised model that removed the attention mechanism from the LSTM cell and combined the α vectors with the language model probability was inspired in part by the work of Gu et al. [60] who built a sequence to sequence copy mechanism. Their copy mechanism was capable of copying tokens directly from the encoders input sequence to the decoder's output sequence by combining decoder output probabilities with attention weights over the encoder's outputs.

5.5 Implementation Details

An implementation of the above model that could work with batched inputs was required for efficient implementation of batched SGD. This version is specified in informal notation below. All mathematical symbols carry the same meaning as in the previous section and *batch* refers to the mini-batch size. The t and i subscripts have been dropped to simplify the notation. Certain pseudo-code operations have also been added for simplicity, such as $\mathbb{R}^{a\ x\ b} \rightarrow \mathbb{R}^{c\ x\ d}$ which indicates a reshape operation. The batch. mul operation splits its 2 3D tensor arguments along the first dimension, performs a matrix multiplication of each corresponding matrix and then combines the result into a 3D output tensor. The operation tile(h) repeats the $\mathbb{R}^{batch\ x\ size}$ state matrix h, k times to form a matrix of size $\mathbb{R}^{(batch\ x\ k)\ x\ size}$

$$V' = \begin{bmatrix} id_{11} & id_{12} & \cdots & id_{1k} \\ id_{21} & id_{22} & \cdots & id_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ id_{b1} & id_{b2} & \cdots & id_{bk} \end{bmatrix} \qquad \in \mathbb{R}^{batch\ x\ k} \tag{5.14}$$

$$V = \begin{bmatrix} \mathbf{e}_{11} & \mathbf{e}_{12} & \cdots & \mathbf{e}_{1k} \\ \mathbf{e}_{21} & \mathbf{e}_{22} & \cdots & \mathbf{e}_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{e}_{b1} & \mathbf{e}_{b2} & \cdots & \mathbf{e}_{bk} \end{bmatrix} \qquad \in \mathbb{R}^{batch\ x\ k\ x\ size} \rightarrow \mathbb{R}^{(batch\ x\ k)\ x\ size} \tag{5.15}$$

$$\tag{5.16}$$

$$M = tanh(VW^V + tile(h)W^h) \qquad \in \mathbb{R}^{(batch\ x\ k)\ x\ size} \tag{5.17}$$

$$\alpha = softmax(Mw^T) \qquad \in \mathbb{R}^{(batch\ x\ k)\ x\ 1} \rightarrow \mathbb{R}^{batch\ x\ 1\ x\ k} \tag{5.18}$$

$$a_i = batch.mul(\alpha, V) \qquad \in \mathbb{R}^{batch\ x\ 1\ x\ size} \rightarrow \mathbb{R}^{batch\ x\ size} \tag{5.19}$$

$$\lambda = softmax(h_\lambda W^\lambda + tile(b^\lambda)) \qquad \in \mathbb{R}^{batch\ x\ (n+1)} \tag{5.20}$$

Chapter 6

Experiments and Analysis

A series of experiments were conducted in order to assess the effectiveness of deep neural language models on the code suggestion task for Python source code. In particular the use of a standard neural attention mechanism and the novel approach of chapter 5 were compared to standard neural and ngram models. This chapter describes the experimental setup and goes on to analyse the results using both quantitative and qualitative criteria.

6.1 Experimental Setup

6.1.1 Preprocessing

Experiments were run on both the raw and normalised datasets of chapter 4. Some additional preprocessing steps were applied in order to prepare the data for the experiments. Comments and blank lines were removed in both datasets and numerical constants were replaced with a special <num> token. Tokens which occurred fewer than 10 times in the raw and 5 times in the normalised training sets, as well as tokens in the validation and test sets that did not occur in the training set, were replaced with a special <OOV> (out of vocabulary) token. Finally, it was ensured that all identifier names were included in the vocabulary of the normalised dataset even if they were under the threshold. This resulted in comparable vocabulary sizes of 210 117 and 221 313 token types for the raw and normalised datasets respectively.

The preprocessing steps applied here were consistent with those done in previous work ([9], [48]).

6.1.2 N-gram baselines

While there was already a strong intuition based on the work of White et al. [48] that even the simplest neural language model would outperform an n-gram model on the code suggestion task, a number of n-gram model variations were included in the experiments as baselines.

The variations included n ranging from 3 to 6 and the smoothing technique being one of Kneser-Ney or Modified Kneser-Ney (MKN) smoothing. The models were named “3-gram”, “4-gram”, “5-gram” and “6-gram” for those using Kneser-Ney smoothing and “3-gram (MKN)”, “4-gram (MKN)”, “5-gram (MKN)” and “6-gram (MKN)” for those using Modified Kneser-Ney smoothing. All n-gram models were trained and evaluated using the Kyoto Language Modelling Toolkit [61].

6.1.3 Neural Language Models

The simplest neural language model used in this work, which can be considered a baseline for the attention models, consisted of a standard RNN language model with LSTM cells (referred to as “LSTM”). This model was trained with mini-batch SGD with a batch size of 75 and Truncated BPTT with a sequence length of 50. The gradient descent optimiser was initialised with a learning rate of 0.7 and this rate was decayed by a constant multiple of 0.9 after each training epoch. [1].

Two variations of the standard attention mechanism for language modelling described in section 5.1 were implemented. These variations, referred to as “LSTM+Att(20)” and “LSTM+Att(50)” used fixed-window attention memory sizes of 20 and 50 respectively but were otherwise identical. The large attention memory size of LSTM+Att(50) was used in the hope that it would be able to capture some medium-term dependencies. They were trained with a smaller initial learning rate of 0.2 due to numerical instability at higher learning rates.

The novel attention model of chapter 5, referred to as “LSTM+Att CS” was implemented using the single task identifier tagging scheme described in section 5.2. The attention memory size was set to 20. Due to memory constraints, it also

[1]A training epoch refers to a full iteration through the entire training dataset. Training a neural model generally involves multiple epochs

required a smaller batch size of 30 to train. Note also that this model was only run on the normalised dataset because it is was developed particularly for use with identifier references. The raw dataset had the majority of identifier references replaced with <OOV> tokens.

All neural models in this work were developed using TensorFlow [62] and trained on an NVidia Tesla K80 or Geforce GTX 1070 GPU. Python source code files were considered a single unit, and each was split into sequences of size 50. While processing a single file, the last recurrent state of the RNN was fed as the initial state of the subsequent sequence of the same file. The recurrent state was reset to zero after completing a single file. All models used embedding and hidden dimension sizes of 200, LSTM forget gate biases initialised to 1 ([63]), a threshold of 5 for gradient norms and had all other parameters initialised random uniformly in the range (-0.05, 0.05). To reduce the chance of overfitting, dropout was applied to the inputs at a rate of 0.1. Sampled softmax using a log-uniform sampling distribution and a sample size of 1000 was also employed to reduce the performance penalty of the fairly large vocabulary size. Unfortunately, due to the large amount of time it took to train the models, it was not practical to conduct a grid-search to determine optimal values for these hyper-parameters, but they were very similar to those used in previous work, including by Tran et al. [37].

6.1.4 Evaluation metrics

Three key metrics were chosen for quantitative evaluation of the models. The first metric was perplexity (sometimes abbreviated as PP) which is fairly standard for language models. Perplexity is an intrinsic metric that estimates the average number of tokens that the model thinks is likely at each step in the sequence. A more certain model will have lower perplexity. The metric is easily derived from the cross entropy loss of equation 2.13 using equation 6.1 where once again, $\mathbf{y}$ refers to the predicted distribution of the model at each step and $\mathbf{y}'$ is the corresponding target distribution, a one-hot encoding of the target token type.

$$\mathrm{PP} = 2^{\sum_{\mathbf{y},\mathbf{y}'} \varepsilon(\mathbf{y},\mathbf{y}')} \tag{6.1}$$

The next two metrics were top-1 and top-5 accuracies which are commonly used for recommendation systems. Top-1 accuracy measured the frequency with which the true target token type corresponded to the one the model assigned the largest probability. Top-5 accuracy measured the frequency with which the target token type occurred in the top 5 highest ranked token types of the model (ranked by probability).

These metrics (or variations of them) were also used in previous work ([1, 48, 46, 41])

6.2 Results and Analysis

The results for the n-gram baselines on the raw dataset and normalised dataset are summarised in table 6.1 and 6.2 respectively with a summary of the test perplexities in figure 6.1. It is clear that perplexity on the test set decreased with increasing n, but the rate of decrease seemed to slow as n increased. Similarly, the top 1 and top 5 accuracies (both calculated on the test set only) increased at a decreasing rate with n. It is expected that a minimum perplexity and maximum accuracy would be reached at n equal to 7 or 8 as found in previous work ([48]). It is also noteworthy that even though the modified Kneser-Ney smoothing technique performed slightly worse in some instances on the training set (for example the 6-gram model), it performed consistently better than Kneser-Ney smoothing on the test datasets for all n. The modified technique therefore generalised better on this dataset.

Neural language model results are presented in table 6.3 for the raw dataset and table 6.4 for the normalised. All models performed worse on the normalised dataset than they did on the raw dataset. In terms of perplexity, even the best performing model on the normalised dataset performed worse than the worst model on the raw dataset, but this trend was not as pronounced in the accuracy scores. On both datasets, adding an attention mechanism significantly improved the results

Table 6.1: N-gram baseline model results on raw dataset with OOV threshold of 10

Model	Train PP	Val PP	Test PP	Top 1 Acc	Top 5 Acc
3-gram	9.14	14.38	16.65	21.60%	54.11%
3-gram (MKN)	9.38	14.06	16.23	21.86%	58.69%
4-gram	5.65	11.94	14.61	24.40%	57.42%
4-gram (MKN)	5.85	11.55	13.99	24.03%	58.80%
5-gram	3.54	10.26	12.32	24.85%	58.08%
5-gram (MKN)	3.70	9.87	11.81	24.53%	59.82%
6-gram	2.62	9.44	11.10	24.63%	57.86%
6-gram (MKN)	2.75	9.06	**10.61**	24.51%	59.74%

Table 6.2: N-gram baseline model results on normalised dataset with OOV threshold of 5

Model	Train PP	Val PP	Test PP	Top 1 Acc	Top 5 Acc
3-gram	12.63	24.54	27.33	12.94%	50.12%
3-gram (MKN)	12.90	24.19	26.90	13.19%	50.81%
4-gram	7.40	21.50	24.43	13.68%	50.51%
4-gram (MKN)	7.60	21.07	23.85	13.68%	51.26%
5-gram	4.39	19.77	21.73	13.78%	50.83%
5-gram (MKN)	4.52	19.33	21.22	13.90%	51.49%
6-gram	3.26	19.18	20.68	14.65%	51.05%
6-gram (MKN)	3.37	18.73	**20.17**	14.51%	51.76%

on all metrics, and this was particularly the case on the normalised dataset where the LSTM+Att(20) model achieved a test perplexity score 2.27 points lower than the LSTM model and the LSTM+Att(50) model achieved almost 4 points lower, a very significant difference. This was also reflected in the top-1 accuracies which saw a roughly 3 and 6 percentage point improvement for the LSTM+Att(20) and LSTM+Att(50) models respectively. This trend was essentially the same for top-5 accuracy on the normalised dataset, but the top-5 accuracies did not increase as significantly on the raw dataset. This suggests that the highest top-5 accuracy of 85.73% may be close to the upper threshold achievable on the Python corpus.

The best performing model overall, on the normalised dataset, was the LSTM+Att CS model which achieved almost 1 point lower test perplexity, and 1 and a half percentage points higher top-5 accuracy than the LSTM+Att(50) model, but had a slightly lower top-1 accuracy. It significantly outperformed the

Table 6.3: Neural Language model results on raw dataset with OOV threshold of 10

Model	Train PP	Val PP	Test PP	Epoch	Top 1 Acc	Top 5 Acc
LSTM	5.15	7.36	7.75	15	60.57%	82.52%
LSTM + Att(20)	4.80	7.01	7.17	10	63.13%	83.80%
LSTM + Att(50)	4.39	5.92	**6.09**	11	**66.25%**	**85.73%**

Table 6.4: Neural Language model results on normalised dataset with OOV threshold of 5

Model	Train PP	Val PP	Test PP	Epoch	Top 1 Acc	Top 5 Acc
LSTM	9.29	13.08	14.01	15	57.91%	76.30%
LSTM + Att(20)	7.30	11.07	11.74	13	61.30%	79.32%
LSTM + Att(50)	7.09	9.83	10.05	12	**63.21%**	81.69%
LSTM + Att CS	6.41	9.40	**9.18**	7	62.97%	**82.62%**

LSTM+Att(20) model on all metrics, indicating that it made better use of its attention memory. It also outperformed the LSTM+Att(50) model despite having less than half the attention memory available to it.

It is not surprising that even the worst neural model performed better than the best n-gram model by a large margin on all metrics. The difference amounted to almost 3 points in Test perplexity on the raw dataset and a large 6 points on the normalised dataset. The differences were even more striking in the accuracy scores. It is worth noting that the n-gram models tested here were not as powerful as the models of Tu et al. [41], but even those models were outperformed in White et al.'s work [48] by simpler RNNs than the LSTM model presented here. Another interesting observation is that the 4-gram models achieved training set perplexities comparable to even the attention models. Larger values of n resulted in lower training perplexities than the neural models. This suggests that the neural models were able to generalise better than the n-gram models and that the n-gram models may be more prone to overfitting.

In order to understand why the models performed worse on the normalised dataset than on the raw, it is worth noting the key difference between the two datasets. By replacing the almost a million unique identifiers noted in chapter 4 with a much smaller vocabulary of randomised identifier names, the normalised

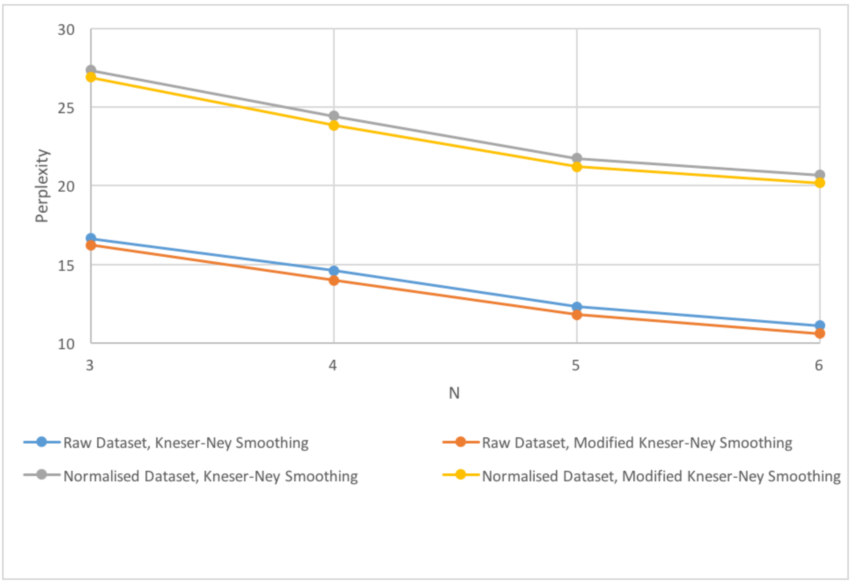


Figure 6.1: Perplexity of the N-gram models on test datasets

dataset ensured that all identifier names were included in the vocabulary on which the models were run. When run on the raw dataset, the majority of identifier names fell outside the vocabulary threshold of 10 and were replaced with OOV tokens. The models run on the raw dataset could therefore suggest the OOV token whenever any uncommon identifier name was called for and be correct in the vast majority of those cases. The models run on the normalised dataset did not have this advantage and were forced to suggest the correct identifier name, particularly when an already-declared identifier was referenced. In the test set, around 9% of tokens were references to identifiers previously declared in the file. By flagging these identifier references, it was possible to split the accuracy figures into accuracies on identifier references and other tokens. The results of this analysis are presented in table 6.5. All models achieved significantly higher accuracies on other tokens than they did on identifier references. The result explains why the models performed worse on the normalised dataset than they did on the raw. The particularly poor performance of the LSTM model on identifier references also provided a strong motivation for the introduction of mechanisms that could better equip the model to deal with the long-term dependencies present in identifiers. The table also shows that the vast majority of the improvement in performance of the attention models over the LSTM

Table 6.5: Accuracies on the normalised dataset split between identifier references and other tokens

Model	Top 1 Acc		Top 5 Acc	
	Identifiers	Other	Identifiers	Other
LSTM	2.1%	62.8%	4.5%	82.6%
LSTN + Att (20)	21.4%	64.8%	29.9%	83.7%
LSTM + Att (50)	30.2%	65.3%	41.3%	84.1%
LSTM + Att CS	27.3%	64.9%	43.6%	84.5%

model came from a significant improvement in performance on identifier reference suggestions.

As a final point, the top 5 accuracy score of 82.62% achieved by the LSTM+Att CS model on the normalised dataset bodes very well for a real code suggestion implementation in an IDE. It indicates that the vast majority of the time, the actual token used by the developer was in the list of the top 5 suggestions offered by the model. Of course real code suggestion systems do not offer suggestions for every single token, so by combining this with heuristics such as when to invoke the model and a threshold on the models certainty, a very accurate code suggestion system could be built.

6.3 Qualitative Analysis

This section presents an analysis of some qualitative test cases in order to gain a better insight into the differences between the models and also a better intuition of why the attention models performed better than the LSTM model. The focus of this section is on the neural models only, as n-gram models have already been analysed in past work and were shown once again in the previous section to be inferior to the neural models on the task of code suggestion. The LSTM+Att(20) model was also excluded from this analysis as it was identical to the LSTM+Att(50) model other than the smaller attention memory.

6.3.1 Attention

Figure 6.2 shows a visualisation of the attention weights of the LSTM+Att(50) model on the simple Python source code file of listing 6.1, in order to get a better insight into the differences between the attention models and the LSTM model. The token corresponding to each step of the input sequence is presented on the y-axis and the top 10 largest weights assigned in the attention memory are presented in the grid. A darker shade of blue indicates a higher weight. Tokens which the model predicted successfully (as the first suggestion after reading the previous token) have a ** next to them and interesting steps in the sequence are underlined and tagged in red.

Listing 6.1: Code example used for attention visualisation

```python
import os


class Class253:
    def __init__(self, arg651):
        self.attribute943 = arg651

    def function1690(self, arg2004):
        var4040 = os.path.join(self.attribute943, arg2004)
        with open(var4040, 'r') as var2496:
            var3334 = var2496.readlines()
        return var3334


var25 = Class253('/home/myuser')
var1200 = var25.function1690('myfile.txt')
print(len(var1200))
```

The step tagged as 1 shows the attention weights after reading the “=” token in the line `self.attribute943 = arg651`. The model goes on to suggest the argument `arg651` correctly, which the LSTM model did not do on the same input (it suggested “`[`”]). The visualisation shows that the majority of the attention weight was on the output vector corresponding to the input token “`.`” after `self.`. While this does not immediately make intuitive sense, it must be the case that the output vector at this step contained some information that the model had captured from the

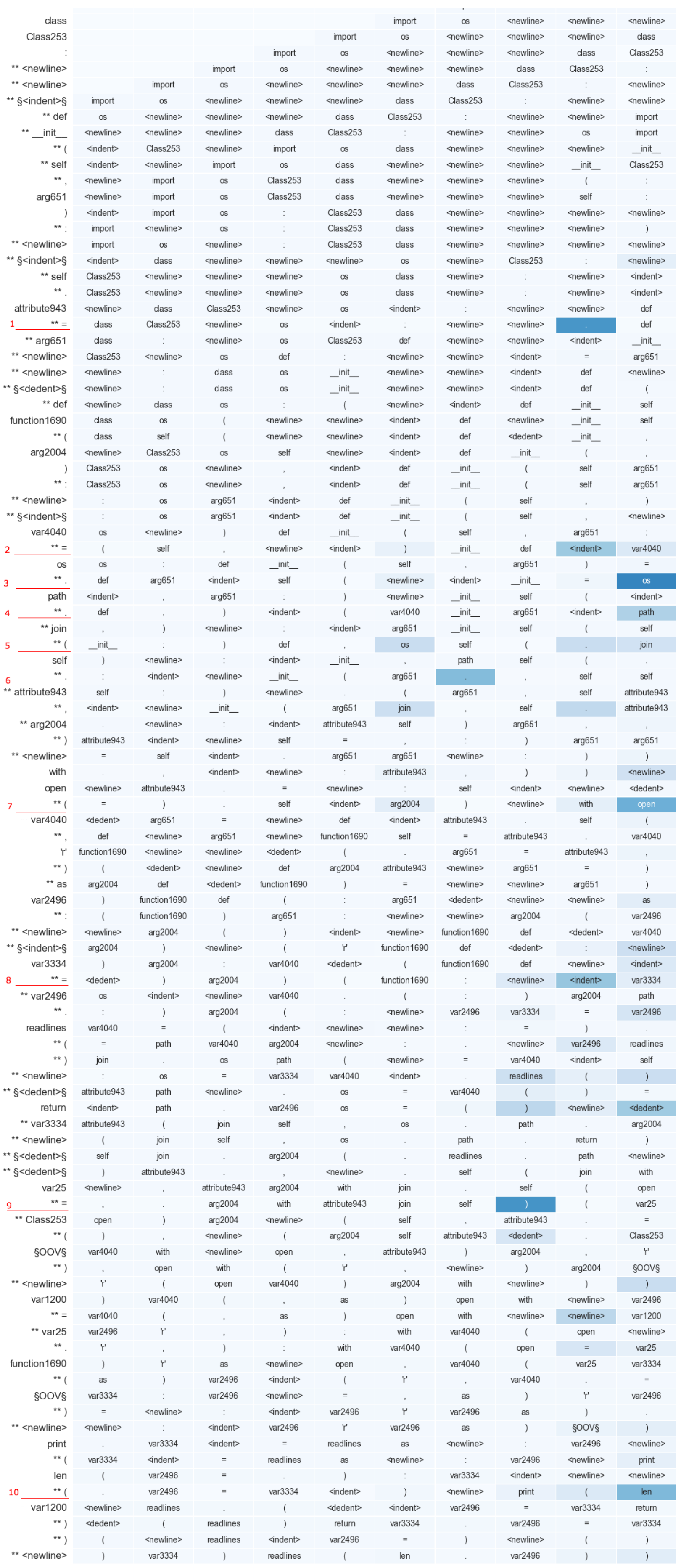


Figure 6.2: A visualisation of the attention weights of the LSTM+Att(50) model on a normalised code sample

`arg651` declaration of the previous line. This information was likely stored in the LSTM cell memory, was output after reading "`.`" and was used to bias the model's suggestion to the correct identifier name. A similar unintuitive situation is observed in the step tagged 6, where the model successfully suggested a class attribute (which again the LSTM model failed to do), by placing attention on the "`.`" token between `path` and `join` of the same line.

Cases 3, 4 and 5, seemed at first to be more intuitive. Here the model placed a large attention weight on the preceding API or function name. This seemed to suggest that the model used the name of the API or function as a clue to what might follow. While it did not correctly suggest `path` following `os.` as the top suggestion, it did correctly suggest `join`. However, the top suggestion it made following `os.` was `var4040`, which does not make sense. Furthermore, the LSTM model actually suggested this entire snippet correctly. In this case the attention weights, which at first glance seemed to be more intuitive, turned out to hamper the model.

The step tagged 8 shows the weights in another instance where the model correctly predicted an identifier reference that the LSTM model did not. The weights were again on the seemingly unintuitive preceding `indent` token. Again in step 9 a large weight was placed on the ")" token when the correct class name was suggested. One possible explanation for this behaviour is to do with the dual-purpose nature of the output vectors at each step. The output vector needs to encode sufficient information to inform the model which token to suggest at a particular step and it also needs to encode information that will be useful to the attention mechanism in later steps. It is possible that the model keeps certain relevant information in its cell memory and avoids outputting this information until signalled in some way by a relatively common input like a period or bracket.

Finally, step 9 shows an example of a long-range dependency, where the class name Class253 was declared 74 tokens before being referenced. This is outside the attention window, but the model was still able to suggest it correctly. Again the LSTM model did not suggest this class name correctly. This shows an example of attention mechanisms improving upon an LSTMs ability to deal with long range dependencies.

The attention weights of the LSTM+Att CS model are presented in figure 6.3. Here, the filtered attention window is presented along with the weights the model assigned to the Language Model or Attention mechanism. The attention weights were scaled by the weight assigned to the attention mechanism. Interesting steps are again highlighted with red lines and tagged with a number (which may not correspond to the previous figure). The first step of interest is tagged 1 and shows that the model chose to place a large weight on the attention mechanism following the snippet `self.attribute943=`. Furthermore, it placed a large attention weight on the only argument to the function, `arg651` which turned out to the be the correct suggestion. Tag 3 shows another case where the model placed weight on the attention mechanism at a very appropriate point, after the "`.`" following `self.`. It also placed the vast majority of the attention weight on `attribute943`, ignoring the other non-attribute tokens in its memory.

Step 8 shows an example where the LSTM+Att CS model was unable to successfully suggest a class name, which the LSTM+Att(50) model was able to do. Step 6 also shows a case where the weight placed on the attention mechanism was unexpected because the token following `as` would generally be a new identifier rather than a reference to an existing one.

Overall, a glance at the LM/Att column shows that the model learned to place weight on the attention mechanism at very appropriate and intuitive places where one would expect an identifier reference to follow. These were following an =, following `self.`, the opening bracket of a function call as well as the comma separating the arguments of the function call and following a `return`. In the majority of other cases, the model fell back to the suggestions of the standard language model, which was already very good at suggesting Python syntax, code structure and API references (this point is explored further in the next section). This allowed the model to correctly suggest `path` following tag 2, which the LSTM+Att(50) did not. It is also clear to see how the weights applied to the attention memory directly correspond to the suggestions made by the model.

Overall, the suggestions made by the 2 models on this particular example were very similar, with the LSTM+Att(50) suggestions perhaps being slightly better by capturing the long-term dependency of the class name. Of course no conclusion can be drawn in this respect from this single example, and the situation was reversed in other test cases involving long-range dependencies, one of which is presented in the next section. What is fairly clear from this example though is that the attention weights of the LSTM+Att CS model are far more intuitive and interpretable than those of the LSTM+Att models.

6.3.2 Code Suggestion Evaluation

This section presents a small sample of interesting code suggestion test cases that highlight the similarities and differences between the various neural models.

Figure 6.4 shows sample predictions involving language syntax or structure from the LSTM model on the raw dataset. In each example, the code context is provided, followed by a box indicating the top 5 predictions of the model and their respective probabilities. In cases where a multi-token suggestion would be useful, the most likely sequence of tokens, as determined by a beam search [64], is presented in bold. The results for the other neural models on these examples were near identical (except of course with identifiers not being out of vocabulary in the normalised dataset) and are therefore not presented. It is interesting to note that the models correctly tracked levels of indentation and suggested the correct number of `dedent` tokens in examples (a) and (c). The models also learned that functions take `self` as their first argument when inside a class, but not when outside a class as seen in examples (d) and (e). Finally, the models were aware that an except must come after a try in example (f). Also of note in example (f) is that the models suggested the correct exception type raised by `shutil.rmtree`. Neural language models, including even the simplest version evaluated here, are clearly very well suited to learning Python syntax and code structure

Suggestions for examples generated by the LSTM model on the raw dataset, illustrating a few cases of common API usage scenarios are shown in figure 6.5. Of particular interest is the suggestion of the correct function, read or write in the

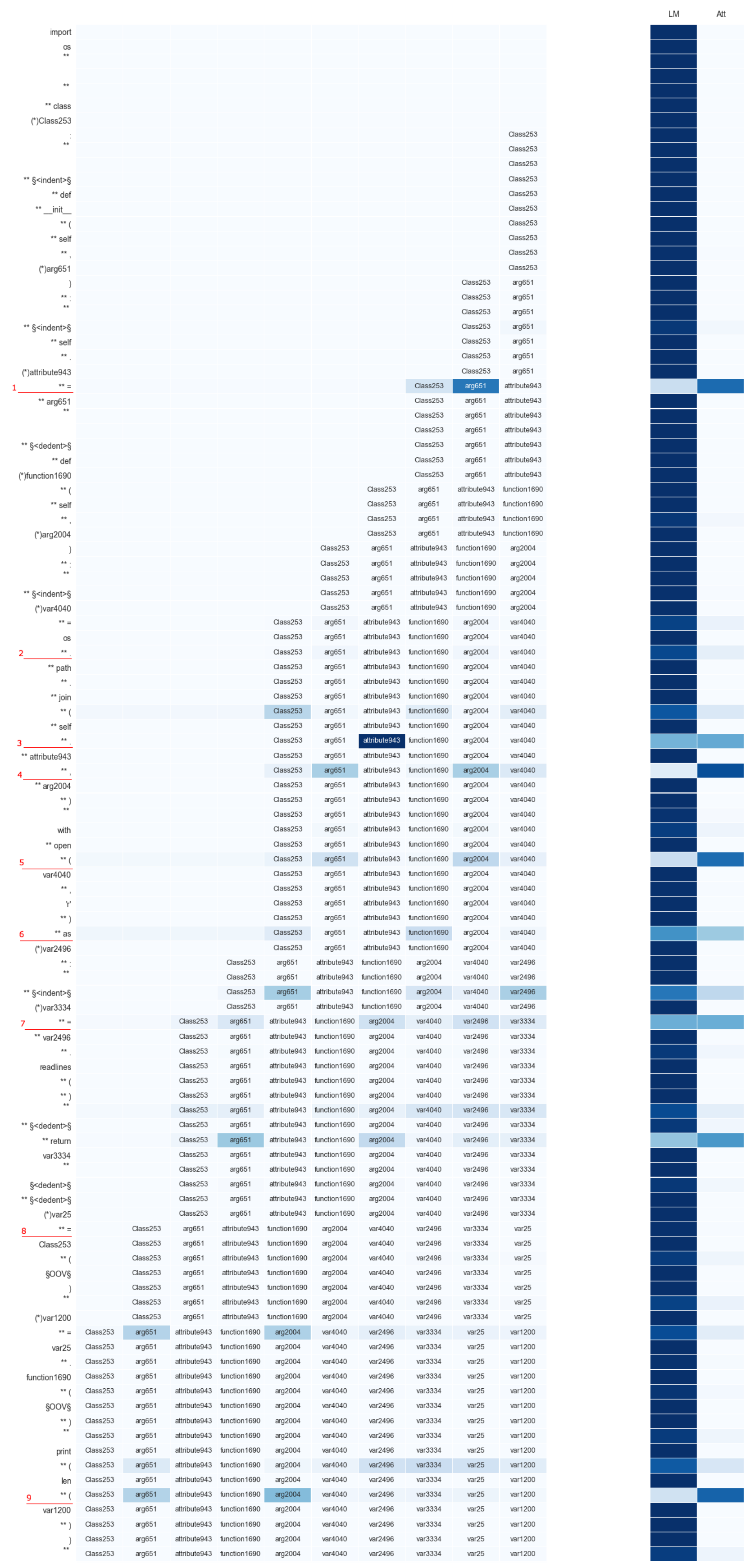
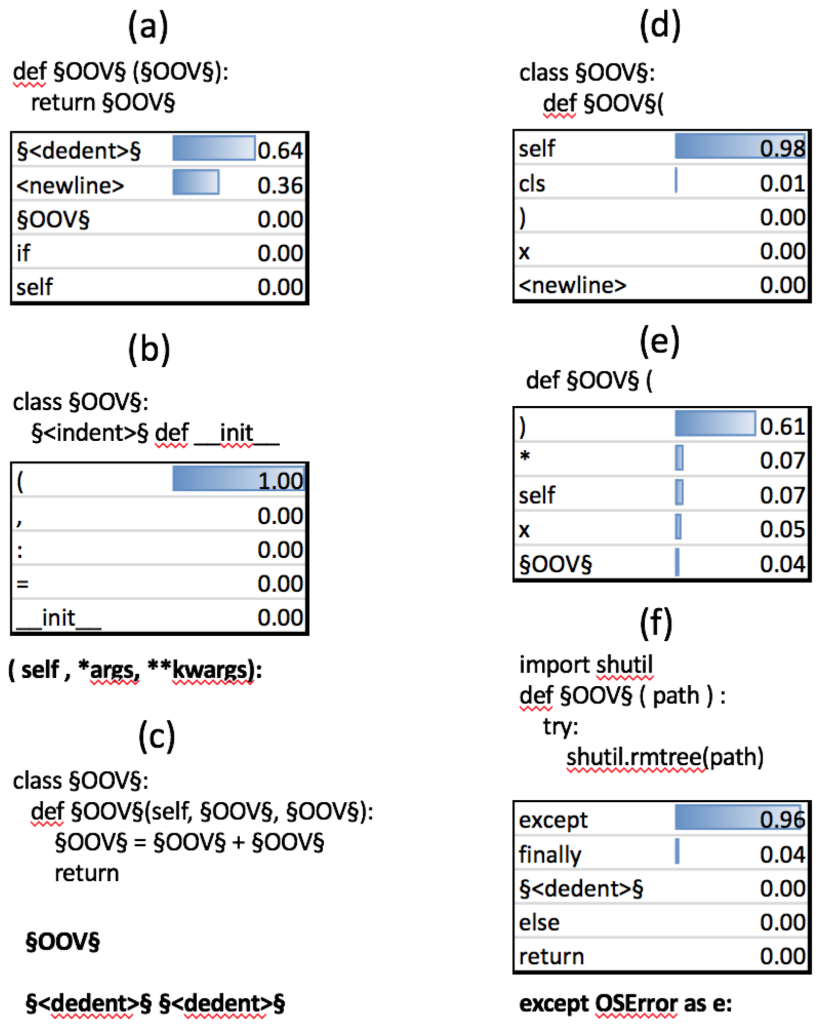


Figure 6.3: A visualisation of the attention weights of the LSTM+Att CS model on a normalised code sample



(a)

```
def §OOV§ (§OOV§):
  return §OOV§
```

§<dedent>§	0.64
<newline>	0.36
§OOV§	0.00
if	0.00
self	0.00

(b)

```
class §OOV§:
  §<indent>§ def __init__
```

(	1.00
,	0.00
:	0.00
=	0.00
__init__	0.00

(self , *args, **kwargs):

(c)

```
class §OOV§:
  def §OOV§(self, §OOV§, §OOV§):
    §OOV§ = §OOV§ + §OOV§
    return
```

§OOV§

§<dedent>§ §<dedent>§

(d)

```
class §OOV§:
  def §OOV§(
```

self	0.98
cls	0.01
)	0.00
x	0.00
<newline>	0.00

(e)

```
def §OOV§ (
```

)	0.61
*	0.07
self	0.07
x	0.05
§OOV§	0.04

(f)

```
import shutil
def §OOV§ ( path ) :
  try:
    shutil.rmtree(path)
```

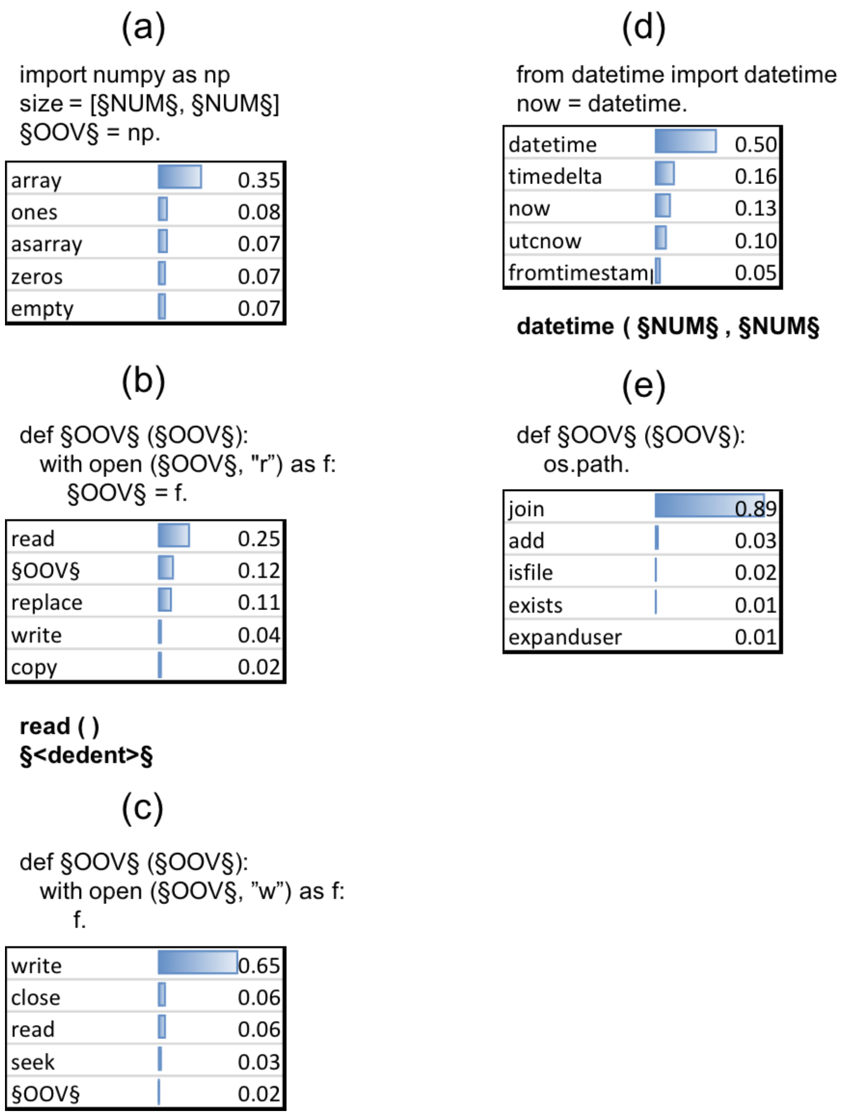
except	0.96
finally	0.04
§<dedent>§	0.00
else	0.00
return	0.00

except OSError as e:

Figure 6.4: A sample of suggestions made by all neural models involving Python syntax or structure

file IO examples (b) and (c). Suggestions by other models and on the normalised dataset were mostly the same except for the 2 examples in figure 6.6 for the LSTM model on the normalised dataset. The difference between example (a) in this figure and example (c) in figure 6.5 seems to suggest that the identifier name `f` was a strong signal to the model which could be exploited on the raw dataset but not on the normalised one. The suggestion in figure 6.5 makes more sense as it is unlikely that a file would be closed immediately without doing anything with it. The LSTM+Att(50) model did fix this problem as illustrated in figure 6.7. The figure also shows that the attention models were also much more certain about their suggestions of read and write for the two file IO examples. It was expected that the attention weights would indicate a large amount of attention being placed on the 'r' and 'w' tokens, which would explain the difference in certainty of the suggestions. Unfortunately an analysis of the attention weights for these particular examples revealed that the attention weights were mostly spread out, with the largest being on the `as` token. As noted in the previous section, the attention weights are difficult to interpret. It is possible that the output vector at the `as` token captured some useful information from the preceding 'w' and 'r' tokens which allowed the model to improve its suggestions.

The next set of test cases involved identifiers or identifier references in some form. Unsurprisingly given the quantitative analysis of the previous section, these cases provided the most variation in suggestions between the models. The suggestions made by the LSTM model on the raw and normalised datasets are presented in figure 6.8 and 6.9 respectively. Cases (a) and (b) show clearly why the models performed better on the raw dataset than the normalised one. The majority of unique identifiers were out of vocabulary in the raw dataset, allowing the model to correctly suggest the OOV token in these cases, which while technically correct, would not be useful in a real usage scenario. On the normalised dataset, the models were forced to suggest an existing identifier name, which the LSTM model was incapable of doing on these examples. Cases (c), (e) and (f) investigate whether the model was capable of generating unique identifiers. This is not something one would generally expect of a code suggestion system and was therefore not taken into consideration in the design of the models. The LSTM model did however make the useful suggestion of `__init__` in case (c) and suggested the very commonly used pattern of using `i` as a loop index in the for loop in case (e) on the raw dataset. The model was able to complete the pattern on both datasets, suggesting the multi-token snippet `var741 in range (NUM):` on the normalised dataset for example. Finally cases (d) and (i) show common patterns of identifier naming which the model was able to learn on the raw dataset. On the normalised dataset, the model was not surprisingly unable to suggest a novel identifier in case (d) and was not able refer to the argument in


(a)

```
import numpy as np
size = [§NUM§, §NUM§]
§OOV§ = np.
```

array	0.35
ones	0.08
asarray	0.07
zeros	0.07
empty	0.07

(b)

```
def §OOV§ (§OOV§):
   with open (§OOV§, "r") as f:
       §OOV§ = f.
```

read	0.25
§OOV§	0.12
replace	0.11
write	0.04
copy	0.02

read ( )
§<dedent>§

(c)

```
def §OOV§ (§OOV§):
   with open (§OOV§, ”w”) as f:
       f.
```

write	0.65
close	0.06
read	0.06
seek	0.03
§OOV§	0.02

(d)

```
from datetime import datetime
now = datetime.
```

datetime	0.50
timedelta	0.16
now	0.13
utcnow	0.10
fromtimestam	0.05

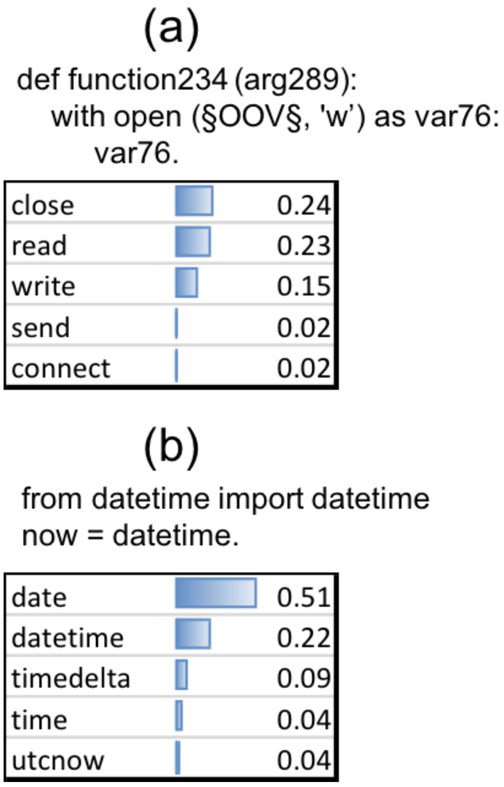
datetime (§NUM§ , §NUM§

(e)

```
def §OOV§ (§OOV§):
    os.path.
```

join	0.89
add	0.03
isfile	0.02
exists	0.01
expanduser	0.01

Figure 6.5: A sample of suggestions made by the LSTM model on the raw dataset involving common APIs



(a)

```
def function234 (arg289):
    with open (§OOV§, 'w') as var76:
        var76.
```

close	0.24
read	0.23
write	0.15
send	0.02
connect	0.02

(b)

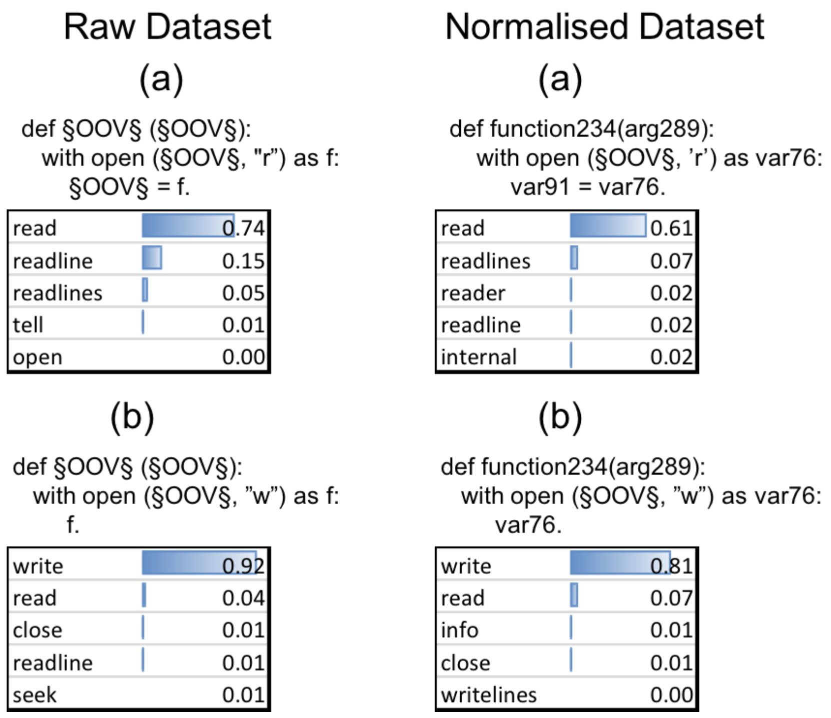
```
from datetime import datetime
now = datetime.
```

date	0.51
datetime	0.22
timedelta	0.09
time	0.04
utcnow	0.04

Figure 6.6: Differences in suggestions made by the LSTM model on the normalised dataset involving common APIs

case (i). The difference between the suggestions of the same model on the different datasets in case (i) suggests that the success of the model on the raw dataset was more the result of learning a very common pattern from the training data rather than learning to reference previously declared identifiers.

The suggestions made by the LSTM+Att(50) (and the LSTM+Att(20)) model on the raw dataset were nearly identical and are not presented here, except to note that the model was more certain about its suggestion in cases (b) and (c), and a lot more certain in case (d). The suggestions of the LSTM+Att(50) model on the normalised dataset are presented in figure 6.10. Cases (a), (b) and (i) show the correct suggestion of recently declared identifiers. The model was capable of learning short-term dependencies in identifiers. Strangely, the model did not suggest `__init__` in case (c), which the LSTM model did and it was also not able to suggest the very common loop pattern in case (e). It was expected that the model would



Raw Dataset (a)

```
def §OOV§ (§OOV§):
   with open (§OOV§, "r") as f:
      §OOV§ = f.
```

read	0.74
readline	0.15
readlines	0.05
tell	0.01
open	0.00

Normalised Dataset (a)

```
def function234(arg289):
   with open (§OOV§, 'r') as var76:
      var91 = var76.
```

read	0.61
readlines	0.07
reader	0.02
readline	0.02
internal	0.02

Raw Dataset (b)

```
def §OOV§ (§OOV§):
   with open (§OOV§, "w") as f:
      f.
```

write	0.92
read	0.04
close	0.01
readline	0.01
seek	0.01

Normalised Dataset (b)

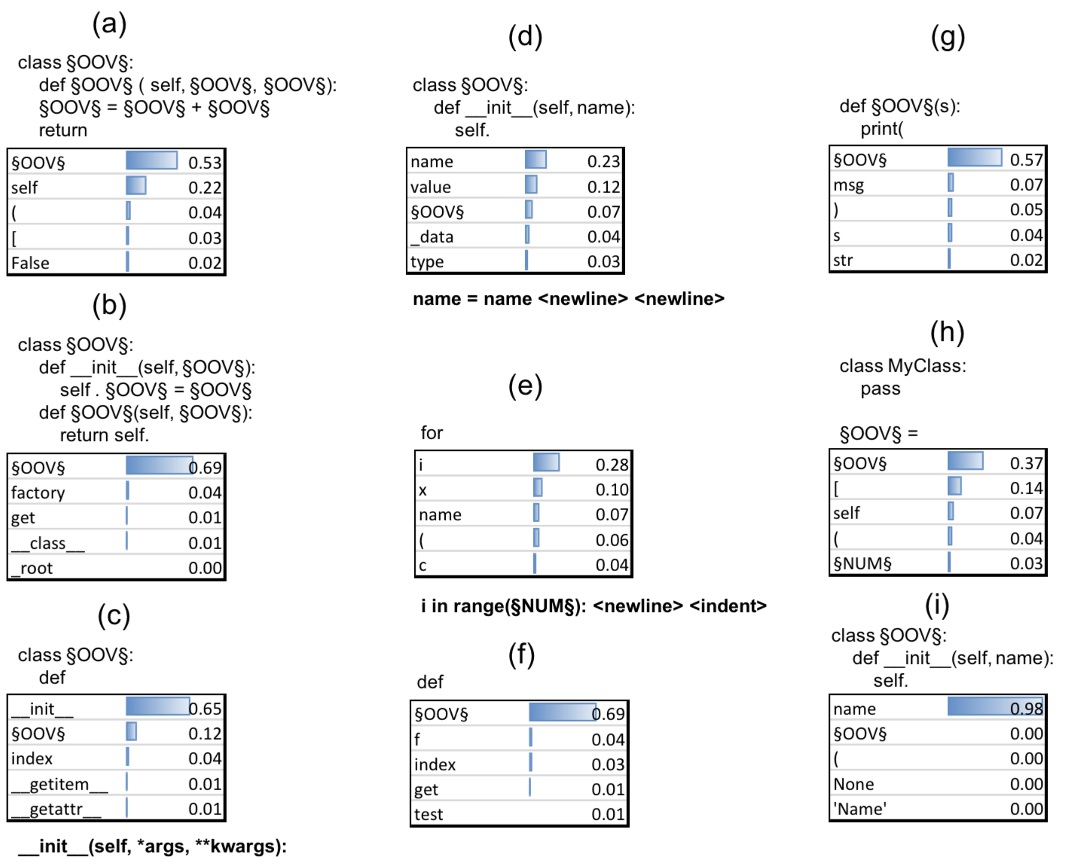
```
def function234(arg289):
   with open (§OOV§, "w") as var76:
      var76.
```

write	0.81
read	0.07
info	0.01
close	0.01
writelines	0.00

Figure 6.7: File IO suggestions made by the LSTM+Att(50) model on the raw and normalised datasets

refer to the class name in case (h) as it did in a more complex setting with the attention weight example of section 6.3.1, however in this particular case the model was unable to do so.

Figure 6.11 shows the suggestions made by the LSTM+Att CS model on the same set of the examples. The model made the correct suggestions in cases (a) and (b), albeit with less certainty than the LSTM+Att(50) model in case (a). In case (c) it suggested `__init__`, the same as the LSTM model and an improvement over the LSTM+Att(50) model. Cases (d) and (f) show that the model was unable to generate novel identifiers, just like the other models. Strangely the LSTM+Att CS model was also not able to complete the for loop pattern even though it could fall back to essentially the same model as the LSTM when not using the attention mechanism. The multi-token suggestion of case (g) was particularly interesting and showed a very appropriate and common pattern where a format string (in this case out of vocabulary) is used with a parameter after a `%` token. None of the other models suggested this pattern.



```
class §OOV§:
    def §OOV§ ( self, §OOV§, §OOV§):
    §OOV§ = §OOV§ + §OOV§
    return
```

§OOV§	0.53
self	0.22
(	0.04
[	0.03
False	0.02

```
class §OOV§:
    def __init__(self, §OOV§):
        self . §OOV§ = §OOV§
    def §OOV§(self, §OOV§):
        return self.
```

§OOV§	0.69
factory	0.04
get	0.01
__class__	0.01
_root	0.00

```
class §OOV§:
    def
```

__init__	0.65
§OOV§	0.12
index	0.04
__getitem__	0.01
__getattr__	0.01

__init__(self, *args, **kwargs):

```
class §OOV§:
    def __init__(self, name):
        self.
```

name	0.23
value	0.12
§OOV§	0.07
_data	0.04
type	0.03

name = name <newline> <newline>

```
for
```

i	0.28
x	0.10
name	0.07
(	0.06
c	0.04

i in range(§NUM§): <newline> <indent>

```
def
```

§OOV§	0.69
f	0.04
index	0.03
get	0.01
test	0.01

```
def §OOV§(s):
    print(
```

§OOV§	0.57
msg	0.07
)	0.05
s	0.04
str	0.02

```
class MyClass:
    pass

§OOV§ =
```

§OOV§	0.37
[	0.14
self	0.07
(	0.04
§NUM§	0.03

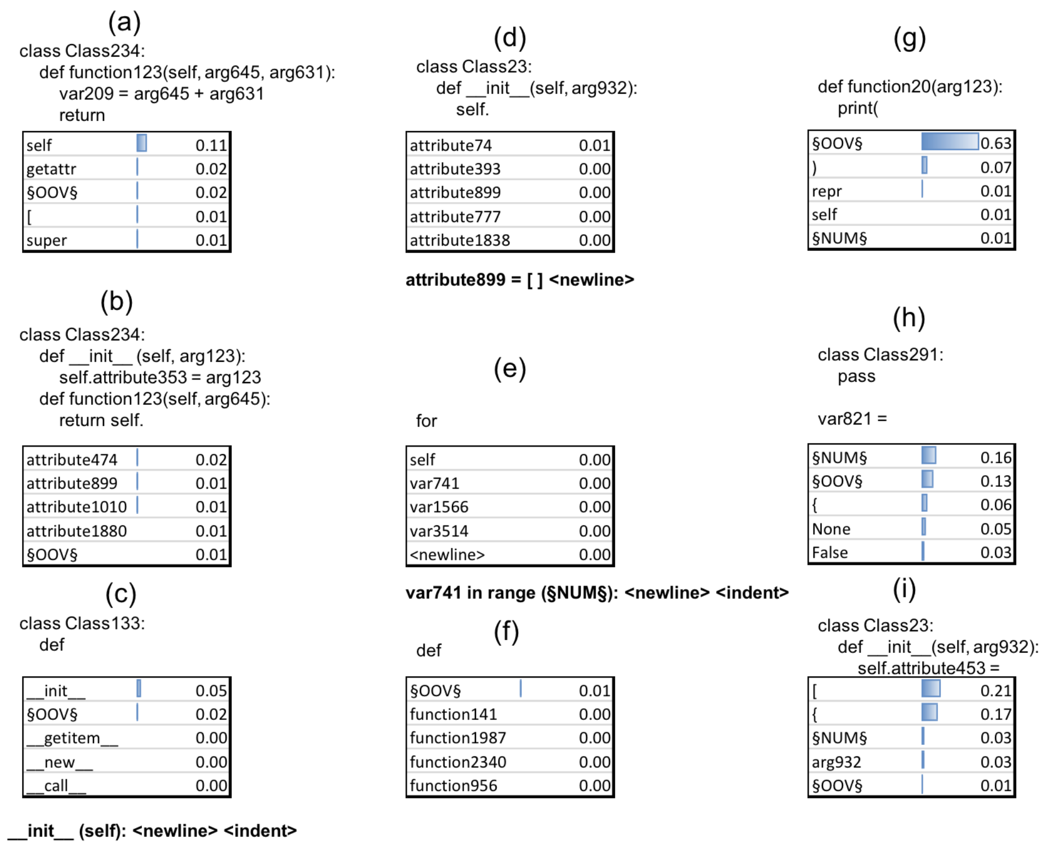
```
class §OOV§:
    def __init__(self, name):
        self.
```

name	0.98
§OOV§	0.00
(	0.00
None	0.00
'Name'	0.00

Figure 6.8: A sample of suggestions involving identifiers made by the LSTM model on the raw dataset

On the whole, the identifier examples show that the LSTM model was unable to refer to previously declared identifiers. The addition of an attention mechanism either in the form of the standard language model attention or the new filtered attention for code suggestion allowed for references to identifiers in the nearby context. This explains to a large degree the improvement in performance of the LSTM+Att models and LSTM+Att CS as compared to the LSTM model.

A major motivating factor for the introduction of the LSTM+Att CS model was to deal with the long range dependencies that exist in source code. Figure 6.12 shows the suggestions made by the LSTM and LSTM+Att (50) models on both the raw and normalised datasets on an example of a type of long range dependency that is very common in source code, that of a reference to a previously declared class attribute. Both models suggested OOV for the raw dataset, which may be


(a)

```
class Class234:
    def function123(self, arg645, arg631):
        var209 = arg645 + arg631
        return
```

self	0.11
getattr	0.02
§OOV§	0.02
[	0.01
super	0.01

(b)

```
class Class234:
    def __init__(self, arg123):
        self.attribute353 = arg123
    def function123(self, arg645):
        return self.
```

attribute474	0.02
attribute899	0.01
attribute1010	0.01
attribute1880	0.01
§OOV§	0.01

(c)

```
class Class133:
    def
```

__init__	0.05
§OOV§	0.02
__getitem__	0.00
__new__	0.00
__call__	0.00

__init__ (self): <newline> <indent>

(d)

```
class Class23:
    def __init__(self, arg932):
        self.
```

attribute74	0.01
attribute393	0.00
attribute899	0.00
attribute777	0.00
attribute1838	0.00

attribute899 = [ ] <newline>

(e)

```
for
```

self	0.00
var741	0.00
var1566	0.00
var3514	0.00
<newline>	0.00

var741 in range (§NUM§): <newline> <indent>

(f)

```
def
```

§OOV§	0.01
function141	0.00
function1987	0.00
function2340	0.00
function956	0.00

(g)

```
def function20(arg123):
    print(
```

§OOV§	0.63
)	0.07
repr	0.01
self	0.01
§NUM§	0.01

(h)

```
class Class291:
    pass

var821 =
```

§NUM§	0.16
§OOV§	0.13
{	0.06
None	0.05
False	0.03

(i)

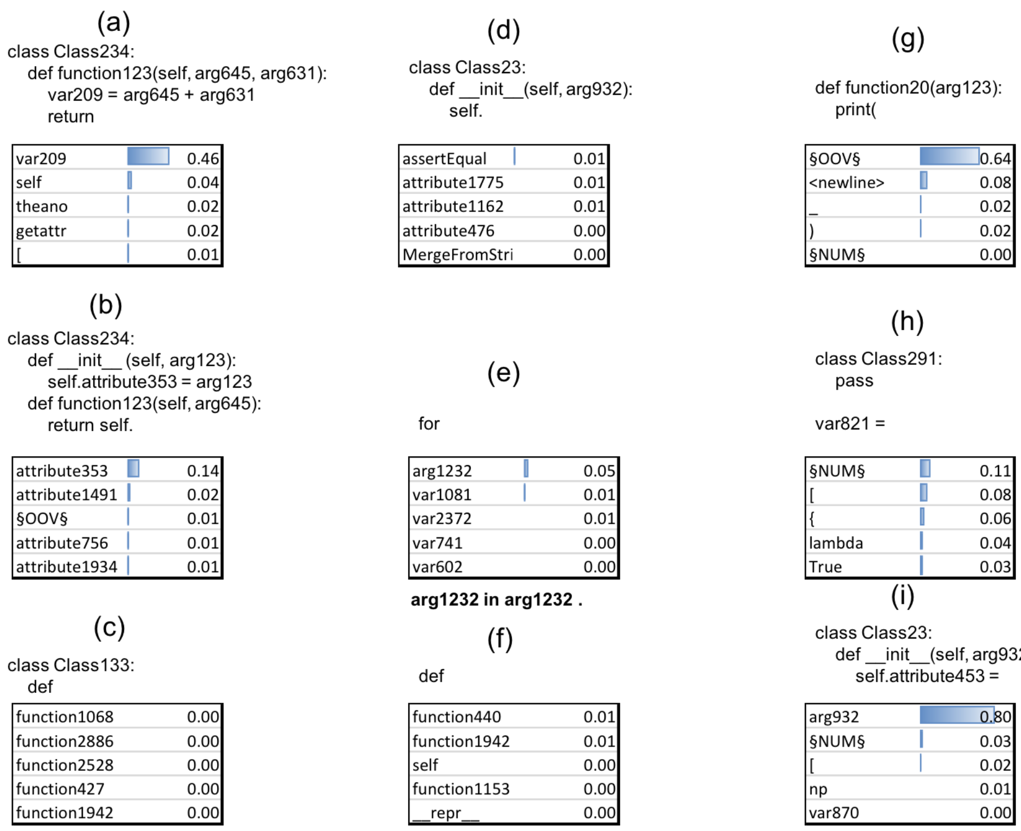
```
class Class23:
    def __init__(self, arg932):
        self.attribute453 =
```

[	0.21
{	0.17
§NUM§	0.03
arg932	0.03
§OOV§	0.01

Figure 6.9: A sample of suggestions involving identifiers made by the LSTM model on the normalised dataset

technically correct, but is meaningless, while the models were completely uncertain of any suggestion on the normalised dataset. The distance of the dependency, while outside the attention window of the LSTM+Att(50) model, is certainly within range of real code as was shown in figure 4.2 of chapter 4, so one would reasonably expect a code suggestion system to get these suggestions right. In fact the IntelliJ IDE does correctly suggest the only attribute on the class as the most likely next token (see figure 6.13).

Figure 6.14 shows that the LSTM+Att CS model did correctly suggest the class attribute as being most likely. Figure 6.14 also shows the attention memory and their weights for the input token "`.`", as well as the weights assigned to the language model and attention mechanism. The model not only knew that an identifier was likely as evidenced by the bias towards the attention mechanism, but also that the



(a)

```
class Class234:
    def function123(self, arg645, arg631):
        var209 = arg645 + arg631
        return
```

var209	0.46
self	0.04
theano	0.02
getattr	0.02
[	0.01

(b)

```
class Class234:
    def __init__(self, arg123):
        self.attribute353 = arg123
    def function123(self, arg645):
        return self.
```

attribute353	0.14
attribute1491	0.02
§OOV§	0.01
attribute756	0.01
attribute1934	0.01

(c)

```
class Class133:
    def
```

function1068	0.00
function2886	0.00
function2528	0.00
function427	0.00
function1942	0.00

(d)

```
class Class23:
    def __init__(self, arg932):
        self.
```

assertEqual	0.01
attribute1775	0.01
attribute1162	0.01
attribute476	0.00
MergeFromStri	0.00

(e)

```
for
```

arg1232	0.05
var1081	0.01
var2372	0.01
var741	0.00
var602	0.00

arg1232 in arg1232 .

(f)

```
def
```

function440	0.01
function1942	0.01
self	0.00
function1153	0.00
__repr__	0.00

(g)

```
def function20(arg123):
    print(
```

§OOV§	0.64
<newline>	0.08
_	0.02
)	0.02
§NUM§	0.00

(h)

```
class Class291:
    pass

var821 =
```

§NUM§	0.11
[	0.08
{	0.06
lambda	0.04
True	0.03

(i)

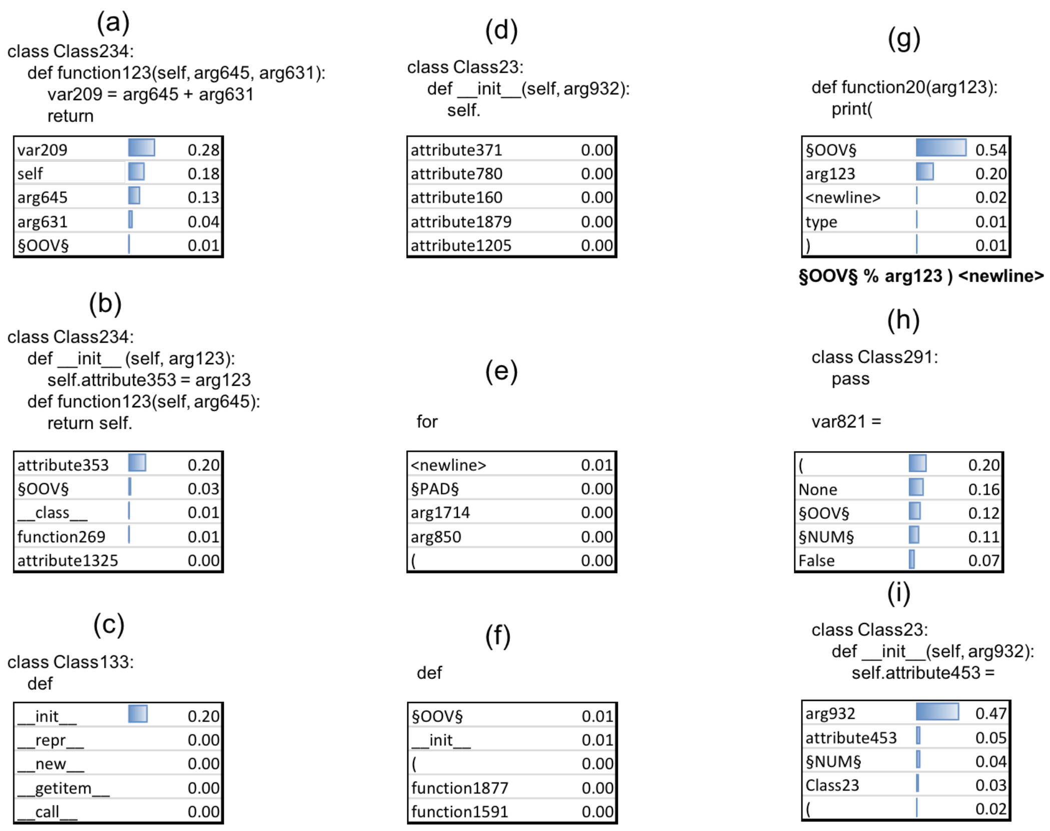
```
class Class23:
    def __init__(self, arg932):
        self.attribute453 =
```

arg932	0.80
§NUM§	0.03
[	0.02
np	0.01
var870	0.00

Figure 6.10: A sample of suggestions involving identifiers made by the LSTM+Att(50) model on the normalised dataset

only possible identifier to follow after `self.` was an attribute, as evidenced by almost all of the weight being placed on the `attribute172` token. While the result was very promising, it would also have been good to have the class's member functions in the top 5 suggestions. It is unclear why the model did not include these in the top 5, as they did appear in the attention memory. It may be the case that the training data contained insufficient instances of class member functions being referenced inside other class member functions.

In summary, all of the neural models evaluated in this work were capable of making good suggestions in cases involving Python syntax or structure. The addition of attention mechanisms improved suggestions slightly for common API test cases and significantly improved suggestions in cases involving identifier references. The LSTM+Att CS model was found to be able to deal with the long-term dependencies it was designed for, while being more interpretable and more efficient



(a)

```
class Class234:
    def function123(self, arg645, arg631):
        var209 = arg645 + arg631
        return
```

var209	0.28
self	0.18
arg645	0.13
arg631	0.04
§OOV§	0.01

(b)

```
class Class234:
    def __init__(self, arg123):
        self.attribute353 = arg123
    def function123(self, arg645):
        return self.
```

attribute353	0.20
§OOV§	0.03
__class__	0.01
function269	0.01
attribute1325	0.00

(c)

```
class Class133:
    def
```

__init__	0.20
__repr__	0.00
__new__	0.00
__getitem__	0.00
__call__	0.00

(d)

```
class Class23:
    def __init__(self, arg932):
        self.
```

attribute371	0.00
attribute780	0.00
attribute160	0.00
attribute1879	0.00
attribute1205	0.00

(e)

```
for
```

<newline>	0.01
§PAD§	0.00
arg1714	0.00
arg850	0.00
(	0.00

(f)

```
def
```

§OOV§	0.01
__init__	0.01
(	0.00
function1877	0.00
function1591	0.00

(g)

```
def function20(arg123):
    print(
```

§OOV§	0.54
arg123	0.20
<newline>	0.02
type	0.01
)	0.01

§OOV§ % arg123) <newline>

(h)

```
class Class291:
    pass

var821 =
```

(	0.20
None	0.16
§OOV§	0.12
§NUM§	0.11
False	0.07

(i)

```
class Class23:
    def __init__(self, arg932):
        self.attribute453 =
```

arg932	0.47
attribute453	0.05
§NUM§	0.04
Class23	0.03
(	0.02

Figure 6.11: A sample of suggestions involving identifiers made by the LSTM+Att CS model on the normalised dataset

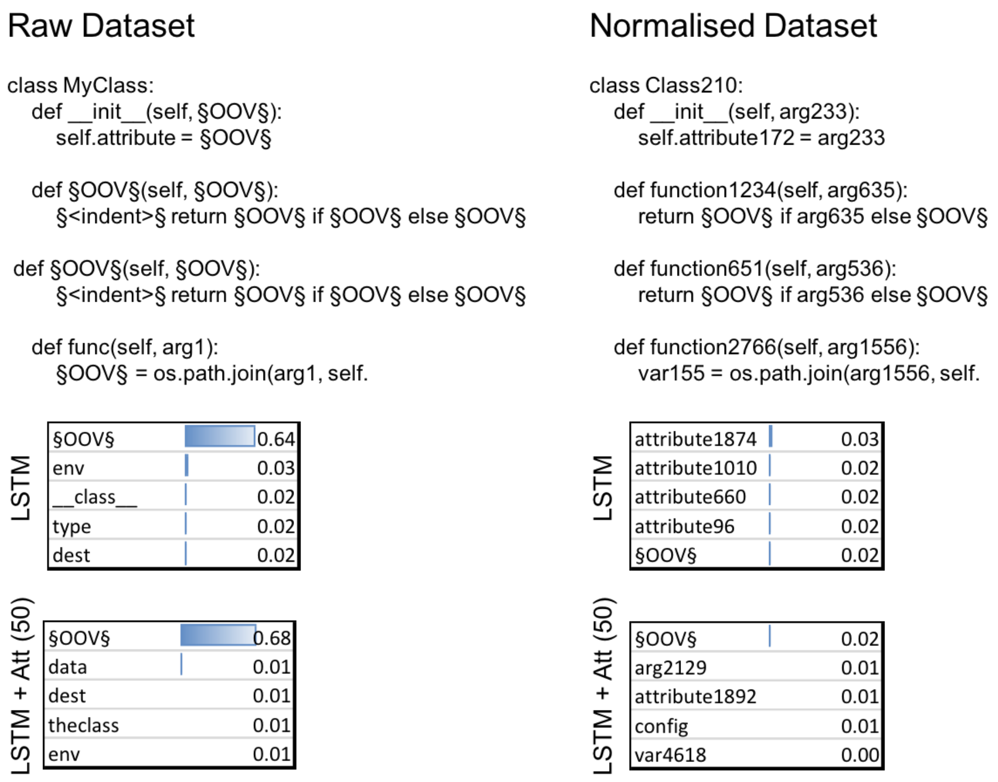
in its use of attention memory than the standard attention language models.

6.4 Results from Initial Attempts

This section provides a brief discussion of the results for the initial attempts at model building prior to the LSTM+Att CS model, as discussed in section 5.4.

Figure 6.15 shows an attention visualisation, similar to figure 6.3 that was typical of the majority variations of the initial model that were tried. The majority of the initial model variations very quickly learned to assign full weight to the language model task and none to the attention mechanism. The attention weights showed that the models tended to track one or two items in their memory, but this of course made no difference to the final output as a result of the zero weight assigned to the attention mechanism.

In some variations, the model did learn to sporadically assign weight to the



Raw Dataset

```
class MyClass:
    def __init__(self, §OOV§):
        self.attribute = §OOV§

    def §OOV§(self, §OOV§):
        §<indent>§ return §OOV§ if §OOV§ else §OOV§

  def §OOV§(self, §OOV§):
        §<indent>§ return §OOV§ if §OOV§ else §OOV§

    def func(self, arg1):
        §OOV§ = os.path.join(arg1, self.
```

LSTM

§OOV§	0.64
env	0.03
__class__	0.02
type	0.02
dest	0.02

LSTM + Att (50)

§OOV§	0.68
data	0.01
dest	0.01
theclass	0.01
env	0.01

Normalised Dataset

```
class Class210:
    def __init__(self, arg233):
        self.attribute172 = arg233

    def function1234(self, arg635):
        return §OOV§ if arg635 else §OOV§

    def function651(self, arg536):
        return §OOV§ if arg536 else §OOV§

    def function2766(self, arg1556):
        var155 = os.path.join(arg1556, self.
```

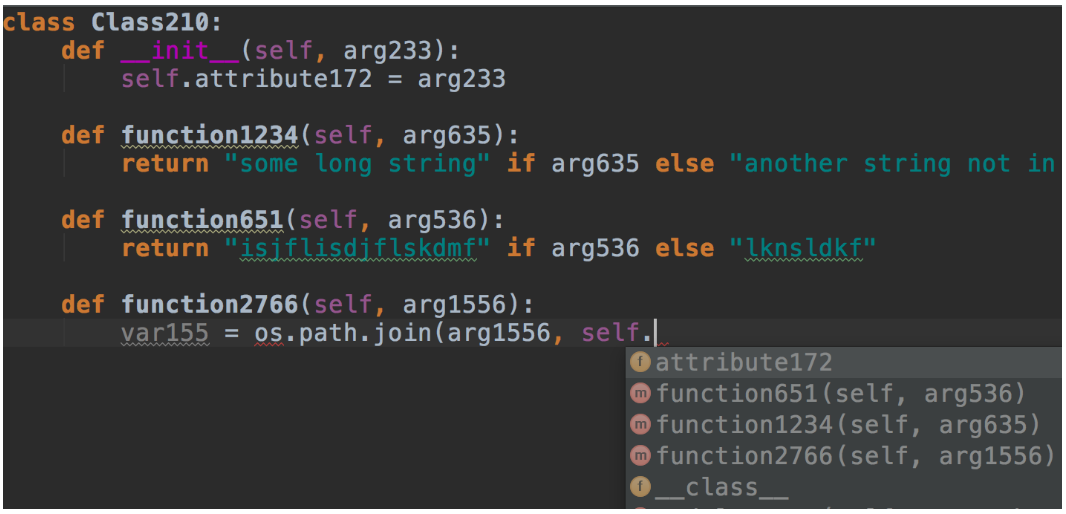
LSTM

attribute1874	0.03
attribute1010	0.02
attribute660	0.02
attribute96	0.02
§OOV§	0.02

LSTM + Att (50)

§OOV§	0.02
arg2129	0.01
attribute1892	0.01
config	0.01
var4618	0.00

Figure 6.12: Suggestions made by the LSTM and LSTM+Att models on a typical long-range dependency example



```
class Class210:
    def __init__(self, arg233):
        self.attribute172 = arg233

    def function1234(self, arg635):
        return "some long string" if arg635 else "another string not in

    def function651(self, arg536):
        return "isjflisdjflskdmf" if arg536 else "lknsldkf"

    def function2766(self, arg1556):
        var155 = os.path.join(arg1556, self.
```

```
f attribute172
m function651(self, arg536)
m function1234(self, arg635)
m function2766(self, arg1556)
f __class__
```

Figure 6.13: IntelliJ IDE correctly suggesting the class's members

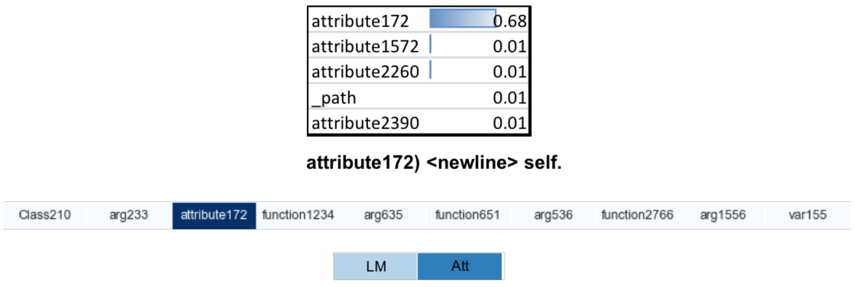


Figure 6.14: Correct suggestion made by the LSTM+CS Att model

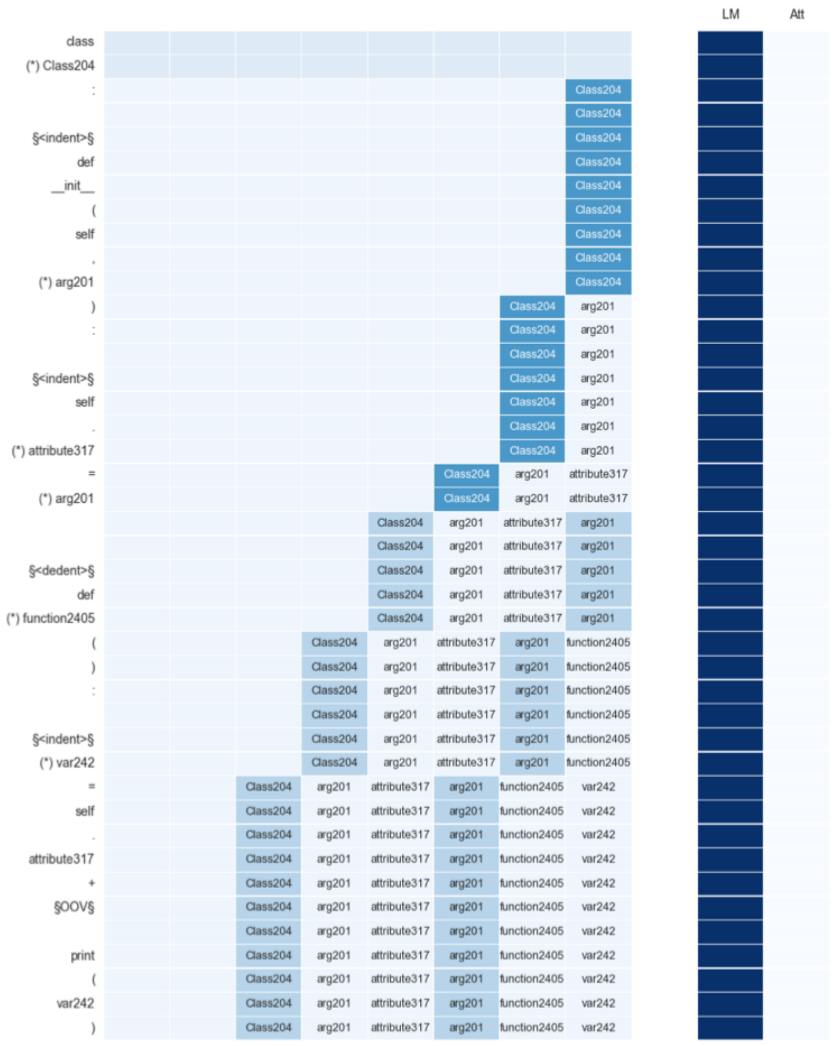


Figure 6.15: Attention weights of a variation of the initial attempt

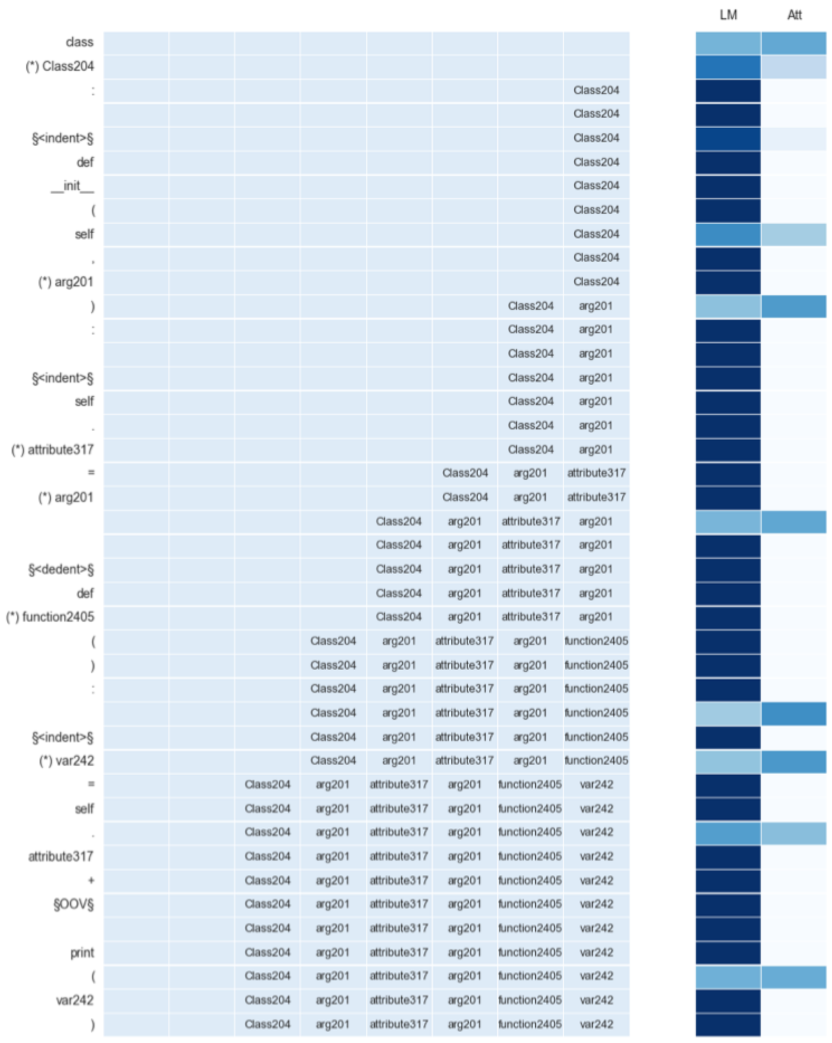


Figure 6.16: Attention weights of a different variation of the initial attempt showing sporadic weights applied to the attention mechanism

attention mechanism, but this was done at unintuitive times and was coupled with uniformly distributed weights over the attention memory and worse validation set perplexity than the LSTM model. An example of this is shown in Figure 6.16

One speculation as to why the initial models always strongly favoured the language model was that unlike in the standard attention for language models, the input to the attention mechanism was very sporadic. As noted by Allamanis and Sutton [46] on their corpus, 56 new identifiers were introduced per 1000 lines of code. Combining this with even a conservative average length of 10 tokens per line, would mean that only 0.5% of all tokens get flagged as being new identifiers that should enter the attention memory. It was thought that this sporadic nature meant that the parameters relating to the attention mechanism (such as those determining the attention component of a split output vector), remained random a lot longer than the parameters for the language model. This may have resulted in the model learning to ignore the attention mechanism as its contribution remained a lot noisier than the language model's which was able to learn much faster. The LSTM+Att CS model, on the other hand, would have been able to avoid this issue. Even early in training, the LSTM+Att CS could make a meaningful contribution to the final output probabilities. For example, consider the case where one identifier was in the attention memory. The model, through the use of softmax, would have assigned some not insignificant probability to this identifier's token type. Since there was only one identifier declared, it is certain that any reference to a pre-declared identifier would be to that identifier. The attention mechanism, through assigning this identifier a probability larger than say a uniform probability over the entire vocabulary of an untrained LSTM language model, would already be able to significantly improve the model's prediction.

The attention visualisation of the LSTM+Att(50) model of figure 6.2 also provided a hint as to why the initial model may not have worked. As noted in the discussion of that figure, the attention weights assigned by the standard language model attention mechanism were somewhat unintuitive. Even in cases where the model was able to successfully suggest an identifier reference that the LSTM model could not, the attention weights were focussed on points later in the input sequence than the step at which the identifier declaration occurred. By forcing the model to only look at identifier declaration points, it may have failed to capture the information necessary to make these identifier references.

Chapter 7

Conclusions and Future Work

This chapter concludes by summarising the key findings of this dissertation as they relate to the stated aims of section 1.3. A few avenues for possible future work are then briefly discussed.

7.1 Conclusion

This dissertation looked into the effectiveness of modern deep neural language models on the task of code suggestion and in particular whether neural attention models would improve their ability to deal with the long-range dependencies found in source code. The majority of past work involved small corpora and focussed on static programming languages like Java. Since dynamic languages arguably have the most to gain from an improved code suggestion system, this work used Python as its target language. A new, large, Python corpus was developed using code obtained from Github and will be made available to future researchers. An analysis of the corpus showed that there were clear long-range dependencies, particularly when it came to references to previously declared identifiers.

It was found that even what is now considered a fairly straightforward neural language model, the RNN with LSTM cells, was very effective at making useful suggestions, particularly when it came to Python syntax and structure as well as common API usage.

The addition of neural attention mechanisms to the language model significantly improved upon its ability to deal with identifier references, the source of long-range dependencies in source code. The LSTM+Att(50) model, a standard neural attention language model, achieved a perplexity score 4 points lower than the LSTM model. It also achieved close to 6 percentage points better top-1 and top-5 accuracies.

Finally, a novel model was developed that allowed for multiple filtered attention mechanisms, each focussing on a different task. The idea being that by providing control over which embeddings entered the attention memory, rather than using a fixed lagging window, the unique long-range dependencies of source code could be more efficiently modelled. The model could also learn when to use each attention mechanism based on the state representation of the input sequence it had read so far. To test the concept, a single task was used which involved tagging all identifier declarations to go into the attention memory. The model was able to identify appropriate points at which to invoke the attention mechanism and was also able to differentiate between the different groups of identifiers. For example, it learned to use the attention mechanism following the tokens `self.` and assigned weight only to the attribute identifiers in that case. The novel model significantly outperformed a standard attention language model with the same size of attention memory across all metrics. Not only that, it was also able to outperform a standard attention model with more than double the attention memory while also being much more interpretable.

7.2 Future Work

The promising results of the neural language models, particularly the accuracies of the various attention models, presents an opportunity to develop this work into an actual code suggestion system integrated into an IDE. This was done, for example, by Hindle et al. [1] who found their n-gram based code suggestion engine to be superior in many respects to the heuristic-based one in the Eclipse IDE. Since Python code suggestion systems are not necessarily as advanced as their static language counterparts, this suggests that a code suggestion system for Python built using more advanced language models should be very effective. In addition to this, language models are capable of providing more than a single token as a suggestion, and are able to capture common idiomatic patterns as observed in the qualitative analysis of chapter 6.

It should be noted, however, that the focus of this work was on the code suggestion task, which is subtly, but importantly different from the code completion task. Code completion involves the recommendation of a completion and extension to the sequence, given a sequence of complete tokens and a partial token. As it stands, the models presented in this work are incapable of modelling partial tokens. The principled way to achieve this would be to use a character-level model that is capable of modelling these partial tokens [65]. However, it may also be possible to model the code completion task a special case of the code suggestion task and use some variation of a hierarchical softmax [29] to restrict the suggestions of the model to those containing a particular prefix.

Extending these models into a practical code suggestion system would also require some further development, particularly around the unit of operation, which is currently at the file level. Existing IDE code suggestion systems generally also provide suggestions involving other files in a project, so the models would need to incorporate project-level information to compete.

The novel attention model of chapter 5 also presents a number of unexplored avenues for future work. In particular, it would be good to test other tagging schemes and attention tasks. For example, one task could specialise in global identifiers, while another specialises in local identifiers. Tasks relating to different identifier groups could also be separated. For example, attribute references generally only follow a reference to self or another object. While the attention model investigated did learn to differentiate attributes from other identifiers, it could potentially further benefit from being treated separately. It would also be interesting to evaluate the effect of using different embeddings for the filtered attention memories. One of the approaches tried in the initial model building attempts was to split the LSTM output vector into two parts, one used for the language model and the other used to drive the attention mechanism. Intuitively, this should work well as it allows for further specialisation of components, but this remains to be tested.

It would also be worthwhile investigating to what extent the results found in this work also apply to other dynamic languages such as Javascript, which has a syntax more similar to Java which was primarily used in previous work.

Finally, the works of Maddison and Tarlow [42] and Allamanis and Sutton [2] provide another interesting avenue for further exploration. They modelled source code using PCFGs, which makes sense given that programming languages have formally defined grammars and unambiguous parsers. Recent neural architectures such as the TreeLSTM of Tai et al [44] and the Recurrent Neural Network Grammar model of Dyer et al. [45] provide some of the components that would be required for a similar neural implementation.

Bibliography

[1] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[2] Miltiadis Allamanis and Charles A. Sutton. Mining idioms from source code. *CoRR*, abs/1404.0417, 2014.

[3] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

[4] Ilya Sutskever, James Martens, and Geoffrey E. Hinton. Generating text with recurrent neural networks. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024, New York, NY, USA, 2011. ACM.

[5] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[7] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In Tony Jebara and Eric P. Xing, editors, *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1764–1772. JMLR Workshop and Conference Proceedings, 2014.

[8] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[9] H. Khanh Dam, T. Tran, and T. Pham. A deep language model for software code. *ArXiv e-prints*, August 2016.

[10] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[11] Xiaojin Zhu. Cs838-1 advanced nlp: Language modeling. `http://pages.cs.wisc.edu/~jerryzhu/cs838/LM.pdf`. [Online; accessed 8-August-2016].

[12] G. Lidstone. Note on the general case of the Bayes–Laplace formula for inductive or a posteriori probabilities. *Transactions of the Faculty of Actuaries*, 8:182–192, 1920.

[13] R. Kneser and H. Ney. Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181–184 vol.1, May 1995.

[14] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ACL '96, pages 310–318, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics.

[15] 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.

[16] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon and David B. Dunson, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, volume 15, pages 315–323. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

[17] Raúl Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[18] Ya-xiang Yuan. Step-sizes for the gradient method. *AMS IP Studies in Advanced Mathematics*, 42(2):785, 2008.

[19] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[20] Michiel Hermans and Benjamin Schrauwen. Training and analysing deep recurrent neural networks. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 190–198. Curran Associates, Inc., 2013.

[21] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166, March 1994.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[23] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[24] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[25] Pieter-Tjerk de Boer, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 134(1):19–67, 2005.

[26] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[27] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012.

[28] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[29] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *AISTATS*, 2005.

[30] Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On using very large target vocabulary for neural machine translation. *CoRR*, abs/1412.2007, 2014.

[31] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[32] Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. *CoRR*, abs/1506.03340, 2015.

[33] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey E. Hinton. Grammar as a foreign language. *CoRR*, abs/1412.7449, 2014.

[34] Vladyslav Kolesnyk, Tim Rocktäschel, and Sebastian Riedel. Generating natural language inference chains. *CoRR*, abs/1606.01404, 2016.

[35] Tim Rocktäschel, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, and Phil Blunsom. Reasoning about entailment with neural attention. *CoRR*, abs/1509.06664, 2015.

[36] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *CoRR*, abs/1601.06733, 2016.

[37] Ke M. Tran, Arianna Bisazza, and Christof Monz. Recurrent memory network for language modeling. *CoRR*, abs/1601.01272, 2016.

[38] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. ACM.

[39] Miltiadis Allamanis, Earl T. Barr, and Charles A. Sutton. Learning natural coding conventions. *CoRR*, abs/1402.4182, 2014.

[40] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 252–261, New York, NY, USA, 2014. ACM.

[41] 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*, FSE 2014, pages 269–280, New York, NY, USA, 2014. ACM.

[42] Chris J. Maddison and Daniel Tarlow. Structured generative models of natural source code. *CoRR*, abs/1401.0514, 2014.

[43] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. Phog: Probabilistic model for code. In *ICML*, 2016.

[44] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015.

[45] Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. Recurrent neural network grammars. *CoRR*, abs/1602.07776, 2016.

[46] Miltiadis Allamanis and Charles A. Sutton. Mining source code repositories at massive scale using language modeling. In Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim, editors, *MSR*, pages 207–216. IEEE Computer Society, 2013.

[47] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[48] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA, 2015. IEEE Press.

[49] Subhasis Das and Chinmayee Shah. Contextual code completion using machine learning.

[50] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. *CoRR*, abs/1602.03001, 2016.

[51] Lili Mou, Ge Li, Yuxuan Liu, Hao Peng, Zhi Jin, Yan Xu, and Lu Zhang. Building program vector representations for deep learning. *CoRR*, abs/1409.3358, 2014.

[52] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *CoRR*, abs/1603.06744, 2016.

[53] Pierre Carbonnelle. Pypl popularity of programming language. `http://pypl.github.io/PYPL.html`. [Online; accessed 30-August-2016].

[54] Carlo Zapponi. Githut - programming languages and github. `http://githut.info/`. [Online; accessed 19-August-2016].

[55] Github Inc. Github press info. `https://github.com/about/press`. [Online; accessed 30-August-2016].

[56] Github Inc. Fork a repo. `https://help.github.com/articles/fork-a-repo/`. [Online; accessed 23-July-2016].

[57] G. K. Zipf. *Selective Studies and the Principle of Relative Frequency in Language*. Harvard University Press, 1932.

[58] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: Mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2):300–336, April 2009.

[59] Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. *CoRR*, abs/1506.03340, 2015.

[60] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O. K. Li. Incorporating copying mechanism in sequence-to-sequence learning. *CoRR*, abs/1603.06393, 2016.

[61] Graham Neubig. Kylm - the kyoto language modeling toolkit. `http://www.phontron.com/kylm/`. [Online; accessed 23-July-2016].

[62] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[63] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In David Blei and Francis Bach, editors, *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2342–2350. JMLR Workshop and Conference Proceedings, 2015.

[64] Mark F. Medress, Franklin S Cooper, Jim W. Forgie, CC Green, Dennis H. Klatt, Michael H. O'Malley, Edward P Neuburg, Allen Newell, DR Reddy,

B Ritea, et al. Speech understanding systems: Report of a steering committee. *Artificial Intelligence*, 9(3):307–316, 1977.

[65] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014.