Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Improving Code Completion with Machine Learning

Author:
Boris Barath

Supervisor:
Prof. William Knottenbelt
Second marker:
Dr. Thomas Heinis

June 15, 2020

Abstract

All modern IDEs feature intelligent code completion and it is used by both new and experienced software developers alike. However, there is still room for improvement in these code completion engines in terms of sorting the given completions according to the context of previously written code. We define a metric for comparing the performance of code completion systems and propose a tool to improve the performance of Jedi, the most widely used code completion tool using Machine Learning. Our proposed approach saves over 3% more keystrokes than using Jedi alone. This approach also significantly increases the relevance of completions as we achieved a 23% higher likelihood of the desired code completion appearing first in the list of completions. We also introduce tooling which enables people interested in working on this topic to better evaluate their code completion systems and provide researchers with a way to directly compare their models against other work in the field.

Acknowledgements

First of all, I would like to thank Professor William Knottenbelt for his supervision on my industry proposed project.

I would also like to thank Jan Matas and the engineers at Deepnote for their help and invaluable feedback and Martin Zlocha for his insights and proofreading of this report.

Finally, I would like to thank family and friends for their advice and for always supporting me.

Contents

List of Figures

Chapter 1

Introduction

All modern IDEs feature intelligent code completion and it is used by both new and experienced software developers. Intelligent code completion greatly increases developers' speed by providing them with suggested completions of the expression or line they are currently writing. The aim is to reduce the number of keystrokes and the number of common mistakes, like typos. Code completion can also decrease the need for developers to refer to documentation by suggesting method parameters and skeletons for looping and branching constructs. Most code completion engines perform static code analysis and keep track of existing classes, fields, methods and keywords in a project. This allows for valid suggestions of the next token according to what the developer has already written. However, these engines often do not perform well at sorting the suggested code completions based on the intention / structure of the code written previously, as completions are often sorted alphabetically. There is room for improvement in such engines with support for features like completion of loops, branches and filling in method arguments with valid, previously defined variables.

1.1 Objectives

Even though there are many commercial code completion engines (as outlined in the next chapter), there is currently only a limited set of open source alternatives. The aim of this project is to **develop an open source completion engine that outperforms other solutions available today by leveraging modern machine learning methods**. The resulting engine should be fast, reliable, easy to use and easy to integrate with existing systems. Ultimately, the result of this project will be deployed as autocompletion engine for Deepnote [4], a collaborative data science platform.
This high level objective needs to be further divided to a couple of smaller steps:

- Find a suitable dataset for training the model that will sit at the core of the completion engine.
- Develop a metric that evaluates different model in terms of time saved for the end user.
- Create a pipeline for quick training of models that automates all necessary steps to preprocess data, feed them to the model and output a trained predictor.
- Impelement an evaluation pipeline that would allow us to quickly evaluate different models and compare their results on an unseen test set.

- Experiment with many architectures, including but not limited to transformer models (GPT-2 [51]), LSTMs, CNNs, (with attention) or simple feed forward networks and identify the best performing model.
- ”Productionalize” the final model and integrate it to Deepnote systems so it can be tested with real users.

1.2 Challenges

The project proved to be significantly more challenging than originally anticipated. Main issues are summarized in the following points:

- **Finding a suitable dataset** When we started working on the project, we had a clear plan in mind: use CodeSearchNet [6] released by GitHub. It contains around 400,000 Python functions, it is freely available and already pre-processed for research purposes. However, we later realized that this was not an optimal choice. The dataset contained only the methods, not the imports and hence the static analysis tools we wanted to extend did not have enough to context to work effectively. In later chapters we will explain this issue in more detail and also propose a different dataset more suitable for our needs.
- **Syntactically correct suggestions** - We spent a lot of time working with end-to end models that would take in the code and directly produce a completion suggestion - without performing prior static analysis. This was supposed to save time, produce larger snippets and reduce the amount of dependencies of the final engines. While we had some success, the resulting code was not always syntactically correct which was very annoying for the users who had to go back and correct the accepted completions. We have instead pivoted to improve on existing static analysis based completion library.
- **Pre-processing, training and evaluation time** - GPT-2 has multiple available models, the smallest of which has 124 million parameters. For our later models, data pre-processing time is measured in an order of hours. Our evaluation method, which is essentially a simulation of code being written requires iterating over every character in every source code file also took up to an hour per model to complete.
- **Correct evaluation** - we encountered some issues with the correctness of our evaluation pipeline that invalidated some initial experiments. This was both due to code errors and having too many error-prone manual steps. We have decided to rewrite the pipeline from scratch to make it fully automatic with an extensive test suite to ensure correctness.

1.3 Contributions

Overall, we consider the project a success, since we completed each objective and got a model outperforming open source alternatives, albeit not in the way we originally anticipated. The many dead ends we took during the work on this project were often exhausting and frustrating, but it revealed a true nature of engineering in the wild: that the plans and requirements are subject to constant change and even greatly thought through plans might hit road-blocks during the implementation phase.

Of course, our main contribution is the extension to autocompletion engine Jedi that improves its efficiency and allows users to save significantly more keystrokes as compared to using it without our model. However, the end result, in terms of number of keystrokes saved was smaller than we hoped for, which brings us to our second contribution. We open sourced [2] our pre-processing pipeline and also our evaluation pipeline in a from of **code-completion-gym**. Taking inspiration from other research environments (gyms) [22], our repository will allow future research to skip the arduous steps of building correct evaluation pipeline and instead focus on writing models that will hopefully outperform ours and make writing code faster and more pleasurable than ever before.
For completeness, this is the list of contributions of this thesis:
This is an incomplete list of the main contributions of this individual project, sorted by their importance

1. Open source autocomplete ”sorter” extension to Jedi, that significantly reduces the number of keystrokes required
2. Open source **code-completion-gym** that allows researchers to quickly evaluate code completion models and compare their results in a reproducible fashion
3. Open source pre-processing pipeline that converts code to token streams suitable for training neural networks
4. Two metrics suitable for comparing code completion systems.
5. Production-ready implementation as a part of Palantir’s python-language-server that works with Deepnote backends
6. Evaluation of datasets suitable for training and testing code completion engines

Chapter 2

Background

A multitude of data science and machine learning approaches were applied to language modeling, text prediction and code prediction. Below is a summary of state-of-art approaches for both text prediction and code prediction, and how these approaches evolved over time.

2.1 Statistical Language Models

Statistical language models (LMs) represent a probability distribution over a sequence of words. LMs learn the probability of word occurrences from a provided training dataset. Given a sequence of words, LMs can predict the next word according to the probability distribution of the training data the model approximates. In modern approaches, LMs can be used to support more sophisticated models. Statistical language modeling is used in many Natural Language Processing applications including Handwriting Recognition, Image Captioning, Machine Translation, Text Completion in smartphone keyboards and others. In this section, we will examine some of the common statistical models and methods, starting with the simplest one, N-grams.

2.1.1 N-grams

N-grams [33] are probabilistic language models represented by an (n-1)-order Markov source (a sequence of n-1 of random variables defined by a Markov chain). A Markov chain exploits the assumption that the probability of the next word only depends on its n-1 predecessors. Since this property mostly holds for natural language, Markov source is a good approach for modeling natural language. To be more precise, they estimate words from a fixed window of previous words. For a sequence of words $(w_0, w_i, ..., w_{j-1})$ and a history (sentence) $H_j = (w_{j-n-1}, ..., w_{j-1})$ an N-gram model predicts w_j using sentence S which maximises the probability $P(w_j|S)$ under the assumption that each word only depends on its n-1 predecessors (known as Markov property). The formula for N-gram probability is: [53]

$$P(w_i|H_i) \approx P(w_i|w_{i-n-1}, ..., w_{i-1})$$

In modern natural language processing applications, N-grams are not used as a stand alone solution, but usually supporting a more advanced model, which leads us to common problems with N-gram language models that make them hardly applicable to our domain.

Problems with N-grams

- **Short distance dependencies** - N-grams can only represent dependencies between a word and its n-1 predecessors, which implies they lack long distance dependencies. This limits the possible usage of N-grams for modeling computer programs, since it would have hard time predicting a usage of a library function imported on the top of the file.
- **Unknown words** - In our specific application of language modeling, it is possible that a developer uses a variable, or library which is not present in the training set. This way, we could never use that word as a prediction, as N-gram models would not predict it. This problem is addressed in the $-grams section below.

Smoothing [33]

It may occur that some words are present in our vocabulary, but appear in a different context in the test set. Under the classic N-gram formulation, they would get assigned a zero probability. One method to avoid this issue is to introuduce **Smoothing**. Smoothing means removing a small bit of probability from frequently occurring events (discounting) and assigning it to unseen events. Therefore even events (words) with 0 probability can be output by the smoothed model. These are common N-gram smoothing algorithms: [33]

- **Discounting** - discounting means subtracting a small count from frequently occurring n-grams which will be later added to n-grams occurring 0 times, so they will have a non-zero probability. Discounting is used in both back-off and interpolation smoothing algorithms below.
- **Absolute Discounting** - a modified version of discounting where a constant, fixed discount d is subtracted from each count.
- **Add-k smoothing** - this algorithm acts on the counts of occurrences of different N-grams or events in the corpus. For events that occur 0 times, it adds a fractional count to increase the probability of such an event from 0.
- **Back-off** - if we are computing a trigram $P(w_i|w_{i-1}, w_{i-2})$ which has zero occurrences, we can fall back and use the bigram $P(w_i|w_{i-1})$ instead. Similarly, we can back off from a bigram to a unigram, simply $P(w_i)$.
- **Interpolation** - instead of only using the main N-gram, take a weighted average of the N-gram, the (N-1)-gram, (N-2)-gram and so on. For N=3, we would take a weighted average of a trigram, a bigram and a unigram.
- **Kneser-Ney smoothing [37]** - Kneser-Ney smoothing is commonly referred to as the state-of-art smoothing algorithm and it is built on absolute discounting.

N-gram based algorithms

- **SLANG**[52] - SLANG is a tool for filling in holes (missing API calls) in a sequence of API calls. It uses a combination of N-grams (n=3, trigrams) and RNNME [45]. The evaluation of the paper concludes that using the average of the probabilities output by the two models for filling in holes in sequences of API calls outperforms using either of the models alone which highlights the fact that N-gram LMs are used supporting more advanced models (RNNME in this case).

- **CACHECA**[29] - CACHECA is an Eclipse IDE [27] plugin which aims to improve the suggestions given by Eclipse's default code completion tool. It is built using a Cache language model [62], which gives a higher probability of occurrence to words that previously occurred in the text. This is also highlighted by locality of code, which says that code has local regularities (a variable will generally occur multiple times in a block of code).[62] CACHECA combines Eclipse's default suggestions with their implementation of a Cache language model ($-grams)

SLANG [52]

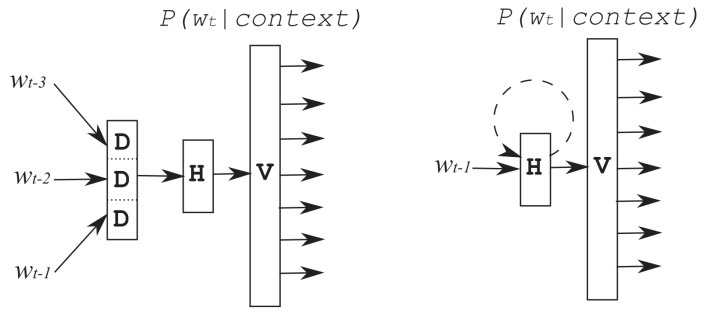


Figure 2.1: 4-gram Feedforward Neural Network LM and Recurrent Neural Network LM [45]

$-grams

$-grams are a combination of N-grams and a cache component, which stores all N-grams seen in local code. $-grams then store 2 probabilities, one from the corpus and another from the cache component. These probabilities are linearly interpolated. [62] The N-gram represents probabilities in the training text (programming languages) while the cache component models the provided text (code written so far).

2.2 Distributed Word Representations

In a distributed word representation, each word in the vocabulary is modeled as a real-valued feature vector. Semantically similar words in such a representations are similar. Figure 2.2 illustrates this similarity. [36] Distributed word representations play an important role in Neural Language Models.

2.3 Curse of dimensionality

The term curse of dimensionality was coined by Bellman in his book about dynamic programming. [20] It is a problem on high-dimensional data - as the dimensionality of data increases, even a large amount of data is sparse. Simply put, if we have a dataset and add another dimension to it, we would need a lot more data to retain the same density of data as we had before we added the new dimension. For example, in N-grams, a word sequence in the test dataset might have never been encountered in the training corpus. Neural language models tackle the curse by using distributed word representations.

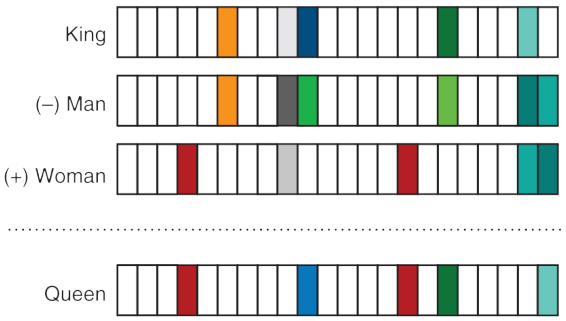


Figure 2.2: An illustration of the feature vector word representation and the similarity between semantically close words. [65]

Summarising N-grams

N-grams represent a simple statistical LM, which is easy and quick to understand and to compute. However, It has no way of modeling long distance context and it cannot generalize to unseen data. Because of this, we need to look at more recent advances in language modeling.

2.4 Neural Language Models

2.4.1 Feedforward Neural Networks

The idea behind NN LMs is projecting the words from a corpus onto a continuous space and to use a probability estimator operating on this space. The first neural networks used in language modeling used a feedforward architecture. [21]

FF Neural Network Language Models

A characteristic feature of FFNNs is the fact that connections between their nodes do not form a cycle. Feedforward neural networks are used to learn a statistical model of the distribution of word sequences. [21] A disadvantage of Feedforward NN LMs is that they lack longer distance dependencies as they essentially represent N-gram language models. This is highlighted by Figures 2.1 and 2.3 by the fact that they take as input a fixed number of previous words, much like N-grams. However, in contrast to N-grams, FFNN LMs do not need to back off to a lower order ((N-1)-gram) as the probabilities are interpolated for any possible sequence of length N-1. [55]

An example FFNN LM

Figure 2.3 shows an architecture of a FFNN LM. The input of the FFNN LM is a 1-of-N encoding where the i-th word is represented by the i-th element of the vector is set to one and the rest to zero. It has a hidden projection layer with a shared mapping P which maps words from the corpus into feature vectors.

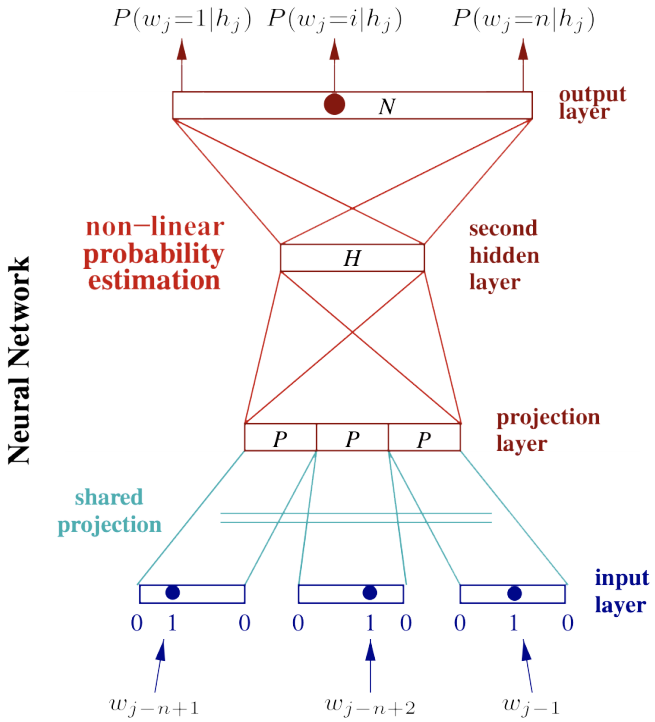


Figure 2.3: A Feed forward NNLM with a projection layer which maps words to feature vectors. [54]

2.4.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are commonly used in computer vision and image processing. [47] They use a type of layer called a convolutional layer, which can extract features by convolving a learnable filter (also called kernel). [38] CNNs also saw some use for different Natural Language Processing tasks.

CNN Sentence Models [65]

CNNs use hundreds of kernels (convolutional filters), each of which extracts a specific pattern of n-gram from the language. These convolutional operations are applied over a window of words to create a new feature. They are followed by max-over-time-pooling, which only chooses the most important feature (the feature with the highest value) for each feature map. [35] [34]
Figure 2.4 represents such a CNN where each word is represented by a k-dimensional vector. The network proposed in [35] once again lacked long distance dependencies, but was addressed by Dynamic CNNs (DCNN) in [34], which proposed a k-max pooling strategy, which selects k most important features.

CNN Language Models [50]

CNNs did not see much use in more advanced NLP applications such as prediction (Language modeling). There were experiments with extending a FFNN Language Model with a convolutional layer. The network in this experiment consists of:

1. Mapping input N-grams to low dimensional vectors
2. A convolutional layer followed by an ReLU activation
3. A mapping layer which maps convolutional output to a lower dimensional vector
4. A highway layer [57]
5. A softmax layer which computes the prediction of the next word

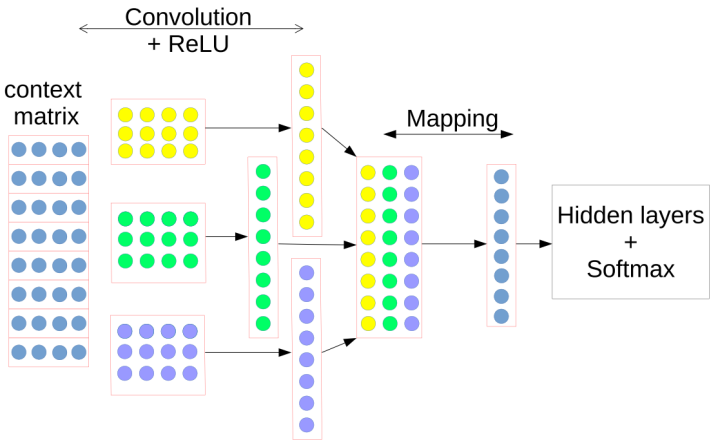


Figure 2.4: Architecture of a CNNLM. [50]

Figure 2.4 shows the CNN described above This CNNLM managed to outperform FF LMs and RNN LMs, however it lacked behind LSTM LMs which currently represent a state of the art and will be described below.

2.4.3 Recurrent Neural Networks

RNNs work on the idea of processing sequences. They perform the same computation over each token in a sequence and each step depends on the previous token and result. [56] They feature a hidden state (also called a memory) which captures information about a sequence of previous inputs. The memory is updated using previous memory and current input token. The inherent sequential nature of language implies that RNNs are well suited for language modeling and many other NLP tasks. RNNs create the possibility to capture unbounded context (long distance dependencies) [65] An advantage of RNNs over FFNNs for Language modeling is that RNNs can process contexts of arbitrary length.

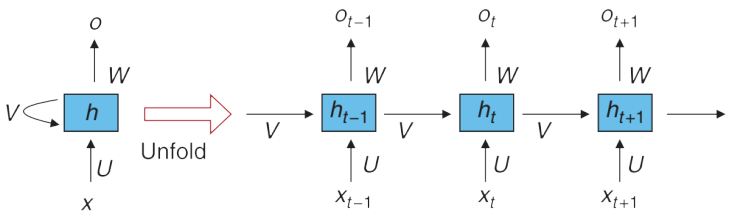


Figure 2.5: Architecture of a RNN. [65]

Figure 2.5 describes a simple RNN cell, and its unfolding. In the figure, x_t and h_t represent input to the network and the hidden state at time t respectively. U, V and W represent the weights which are shared across time. New hidden state is computed with: [50]

$$h_t = f(Ux_t + Wh_{t-1})$$

where f is a non-linear function like $tanh$ or ReLU.

RNN Language Models

First attempts at RNN language models (RNN LMs) were based on a simple recurrent neural network. They function just like normal RNNs, where the input consists of the next word in a sequence, and the hidden layer's state from the previous time step. During training, the data from the entire corpus is sequentially fed to the RNN. The output then represents a probability distribution of next words given the context (hidden state) and previous word. [46] A disadvantage of this architecture was slow training times, but this was addressed in [43] by using truncated back-propagation through time (truncated BPTT), which is an extension of back-propagation for RNNs, wherein at each word, an approximate gradient is computed by using a fixed number of predecessor words. Hence, the context length used in training is bounded. Figure 2.6 illustrates that using BPTT helps reduce perplexity in neural LMs in comparison to back-propagating loss through all time steps.

	Penn Corpus		Switchboard	
Model	NN	NN+KN	NN	NN+KN
KN5 (baseline)	-	141	-	92.9
feedforward NN	141	118	85.1	77.5
RNN trained by BP	137	113	81.3	75.4
RNN trained by BPTT	123	106	77.5	72.5

Figure 2.6: Comparison of perplexity (lower is better) on two corpora of various language models. [43]

Context Dependent RNNLM [44]

A further extension to RNNLM is the addition of a vector which adds contextual information about the sentence being modeled. [44] The model is extended by adding a feature layer using a context vector computed on sentence history using Latent Dirichlet Allocation (LDA) which enables the RNNLM to remember long term information. This feature layer represents topic information and is connected to both the hidden layer and the output layer. [44]

Comparison with previously seen LMs

Figure 2.8 already shows that a simple RNN architecture outperformed KN5 (interpolated 5-gram model) with modified Kneser-Ney smoothing and FFNNLM. Furthermore, it is apparent that the addition of a feature layer further decreased perplexity on the test set. Perplexity is a measurement of how well a probability distribution predicts a sample. In language modeling, lower perplexity means that a LM can predict text from an unseen test set better. It also supports the fact that combining a Neural Language Model with an N-gram model further improves the model's predictions.

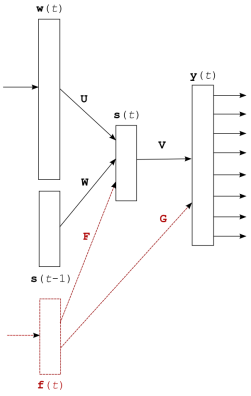


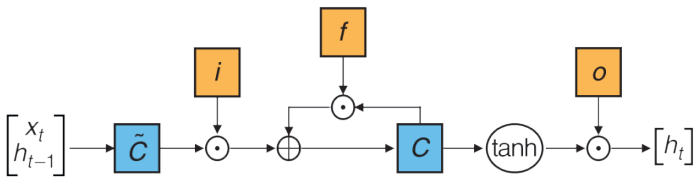
Figure 2.7: RNNLM extended with a feature layer f(t). [44]

Model	Individual	+KN5	+KN5+cache
KN5	141.2	-	-
KN5 + cache	125.7	-	-
Feedforward NNLM	140.2	116.7	106.6
Log-bilinear NNLM	144.5	115.2	105.8
Syntactical NNLM	131.3	110.0	101.5
Recurrent NNLM	124.7	105.7	97.5
RNN-LDA LM	113.7	98.3	92.0

Figure 2.8: Comparison of perplexity (lower is better) on the Penn Treebank Corpus of various language models. [44]

2.4.4 Long Short-Term Memory

LSTM [31] networks are a modified version of RNNs. An LSTM cell has three gates: an input gate, an output gate and a forget gate (labelled *i*, *o* and *f* in Figure 2.9) These gates decide whether to pass data forward or to erase the memory. The forget gate allows for the error to back propagate an unlimited number of time steps. The hidden state is computed as a combination of these three gates.



(1) Long Short-Term Memory

Figure 2.9: Structure of an LSTM cell. [50]

LSTM Language Models

LSTM models avoid the Exploding / vanishing gradient problem and allow LMs to model longer-distance dependencies. [31] [58] An example of a back propagation algorithm for LSTM training is epochwise BPTT, which performs two passes, computing the gradient and updating the weights once for the entire sequence. Epochwise BPTT is faster that truncated BPTT from RNN LM training. During epochwise BPTT, the length of the training sequence is truncated in that it cannot exceed the start of the sequence. [59] [41] [30]

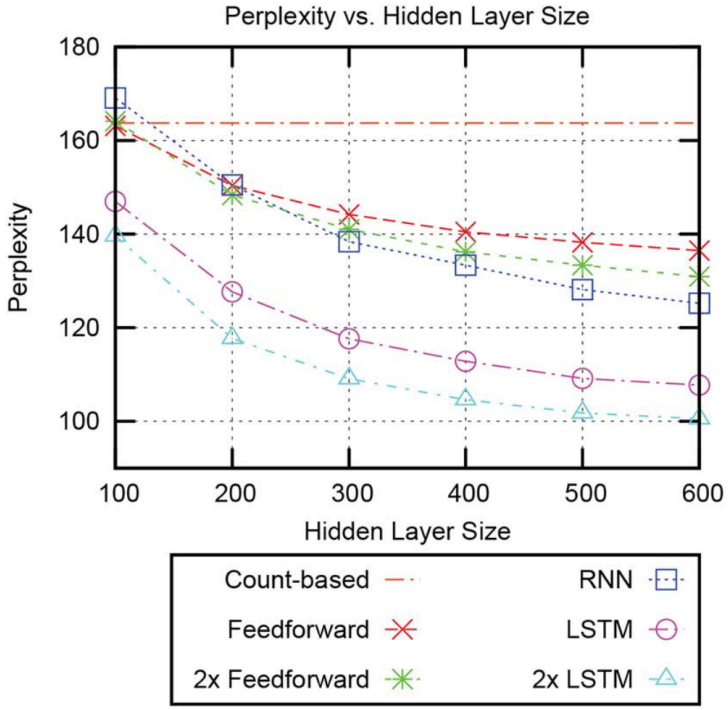


Figure 2.10: Relation of hidden layer size and perplexity for different LMs. [58]

Comparison to RNNLMs

Figure 2.10 shows that irrsepective of hidden layer size, LSTM LMs outperform RNNLMs, Feedforward NNLM (10-gram) and count-based (4-gram) language models.

2.4.5 Encoder-decoder RNN Architecture

The Encoder-Decoder RNN is designed for sequence to sequence prediction, which means taking a real-valued sequence and outputting a prediction of another real-valued sequence. An example application of sequence to sequence prediction is machine translation. Sequence to sequence models (encoder-decoder LSTMs) were first introduced by Google in [60]. One interesting observation in Google's paper is that reversing the input sequence improved the predictions, in particular on long sentences.
The encoder-decoder architecture consists of two RNNs, one (the encoder) is for reading the (variable length) input and translating it into a fixed length vector. An example problem is translating a phrase from one language to another, where the phrase can have a different

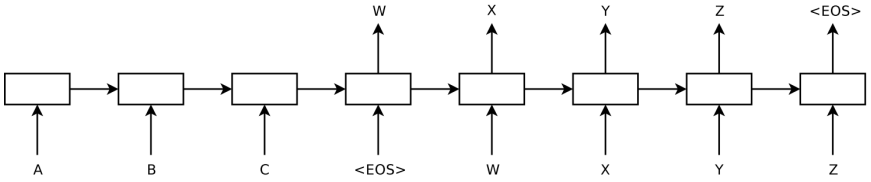


Figure 2.11: An example encoder-decoder sequence to sequence model, which reads the sequence ABC and outputs the sequence WXYZ. It stops generating output after the end-of-sentence (EOS) token. [60]

number of words in the two languages. The decoder maps such a fixed length vector representation into a variable length sequence output (the predicted sequence). [23] Figure 2.11 shows how the first LSTM reads input one timestep at the time and generates an internal representation. Then, the other LSTM outputs the predicted sequence.

2.4.6 Attention Mechanism

The attention mechanism is aimed at improving performance of encoder-decoder models as it was shown that the fixed length intermediate representation is a bottleneck on performance of such models. This can happen if particularly long sentences need to be mapped into a fixed-length representation, and [23] showed that performance deteriorates as sentence length increases

Instead of mapping the whole input sequence into a fixed-length vector, it maps the words in the input sequence into words, which are passed into the decoder. This avoids having to compress arbitrarily long sentences into a fixed-length vector and chooses a subset of (the most suitable) vectors. Intuitively, the decoder pays attention to certain parts of the input sequence (hence the name Attention Mechanism). It was shown that this model handles long sentences better than the Encoder-decoder architecture above. [19]

Self-Attention

Self attention is an extension of attention in which different positions of a single sequence are related in order to compute a representation of the sequence. [63]

2.4.7 Transformer Models

The transformer architecture is built using only stacked self-attention mechanisms in an aim to avoid recurrence, as the training of RNNs can not be parallelized. [63] Both the encoder and the decoder have six layers. Each encoder layer consists of a self-attention mechanism (looking both at previous and future tokens) and a FFNN. In contrast to the encoder layer, the decoder has a masked self-attention layer (which only looks at previous tokens) and an additional attention mechanism which allows it to pay attention to specific segments from the encoder.

The transformer architecture was shown to outperform various RNN and CNN based models in a machine translation task. [63]

2.12

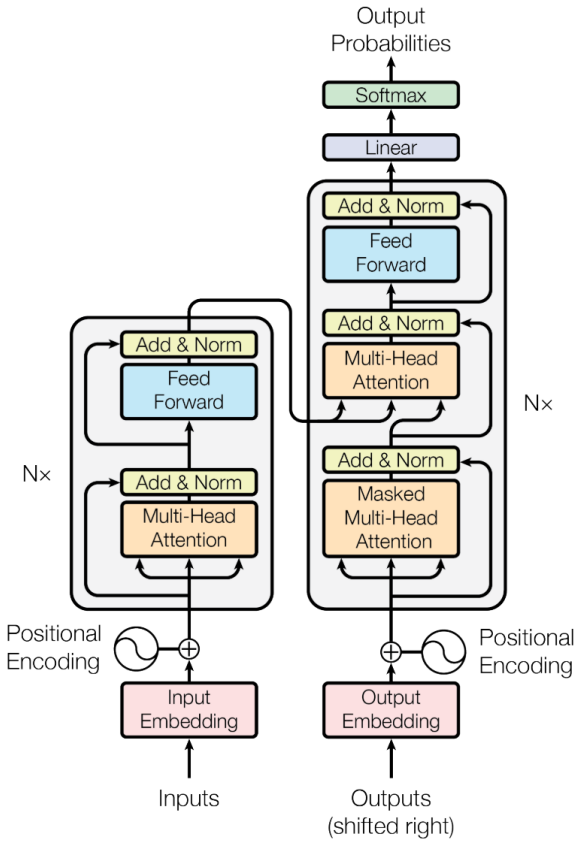


Figure 2.12: Transformer model architecture. [63]

2.4.8 Contextualized Word Representations

The application of Deep Learning in NLP was made possible by the representation of words as fixed-length vectors. This representation posed a problem wherein all senses of a word have to share one representation. In 2018, LMs like BERT [26] and ELMo [49] introduced contextualized word representations, which are context-sensitive. This means that representations of the same word in different contexts are different. These new word representations have greatly improved metrics in various NLP tasks. [28]

Contextualizing models

Below are short summaries on natural language models which use contextualized word representations. All three of these models are available pre-trained. [26].

- **ELMo** [49] - Embeddings for Language Models (ELMo) is built on top of an 2-layered LSTM. It uses the hidden states of both layers, and looks at the entire input sentence to extract context for a word vector. ELMo introduced the notion of deep contextualized word representations.

- **BERT** [26] - Bidirectional Encoder Representations from Transformers (BERT) uses a fine-tuning approach which means the model comes pre-trained, with minimal task-specific parameters and is applied to specific tasks simply by fine-tuning pre-trained

parameters. From the name, it becomes apparent BERT is built using a Transformer architecture. BERT is built using a transformer encoder stack.

- **GPT-2** [51] - Generative Pre-trained Transformer (GPT-2) also uses a fine-tuning approach built on top of a Transformer architecture. In contrast to BERT, GPT-2 is built using the transformer decoder stack. GPT-2 also behaves like a traditional LM - it outputs one token at a time. The way GPT-2 works is that after each output word, it is actually added to the input for the next word to be generated. This idea is also referred to as auto-regression.

2.5 Code completion with Neural LMs

Code completion is a challenging task for Language Modeling. In code, there are both very long distance dependencies, and local (to only one block of code) dependencies, like defining a variable only within one function. One proposed approach was combining a model using Attention (which was shown to outperform RNNs and LSTMs on modeling long distance dependencies) and pointer networks. [39]

Pointer Networks (Ptr-Net)[64]

Pointer networks are built on top of modified attention mechanism with pointers to input words. Figure 2.13 illustrates these pointers in image (b).

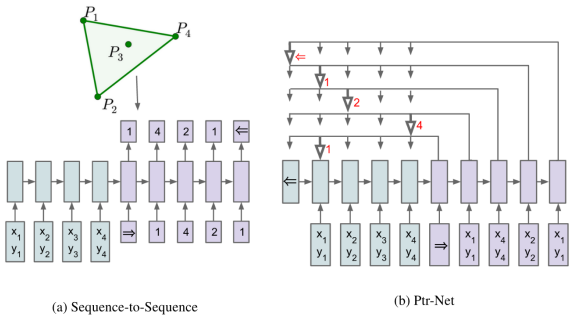


Figure 2.13: Comparison of a Transformer (sequence to sequence model with a Ptr-Net. [64]

Ptr-Nets provide a solution to the problem of predicting out-of-vocabulary words, by predicting words from the input sequence. Ptr-Nets however are unable to predict words outside of the input sequence, which means they are only suitable for predicting words occurring in local context (like a block of code)

Because of this limitation, a mixture of Attention (long dependencies) and Ptr-Nets (local dependencies) called the Pointer Mixture Network was proposed in [39] which predicts words either from the global vocabulary, or copies one from local context. This paper also proposed a modified version of the Attention mechanism which uses the structure of a programming language's AST.

Figure 2.14 shows that on Python code, the Pointer Mixture Network outperformed other language models (where TYPE = type of AST node, VALUE = value of AST node).

	JS		PY	
	TYPE	VALUE	TYPE	VALUE
Vanilla LSTM	87.1%	78.6%	79.3%	67.3%
Attentional LSTM (no parent attention)	88.1%	80.5%	80.2%	69.8%
Attentional LSTM (ours)	**88.6%**	80.6%	**80.6%**	69.8%
Pointer Mixture Network (ours)	-	81.0%	-	**70.1%**
LSTM [Liu *et al.*, 2016]	84.8%	76.6%	-	-
Probabilistic Model [Raychev *et al.*, 2016]	83.9%	**82.9%**	76.3%	69.2%

Figure 2.14: Comparison of the Pointer Mixture Network with other state of the art Language Modeling approaches. [39]

2.6 Code completion systems

There are a handful of code completion systems currently being used, below is a list a few such systems:

- **IntelliSense** [8] - Microsoft's implementation of a code completion / suggestion tool which is included in Visual Studio and VSCode.
- **Jedi** [5] - A python static analysis tool aimed at automatic code completion in python.
- **Deep TabNine** [1] - Deep TabNine is a closed source code completion system which was trained on GPT-2.

2.7 Common corpora used in LM

Language Modelling is a Natural Language Processing field with a lot of history, as was pointed out by the sections above. It is a very diverse field, and as such there is a multitude of different methods of evaluating performance of models and plenty of datasets for training such models on natural language. PapersWithCode [15] contains a list with some of the popular datasets for evaluating LMs. It can be accessed at [paperswithcode.com] Figure 2.15 shows a screenshot from paperswithcode.com which shows the leader board of the best performing LMs for various datasets. It is worth noting that GPT-2 occurs in half of the datasets in the screenshot, which alone highlights the fact that GPT-2 is a state of the art Language Model for various problems based on natural language.

TREND	DATASET	BEST METHOD	PAPER TITLE	PAPER	CODE	COMPARE
	WikiText-103	Megatron-LM	Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism			See all
	Penn Treebank (Word Level)	GPT-2	Language Models are Unsupervised Multitask Learners			See all
	enwiki8	GPT-2	Language Models are Unsupervised Multitask Learners			See all
	Text8	GPT-2	Language Models are Unsupervised Multitask Learners			See all
	One Billion Word	Transformer-XL Large	Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context			See all
	Penn Treebank (Character Level)	GAM-RHN-5	Recurrent Highway Networks with Grouped Auxiliary Memory			See all
	WikiText-2	GPT-2	Language Models are Unsupervised Multitask Learners			See all
	Hutter Prize	24-layer Transformer-XL	Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context			See all

Figure 2.15: Screenshot from paperswithcode showing a leaderboard of best performing LMs [15]

Chapter 3

End to end Models

The original goal of this project was to develop an open source implementation of an end to end model for code completion. Out literature review revealed that OpenAI's GPT-2 is most commonly used nowadays for similar applications and it hence became and obvious starting point.
GPT-2 is a transformer-based language model trained on 40GB of internet text to predict the next word in a sequence that can be fine-tuned for more specific applications such as question answering, text summarization and others [51].

3.0.1 Fine-tuning GPT-2

GPT-2 is not trained on code, so it requires so-called "fine-tuning", which adapts an already trained model for a new use case. In our situation, we took the model trained on internet text and tried to re-train it so it can model Python source code. To do this, we used an open-source library called `gpt-2-simple` [11].
As a next step, we had to select a suitable dataset. Our initial choice was GitHub's CodeSearchNet. [6] This dataset contains many languages, so we first extracted code written in Python3 which yielded a total of 400 000 Python functions. We pre-processed the dataset by removing all comments and docstrings in order to simplify the task of learning the language syntax, which is not followed inside comments. Finally, we created a corpus by concatenating all pre-processed python functions into a big file as described in next section

Training configuration

We experimented with fine-tuning various publicly available GPT-2 models, namely (124, 355 and 774 million parameter versions).
As input, we used the Python functions from CodeSearchNet, which we pre-processed by removing comments and Docstrings. Furthermore, we concatenated all these functions into one corpus which was used for GPT-2. In this corpus, the functions were separated by prepending `<|startoftext|>` and appending `<|endoftext|>` character sequences to each function in the file. These sequences are necessary for GPT-2 to be able to distinguish the beginning and end of each token sequence within our corpus. We then configured these sequences to be seen as separators by using `--prefix "<|startoftext|>"` and `--truncate "<|endoftext|>"` when generating text.
Since the corpus was quite large at over 250MB, we decided to encode the dataset into Numpy's compressed `.npz` format, which enabled us to load the data into a GPU faster.

Generating samples

The training code was configured to generate a fixed number of tokens of randomly seeded sample text every 100 training steps. After our experiments with fine-tuning GPT-2 for 10k steps, the results were quite disappointing and we noticed that, while sometimes returned syntactically correct code snippets, they were never longer than 1 or 2 lines at a time.
Apart from experimenting with all publicly available model sizes, we experimented with different parameters. However, to our disappointment, the models very often failed to provide syntactically correct code. One of the more successful runs is shown bellow:

```
            if not current.get(paths):
                paths.remove(current.get().join('', '<b>\{3.2f\}(\
\{\r
\}\w)\
\}')
           return self.get_paths(path.join("\x3d\y\x14\x12.0").split('', '\x12')).map(((n,
\}\w)\
\)\, path))
 <|endoftext|>

<|startoftext|>
def __debug__(self, **callback, **defeats):
        raise ValueError(
```

As we can see from the sample, the model definitely picks up the keyword, the syntax for defining methods or even invoking them. Nice example is the .JOIN(" snippet on line 2, where the model produces an idiomatic python expression for joining lists into strings with empty separator, which is very common in code. On the other hand, the model obviously failed in producing syntactically correct code and hence an engine backed by it would only create very sub-optimal user experience, where the user would need to come back and fix up many suggested completions.
One particular difficulty with predicting python is the importance of white-space. The block of code is marked by the size of its indent and our model failed to understand this, always producing snippets with random indentation. We discovered that this might be caused by dataset containing the mix of tabs and space based indents, but even correcting this flat in the pre-processing step did not help.
Our effort led us to consider possible issues. State-of-art GPT-2 model was trained on 40GB of internet text (over 8 million documents), so maybe we would need similar amount of data to achieve good results on code. However, acquiring this much correct Python source code as a dataset alone would be a giant task. A further issue would be function names and variable names - unlike in the English language, there can potentially be infinitely many distinct variable and function names. This would pose a problem as we would have to find a way to rename all variables in our dataset in a deterministic way to enable a language model to learn code structure without memorising specific names. This also applies to hard-coded strings and integers which would also have to be replaced.
Overall, we found that even after a lot effort invested into training GPT-2, the model failed to meet the quality standards we required for the completion engine.
After this discovery, we concluded that relying on a language model alone to ensure syntactically correct completions was near impossible and that additional mechanisms as well as more data and data pre-processing would be required to get closer to satisfying results.

3.0.2 Assessing the competition

Seeing these results, we decided to try out TabNine [1], a smart code completion tool built on top of GPT-2 and try to push it to its limits. We used TabNine in Visual Studio Code for some time while working on this project. What we quickly noticed is that TabNine mostly suggests function calls and snippets of code present in other files within a project. When working on code with a different structure and on stand-alone scripts, we noticed that TabNine sometimes returned out of place suggestions that did not occur anywhere in the project or in the working directory. One such example can be seen in 3.1.

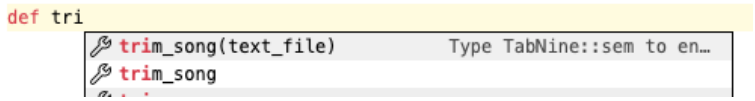


Figure 3.1: An out of place suggestion - `trim_song` did not occur anywhere in the project.

3.0.3 Looking for a better solution

This pushed us to the realisation that ensuring code completions are syntactically correct is more important than making the completions longer, because returning long but unusable code completions is not a sensible solution. Our approach would be revolving around returning one token at a time that is guaranteed to be valid in the location rather then returning long snippets that might not make sense. We thus had to rethink the approach we would be taking and we decided to scrap end to end models and instead try improving existing code completion libraries to yield better results. This would be done by using a code completion tool and training a model to sort predictions from such a tool so that the correct completion would appear as the first one.

Chapter 4

Implementation

This section gives details on the tools and libraries used in this project, describes our best performing machine learning model, how we trained it, how we built a code completion tool on top of it and finally lists various software engineering practises that were followed while working on this project.

4.1 Core Tooling

4.1.1 Code completion library

After the previously mentioned experiments with Language Models, we decided to tackle the problem of smart code completion by training a Machine Learning model to sort completions from a code completion library rather than have the model generate the completions itself. This decision was made because some code completion libraries complete one keyword at a time and this ensures that the completions are syntactically correct. If we decided to work with Language Models that directly generate completions, we would need to build a complex mechanism, on top of the model itself to ensure syntactic correctness and also limit the model, so it does not return too long snippets. We would end up with a mechanism filtering results of the model, which, at the end of the day, will produce mostly small snippets like the existing code completion library since long snippets are more likely to be incorrect.

Based on the reasoning above, we conducted the research on code completion libraries as the first step:

Microsoft Language Server

Microsoft Language Server is a server implementation that provides programming language specific features like code completion, go to definition, reference finding and many more.

Microsoft offers a language server for Python [10], which is implemented in C#. MS Python language server is commonly used as the python extension for Visual Studio Code [42] and Sublime Text, both of which implement clients of this language server.

Language servers support MS Language Server Protocol (LSP) [12], whose aim is decouple language server implementations from their IDE plugins and enable seperate development of a language servers and extensions for IDEs to communicate with language servers.

Jedi [5]

Jedi is an open source static analysis tool written in Python for Python with a focus on code completion. It has a simple API and offers support for all aforementioned text editors and IDEs (and many more).
We decided to use Jedi as the library from which we would be getting our code completions since Jedi is written in Python while MS python language server is written in C#. This means that using Jedi is as simple as importing it into Python, while we would need LSP client to use MS python language server, adding another layer of abstraction and increasing complexity of out system. It is worth noting that Jedi returns the list of completions sorted alphabetically which means that the desired completion might not always appear first in the list. Furthermore, we discussed the choice of completions provider with Deepnote and the company advised us to to use a solution that consumes as little resources as possible. Based on our experiments, memory footprint of Jedi is up to 10x smaller than memory footprint of MS language server which made Jedi the obvious choice.

4.1.2 Machine Learning framework

When working on a Machine Learning project, it is crucial to select a suitable ML framework. Below is an outline of the most popular ML frameworks along with a comparison between them.

Tensorflow

Tensorflow [18] was created by Google and is one of the most well-known ML frameworks. It currently has 145,000 stars on GitHub. Currently, Tensorflow is widely adopted by both industry and researchers. In Tensorflow, models are viewed as a Directed Acyclic Graph. These graphs have to be defined statically before a model can be run. A disadvantage is that it is quite low-level and requires writing boilerplate code to get started with working on ML models. Tensorflow comes with Tensorboard, a tool useful for visualization and debugging of Tensorflow models.

PyTorch

PyTorch [48] is a newer ML framework that differs from Tensorflow by the fact that its representation of models can be modified dynamically. For example, this can benefit Recurrent Neural Networks. While in Tensorflow, all input sequences for an RNN have to be zero-padded to the same length, in PyTorch it can vary. PyTorch can be debugged by using standard Python debuggers, while Tensorflow can only be debugged using tfdbg, a specific debugger. Similarly to Tensorflow, PyTorch is a low-level framework and training steps like gradient descent have to be implemented by the developer.

Keras

Keras [24] is a high-level library that can be run on top of multiple ML frameworks, including Tensorflow. It was developed in order to enable fast iteration and experimentation with various ML models. Keras contains fully configurable implementations for all standard ML layer types - RNNs, LSTMs, Convolutional layers, Dense layers etc. Keras also has a Functional API which enables users to create more flexible models with multiple inputs, multiple outputs, shared layers, etc.

We decided to use Keras for this project because it offers a very simple API which allows us to create various model architectures quickly. Furthermore, there are many great Keras tutorials and sample implementations. Getting started with Keras was easy as we also used it while working on previous coursework.

4.2 Data pre-processing

This section describes the final method of pre-processing data that was used in our best-performing model.
In order to enable us to use a new dataset quickly, we created a pipeline which takes a Python library and transforms it into a dataset suitable for our model. Our best-performing model takes two inputs: 40 latest tokens from the code written so far and a completion from Jedi.

Step 1: Cleaning the data

In order to focus on purely the problem of saving keystrokes by completing code, all datasets were automatically pre-processed by removing all Docstrings and comments. Furthermore, all files are then compiled using `py_compile` [16] to ensure that they are still syntactically correct. Errors might happen when strings are defined using Python's triple quote (`"""some string"""`) notation. Any files that have errors in them will be pointed out by this step and are then fixed manually so they compile.
This occurs if source code contains a string defined using the triple quote notation `x = """ Sample string """` the pipeline will change it to `x =` which will not compile for obvious reasons. We decided to keep this step in as the pre-processed code is cleaner if we remove docstrings entirely as opposed to making these strings empty and keeping the quotes.

Step 2: Splitting the data

The files are then split into Train, Validation and Test splits using a 80/10/10 split. This step includes listing all Python files in the library, shuffling the list and moving the files into `train`, `val` and `test` folders.

Step 3: Shuffling and preparing data for our model

The data is then feature-engineered for our model, by generating a dataset as follows:
In this step, we iterate over a Python file character by character, and for each character:

- Take the latest 40 previous tokens as `token_stream`
- Query Jedi to get completions on the unfinished token. (each of these completions makes up a sample) If Jedi returns no completions this sample is omitted.
- Take the actual token that is to be completed - this is necessary for step 4.

In this step, we also filter out all tokens that occur less than 5 times in the dataset (a hyperparameter we experimented with). These are mostly variables and strings. After all files have been processed in this way, all lines from these files are shuffled.

Step 4: One-hot encoding

Finally, in order to input this data into our model, we convert the data into a suitable format. Since we are dealing with text data, we decided to use one-hot encoding for our data. One-hot encoding text data is a common practise in NLP tasks and is described in [61]. This step is done in our data loader which is called directly by the model when it is being trained. The steps performed by our data loader are:

- One-hot encode and zero-pad the sequence of 40 previous tokens.
- One-hot encode the Jedi completion
- Compare the Jedi completion to the actual token to be completed. If they are equal, the label is 1 otherwise 0 - these are our labels for binary classification

The data loader then feeds this data into the model both for training and validation.

4.3 Model Design

Now that we defined data pre-processing, we will describe our neural architecture. The model given below performed best on our evaluation and is a result of comparing various different model architectures, hyperparameters and ways of pre-processing our data which is described in Chapter 5.

Inputs and Outputs

In order to sort completions from Jedi, we decided to create a model that performs binary classification on the completions. It takes two inputs, a sequence of 40 latest tokens that occurred in code and a completion from Jedi. The model's output is 1 when the completion matches the actual token that is to be written or 0 otherwise. Since the model will return numbers between 0 and 1, we can sort Jedi completions by the 'score' our model gives them, since these scores express the 'confidence' of our model that a specific completion is the correct one.

Model Architecture

Figure 4.1 illustrates the architecture of our model. The first input is labelled tokens and it has shape 40x1991 where 40 represents length of the input sequence and 1991 represents the size of the one-hot encoding. After removing all tokens from the data that occur less than 5 times, we have 1991 distinct tokens. Tokens are then fed into an LSTM which learns a representation of the sequence. The second input is labelled completion and it takes one Jedi completion which is also one-hot encoded. We then concatenate the completion and the LSTM's representation of Tokens, which is fed into a Dense layer. The dense layer performs binary classification: If completion is the one that will be selected, return 1 else 0.

4.4 Training the Model

We trained our model on Deepnote using a Nvidia T4 Tensor Core GPU. The model was trained with Keras' `fit_generator` function, as we could not fit the whole dataset into memory. The data loader also performed the final steps of pre-processing of the data (encoding

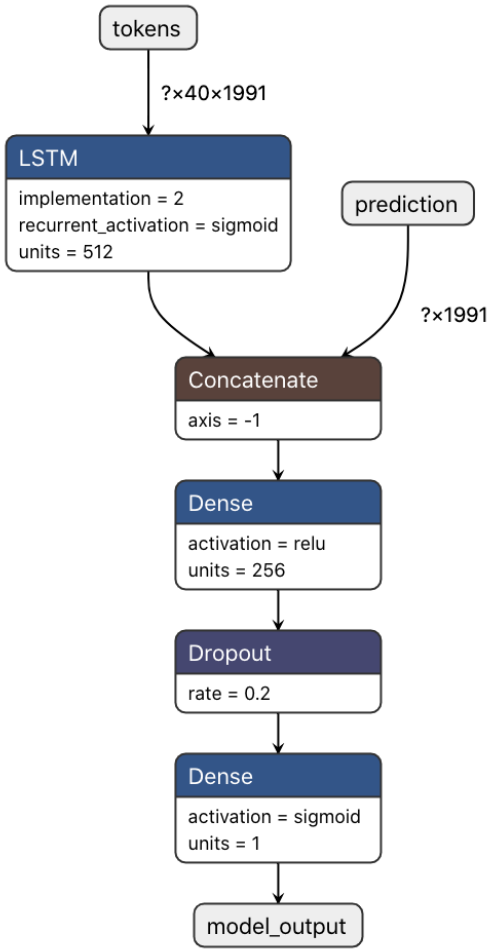


Figure 4.1: Architecture overview of our best-performing model with shapes of the inputs and parameters of the layers.

and padding). We noticed that when training our model, the validation loss increased with each epoch, so we decided to train the model for 1 epoch as this yielded the best performance in the evaluation.
When we trained our model for more epochs, it started overfitting on the training data and validation set accuracy was dropping. The main cause of this could be that instead of trying to correlate the previous tokens and Jedi completion, the model might have started memorising labels (0 or 1) for specific completions or for specific token sequences.

4.4.1 Hyperparameter Search

Apart from experimenting with different architectures, we did extensive hyperparameter search on these parameters:

- **Sequence Length** - We experimented with different lengths of sequences of most recent tokens. Generally, the longer the sequence the better our model performed.
- **Filtering tokens based on count** - We experimented with various minimum token counts. This means that if a token occurs less than N times in the dataset, we omit it

from the input sequence. This step also alters the one-hot encoding size. After experimenting with larger and smaller thresholds, we reached the conclusion that filtering out all tokens that occur less than 5 times in the dataset yielded the best results.

- **Number of neurons** - We also experimented with different numbers of neurons in both the LSTM layer and the dense layer after concatenation. To our surprise, altering the number of LSTM units barely affected the performance of our model (only about 0.2%). A possible cause of this is that the model gives much higher priority to the actual Jedi completion than the input sequence.
- **Regularisation** - In order to try and reduce the importance our model assigned to the completion input,, we experimented with dropout regularisation. When we set dropout too high (above 0.4), the model failed to understand our data and accuracy dropped significantly. However, setting dropout to 0.2, we achieved our best result.

4.4.2 Autokeras

Apart from using hyperparameter search, we also experimented with using AutoKeras [32], a neural architecture search system. AutoKeras is a pre-release tool which is aimed at automatically finding best-performing neural architectures. Initial experiments showed that AutoKeras was capable of finding convolutional neural network architectures which performed 1-2% worse on the test set than our custom models, so we deemed it reasonable to assume that trying out more models could match or even beat our models. However, in order to represent our desired inputs and outputs in the AutoKeras API, we had to use the AutoModel interface, which had problems exporting its models, as it relied on an unreleased Tensorflow version. Further problems occurred after trying to save these models in at the time all release candidate versions of Tensorflow as neither exported these models without errors.
Because of uncertainty when these issues would be fixed and a limited time frame of this project, we decided to scrap further experimentation with AutoKeras. Perhaps when AutoKeras is released this will be worth investigating further.

4.5 Code completion system implementation

After we created and trained a machine learning model which is able to score Jedi completions using context from the code given, we built a code completion system using Jedi and our model. Below are the steps our system performs to return code completions.

- **Initialization** - The system is initialized by loading a model architecture, trained weights and a file that represents the one-hot encoding for tokens.
- **Getting the completions** - When queried, the system takes code written so far and queries Jedi for completions.
- **Model input** - The system then takes the input code, tokenizes it, trims it to length 40 and encodes it the same way that was used during train time. The Jedi completions are also encoded.
- **Sorting the completions** - For each completion, the model returns a score between 0 and 1. These are then sorted in descending order based on the score and returned.

4.6 Completion sorting language server

Previously, when IDE developers wanted to include support for a programming language, they had to implement it themselves (from integrating it into the editor to implementing specific language features). Language servers (LS) provide a clean solution to the problem of bringing useful tools like code completion, refactoring and go to definition to editors and IDEs.

Microsoft proposed a solution to this problem by using Language servers and creating a protocol that defines communication between two parties - Language servers and their clients (editors and IDEs). This solution removes the need to implement language support separately for each editor.

4.6.1 Language Server Protocol (LSP) [12]

Originally developed by Microsoft for their Visual Studio Code [42] editor, this protocol defines communication between editors and language servers.

By using LSP, the work that needs to be done is split into two parts. First, the programming language developers can implement a server which, using LSP, can provide programming language specific features to any client. Meanwhile, all that a developer needs to do to integrate such functionality into their IDE is write a plugin, which acts as a client to a language server without having to worry about implementing any language-specific features.

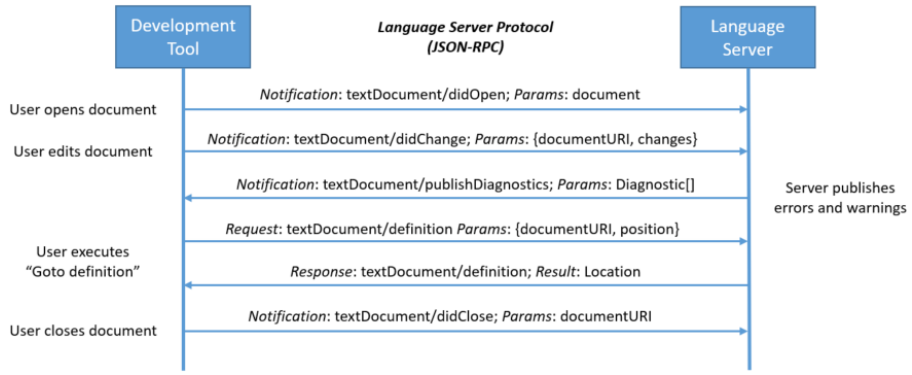


Figure 4.2: An example communication between a language server and an editor [12]

Figure 4.2 shows an example communication between a client and a language server. The communication begins with the editor notifying the LS that a file has been opened. Importantly, the contents of the source code file are now stored in memory of the LS and the client has to notify the LS about any changes to the file. In the example above, the users sends the LS a go to definition request which contains the location of the currently edited file and the position of the token whose definition is being queried. The server then responds with the location of the desired file and the position of the token's definition within that file. The editor then notifies the LS that the current file has been closed.

For developers that want to implement support for a specific language into their editor, Langserver [9] provides an overview of open source LS implementations along with which language features they support.

4.6.2 Extending a language server

As Deepnote currently uses a language server to get code completions for users' projects, we decided to extend a LS with our completion sorting mechanism, so it can be easily tested by Deepnote's user base. For this task, we used Palantir's Python Language Server [14] for two main reasons. Most importantly, it is implemented in Python, so we could easily use our model for sorting the completions. This LS implementation uses Jedi for its code completions, just like we do in this project.
The steps it took to include our sorting mechanism into the LS is described below.

- The files containing our model's architecture, its trained weights and the one-hot encoding definition were included into the package of the LS.
- Upon initializing the LS, our model's architecture and weights are loaded.
- We modified the language server's Jedi interface in order to return the code written so far, not just the completions. This was important as our model uses this context when sorting completions.
- The handler for completion requests was extended to perform steps needed for our model to sort completions (tokenizing and trimming the input sequence, one-hot encoding inputs).
- Before the completions are returned, they are sorted according to our model's output.

4.6.3 Developing against Visual Studio Code

The LS implementation we decided to extend comes packaged with a plugin for VS Code, which enabled us to easily test our extended language server in real time. Immediately after starting work on this task, we noticed that VS Code re-sorts the completions it gets using a custom, fuzzy scoring function. This alters the order of completions that are showed by also scoring permutations (swapping out of two adjacent characters) of the token being written to account for the developer making typos. Even after disabling this feature, VS Code re-sorted the completions we returned in an alphabetic order. We carried on by only using VS Code as a means of querying our LS with completion requests. Instead, we were verifying the returned completion sort order by logging it before returning the completions. This is still a valid approach as compared to VS Code, the text editor in Deepnote currently sorts code completions alphabetically, but this can be easily disabled.

4.7 Software engineering practices

Working on this project required applying and acquiring new knowledge in software engineering and data science. Below is a summary of the software engineering practises applied while working on this project

Automation

Since creating and training machine learning models is time consuming, we decided to create pipelines to automate many of the steps in order to speed up the way we iterate on various datasets and models. We created a pipeline for most of the many tasks that have to be

performed - from pre-processing a dataset, through training a model up to evaluating the model, like mentioned in Section 4.2.

Unit Testing

In order to ensure that our code and evaluations are fair and correct, we thoroughly tested our code by creating unit tests to verify if all the 'moving parts' of the project behave as they should. For example, we tested the evaluation pipeline with deterministic 'code completors' to verify that our evaluation tool correctly logs every keystroke and that the saved number of keystrokes is calculated correctly in all edge cases.

Extensible code

The aim of the our evaluation library is to enable developers to compare their code completion systems on the same code, so we created a clean, simple interface for 'Predictors' - objects, which given an incomplete snippet of code, return a list of possible completions for that snippet of code. The Predictor interface simply requires a `.predict()` function, which takes a code snippet, some auxiliary information about the code and returns a list of completions. A custom constructor can of course be added if the user would like to load additional data or run a child process while initialising the Predictor. More information on the evaluation library is given in Chapter 6 and in Appendix A.

Version control

As we were working iteratively, it was important to version control our code so we could easily track changes and discover when and where we introduced bugs. When training machine learning models, we saved the architectures and weights in files named after the model architectures and parameters for quick lookup.

Chapter 5

Data and Evaluation

5.1 Evaluation Method

Since the aim of this project is creating a tool which speeds up writing code, we decided that a suitable method of measuring how useful our tool is would be counting how many keystrokes (in %) our tool saves, as this represents the purpose of such code completion libraries - speeding up the process of writing code. A description of the steps of our evaluation algorithm used to measure this metric can be found in Section 5.1.2
Since our chosen metric is a simulation of real-world use, we created a library which greatly simplifies comparing and evaluating code completion based on various models. An explanation of how this library works is given below:

5.1.1 Evaluation Toolkit

The evaluation toolkit we created simulates our code completion system being used in a code editor. Given a model, and a path to test files, the tool simulates code in the test files being written character by character, each being counted as a keystroke. On every character, the given model is polled for a list of completions. If the list contains a valid completion (which is easily detected as the evaluation tool knows ground truth of the file being written), it can be selected and the tool calculates how many keystrokes were saved. The complete algorithm for calculating keystrokes saved is described below.

Counting keystrokes

During the evaluation, every character written counts as a keystroke. Like in most editors and IDEs, selecting a completion involves either pressing TAB or ENTER and hence counts as a keystroke. Furthermore, selecting a completion from the list of given completions involves using the arrow keys to highlight the desired completion in the list. Each press of the arrow keys also counts as a keystroke. It is worth noting that the keystroke counter always chooses the minimal possible amount of arrow key presses - as many as it takes to get to the correct completion from the top of the list. (more detail is given below)

5.1.2 The algorithm

The evaluation tool we created determines whether to select a completion from the list as follows:

- Calculate how many keystrokes it takes to complete the current word
- Calculate how many keystrokes it takes to select the correct completion. As mentioned above, this takes into account selecting the completion (1 keystroke) and using the arrow keys to navigate to the correct completion (this takes keystrokes equal to the index of the desired completion in the list.) In this step, we save the index of the correct completion in a dictionary.
- If selecting the correct completion takes less keystrokes than writing the token, it is selected and we move to the end of the token. Otherwise, we simulate writing another character.

Examples

To clarify the approach, below are two examples, supported by figure 5.1:

1. Assume the correct token to be completed is `end_line`. The red numbers in 5.1 represent how many keystrokes it takes to select the correct completion. In this case, it would take 7 keystrokes to complete `end_line`, but only 3 to select it from the list - we saved 4 keystrokes.
2. Now assume the correct token is `eval`. Selecting it from the list would take 5 keystrokes, while completing it would only take 3. In this case the model does not count any keystrokes as saved.

This approach obviously has its limitations, since not every key-stroke is created equal. Most programmers are used to touch typing, so exiting their home row to select a completion using arrows is certainly more costly than just selecting the top suggestion with space or tab. Similarly, the keystrokes themselves have different cost, where the keys on home rows are the cheapest, the keys above and below come next, and keys that require lateral movement of the wrist come last, since they aggravate common strain injuries of developers (eg. Carpal tunnel syndrome). We decided to neglect this in our metric because the ”cost” of pressing a key is largely a matter of personal preference and also it cannot be generalized across many keyboard layouts (eg. QWERTY, Dvorak, Colemak). However, we later added this in an auxiliary metric which measures how many desired completions are in the head of the returned list.

Measuring relevance of sorted completions

We later created another metric for evaluating our model which is based on measuring the likelihood of the desired completion appearing first (or in the top-N) in the list of completions, if it is present. This is done by logging the position of the desired completion each time it appears in the list of completions.
We decided to include this metric as it seems natural from our experience using code completion tools. We believe that a user is far more likely to select a completion if it appears first in the list, since this only requires the user to press enter or tab, rather than having to use the arrow keys to select a completion which is lower in the list.
After running the evaluator, it returns the average number of keystrokes saved over the test files, which can be directly used to compare the performance of various models. The result also contains a Python dictionary, which maps an index in the list of completions to the number of times the desired completion occurred at that index.

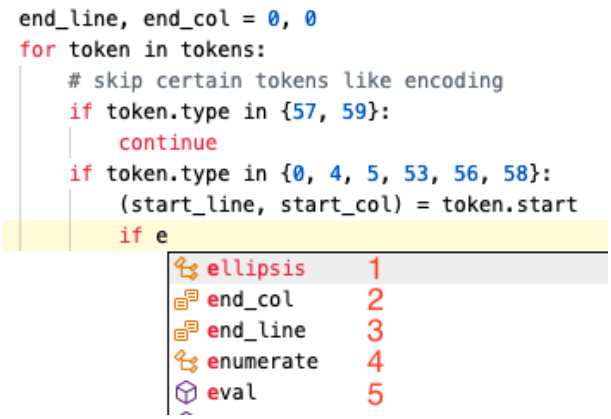


Figure 5.1: Example code completions - The red numbers on the right of the dropdown menu represent how many keystrokes selecting a certain completion takes.

5.2 Data used

5.2.1 CodeSearchNet [6]

We originally intended to use GitHub's CodeSearchNet dataset, which contains around 400,000 separate Python functions (it also contains Javascript, Ruby, Go, Java, and PHP code). CodeSearchNet was created as a dataset for the task of code retrieval using natural language. Since it contains a vast amount of Python functions, we anticipated it would be a good dataset to train our code completion models on.

After experimenting with Jedi and our models, we realised that the dataset is not well suited for our problem - as it only contains separate functions and it lacks both larger context and imports.

In Python, a `import name` statement does two things. First, it searches for a module named `name` and if it is found, a `module` object with its functions and variables is created in the local scope. [7]

For Jedi, imports are important as it parses these statements and includes functions from the imported modules in the returned completions. After running Jedi to see how many keystrokes it can save when simulating the writing of these functions, we found out that because the lack of context and imports, Jedi could only save less than 10% of keystrokes.

5.2.2 Tornado [17]

After seeing the results on CodeSearchNet, we decided to instead use an actual Python library as a dataset, as it would provide us with coherent, structured code that also contains imports. We selected Tornado, since it is an extensive and widely popular library used in many Python projects. Main advantage is that it mostly uses Python as opposed to also invoking C-Python bindings used in other popular libraries. This decision is further supported by the paper "On the localness of software" [62], since it also mentions using the Tornado library.

The dataset consists of the 114 Python3 files found in the Tornado library.

5.3 Evaluating our approach

Now that we defined an evaluation method, fixed a test dataset and implemented our evaluation pipeline, we present a comparison of the performance of various code completion systems evaluated using our approach on 12 randomly selected files from the Tornado library.
It is worth noting that all systems were evaluated with files that do not contain comments or docstrings, as these would add more keystrokes but not be completed. The numbers in % are given as number of keystrokes written in comparison to having to write all keystrokes (denoted by Predictor=None)
In order to evaluate a code completion tool, all we have to write is a wrapper for it so our evaluation toolkit can query it for completions. These wrappers will be referred to as Predictors below.

Predictor	Keystrokes	$P(dc_1\|dc)$	$P(dc_{top3}\|dc)$
None	100%		
Jedi	84.59%	40.98%	63.99%
Count-based Jedi	81.97%	60.77%	85.45%
Jedi sorter (LSTM)	**80.98%**	**67.01%**	**90.13%**
Ideally sorted Jedi	78.57%		

Table 5.1: Comparison of various models and an ideal case.

- **Jedi** - A predictor which queries Jedi and returns the completions in the same order they were returned by Jedi

- **Count-based Jedi** - This predictor takes an auxiliary file, which contains the counts of all tokens present in the train set. It sorts Jedi completions according to their counts. Interestingly, this very naive approach offers a significant improvement over Jedi.

- **Jedi sorter (LSTM)** - A predictor built from our best model according to Section 4.5. Adding the sequence as an input in addition to the Jedi completion alone allowed us to further increase the number of keystrokes saved.

- **Ideally Sorted Jedi** - For this result, we modified our evaluation pipeline to simulate how many keystrokes we would save if the correct completion from Jedi would always appear first in the list of results - that is, selecting the correct completion would always take just one keystroke.

In order to explain the difference between our Jedi sorter and Ideal Jedi, it is important to mention that we used 12 Tornado files as raw code our evaluation toolkit to get the results in Table 5.1. For the same 12 files, we also pre-processed them the same way model input was pre-processed so we could directly evaluate the model without wrappers. When we evaluated our model on the pre-processed test set, we achieved accuracy of around 88%. This result offers an insight as to why there is a difference of 2.41% in our evaluation above.
We also included two more columns in the table representing our second evaluation metric. The column $P(dc_1|dc)$ denotes the likelihood of the desired completion appearing first in the returned array, given that it is present. The column $P(dc_{top3}|dc)$ represents the likelihood of a completion occurring in the first 3 elements of the list, given that it is present in the list.

5.3.1 Plotting the likelihood of a completion's index

As we mentioned previously, when selecting a code completion that does not come first in the list of completions, arrow keys have to be used to navigate to said completion. Using the arrow keys requires the programmer to move their hand from the neutral position that many use in touch typing. Studies have shown that lateral movement (or any movement) of the wrist while typing can have a negative effect on it. This can lead to various strain injuries like carpal tunnel syndrome. Research [40] has shown that keeping wrists in a neutral position can significantly reduce wrist strain. This further supports our focus on returning the desired completion first in the list, as selecting it does not require moving wrists from this neutral position since the tab key is easily reachable from it.

Apart from measuring the percentage of keystrokes compared to not having code completion in place, we also measured the positions of correct completions in the returned list. We generated 2 bar charts which directly compare these likelihoods for Jedi and our Jedi sorting model.

Figure 5.2 illustrates the likelihood of the correct completion occurring at index `i` in the returned list. As the chart shows, the likelihood of the correct completion being at index 0 is far higher for our model than for Jedi alone. Similarly, the likelihood for index 1 is also higher than Jedi's - a desired result, since selecting a completion at index 1 only takes 2 keystrokes. This chart highlights how our model managed to reduce the likelihood of the correct completion occurring at indices greater than one, which is desirable as these are rarely selected.

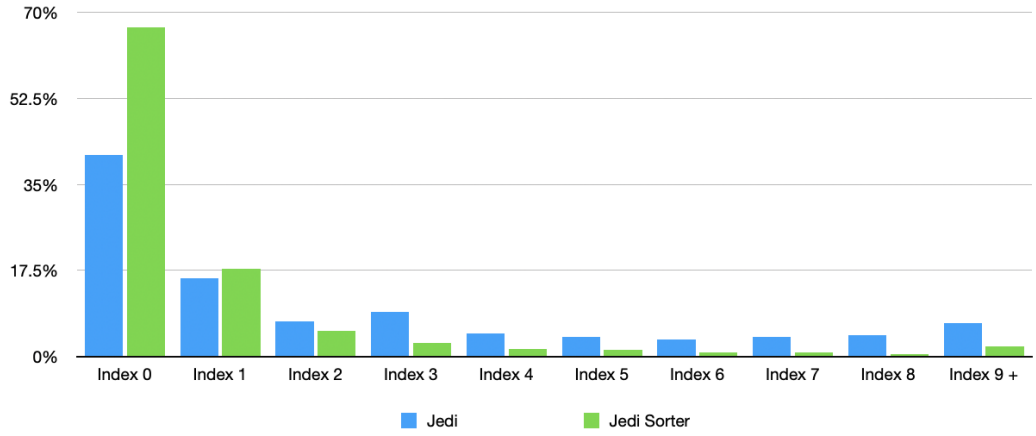


Figure 5.2: Given that the list of completions contains the desired one, comparing the likelihood of the desired completion being at a specific position in the list.

In figure 5.3 we took the same data but plotted it in a cumulative manner, which illustrates the likelihood of the correct completion being in the N first elements of the list.

Note that these charts and comparisons were generated from running Jedi and our Jedi sorting model on the same test dataset. However, we present these numbers normalized in % rather than absolute counts as the number of times these two completion systems returned the desired completion differs. This comes from the fact that the following can happen: our model returns the desired completion at an index where selecting it takes less keystrokes than writing it, so it is selected and the keyword is skipped. Jedi on the other hand returns a list of completions, where the desired one is at a large index and selecting it takes more keystrokes than completing the keyword manually, so our evaluation tool simulates writing

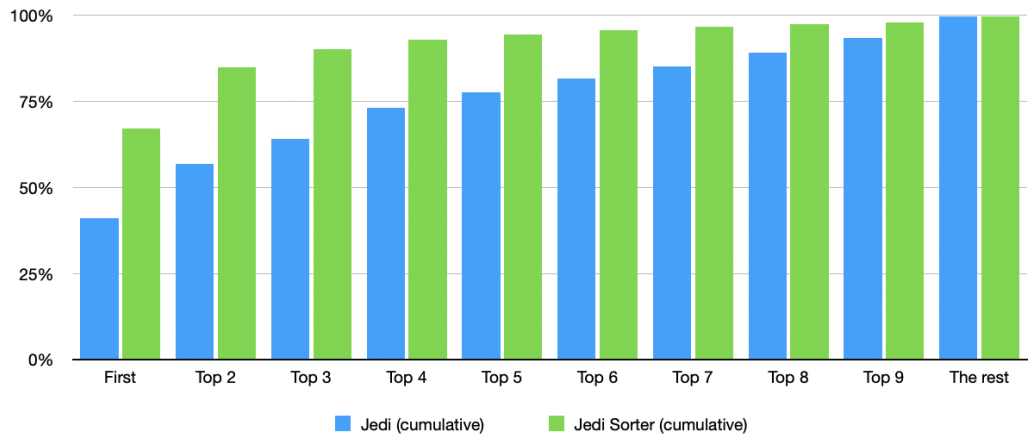


Figure 5.3: Given that the list of completions contains the desired one, comparing the likelihood that the correct completion is in the top-N of the list.

another character - querying Jedi again, which generates another entry of the position.
To put this result into perspective, assume an engineer that is writing code 4 hours daily. Our model would then save them around 7.2 minutes per day - which amounts to over 30 hours annually.

5.4 Comparing implementations

As we were comparing different architectures (LSTM, LSTM with Attention, CNNs) and hyper-parameters (LSTM units, dropout, number of conv. layers, kernel size etc.), we noticed that most models' test set performance was between 86 and 88%. The most probable cause of the similar performance of all models comes from the architecture of our model, mainly the concatenation.

TabNine

We also decided to write a Predictor built on top of TabNine so we could compare the performance of our models against TabNine. After running the evaluation pipeline on the same test files as above, we realised that TabNine only needed 58.04% keystrokes (72.02% when we passed in the project path instead of the file path). This is a surprisingly good result. However, one reason why we might have got this result is the following. We are passing the (absolute) path to the file currently being simulated to TabNine. Since all other files from Tornado are complete and present in the same path, it might occur that the file we are currently simulating defines a function, which is already used in another file. Since TabNine sees this function being called in another file, it can suggest the function's name (and parameters) when we are simulating it being defined. A function definition with its parameters is usually quite a long line, so this saves a lot of keystrokes.
However, this rarely happens in the real world - we usually do not (and cannot) call a function before it is implemented. This leads us to believe that TabNine results are slightly inflated, and if the presence of such function calls were eliminated, the number of keystrokes saved would drop.

Chapter 6

Autocompletion Pipeline Gym / Development Framework)

Creating and training machine learning architectures is a time consuming task. However, when trying to find a machine learning-based solution to a problem there is a lot more work to be done - namely pre-processing data and creating a suitable evaluation method. Implementing these takes time away from creating and training machine learning models as you need data in order to train them and you need an evaluation method to assess their performance and compare them.

OpenAI Gym

Gym [22] is a library for developing and comparing reinforcement learning algorithms. It provides an environment with a collection of test problems that allows the comparison of general reinforcement learning algorithms.
Gym has proven to be a very useful tool for many reinforcement learning researchers and engineers, which is highlighted by the fact that their GitHub [13] page currently has 20.9 thousand stars and the paper introducing Gym was cited over 1400 times.

Our proposal

Since the aim of our project is very specific, there is no existing toolkit or library for comparing different models or code completion systems. We decided to make our pre-processing and evaluation pipelines open-source to enable other researchers and engineers to train and compare code completion systems on datasets of their choice. Our implementation does the following:

- Provides resources for converting source code into a dataset suitable for training models for code completion.
- Provides a simple API so a wrapper can be written for many existing code completion systems and new models alike. Furthermore, given a path or list of source code files, it calculates how many keystrokes the system saves and returns the positions of desired completions in the list of completions returned.
- Enables researchers to directly compare code completion systems and models on any source code data.

6.1 Dataset Generator Pipeline

This pipeline is Python-based tool which can be easily configured by users using various parameters to transform source code to datasets with user-specified parameters (data points). The pipeline begins with cleaning source code by removing comments and docstrings. Splitting the data into train, validation and test splits is also handled by the pipeline. If the user desires to do so, the library can also split the data into files that contain equal numbers of data points. (We found this to be useful when working with large amounts of data and data loaders). Each data point is represented by a single JSON object.

Customizable parameters

- **Path to source code** - Path of the library that will be converted into a dataset.
- **Splits** - Determine how many % of source code files should be in the train, validation and test sets. The default is set as 80/10/10 respectively.
- **Data features** - The user can specify what features should be extracted from the source code for each data point and how they will be stored in the JSON output. For our model, each data point contains the following features: 40 most recent tokens, a Jedi completion and the correct next token.
- **Shuffling and splitting** - The user can decide whether data should be kept as is (by default data generated from one source code file is kept in one file) or if data should be split into equally sized files.

6.2 Autocompletion Evaluator

We have created a library which enables users to directly compare code completion systems and models. It consists of two parts listed below.

- **Predictor API** - The predictor API is a simple interface with just one 'predict' function. It enables the user to write a wrapper for their code completion system so it can be evaluated using the evaluator. The user can initialise their predictor with any auxiliary data - whether it is a model and weights, or a path to an executable.
- **Evaluator** - A library which contains a Counter object. All the user has to do is call the Counter object, passing it two arguments.
    1. Predictor - a wrapper for the user's model implementing the interface described above.
    2. A list of files or a path to a library on which the evaluation should be performed.

We hope that our contribution will make it easier for other interested researchers to get started working smarter code completion systems.
An example Predictor implementation is given in Appendix A, where we are implementing the baseline of our project - a predictor that returns the completions from Jedi.
Our open-source evaluation pipeline is available at https://github.com/borisbarath/code-completion-gym

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Overall, we consider this project a success. Even though the approach we would be taking had to be reconsidered, this has enabled us to explore various language modeling techniques. A great importance was placed on creating a pipeline that would enable us to work in a more streamlined fashion. We managed to create an extensive toolkit for the problem at hand - from data pre-processing pipelines up to a tested, deterministic code completion evaluation pipeline.
Our model built on top of Jedi, which using our metric managed to outperform both Jedi itself and some simpler models based on top of it, has achieved our main goal. The model saved over 3% keystrokes over Jedi, which yields 23% improvement relative to it. This result proves that sorting Jedi completions is a good way of increasing their relevance to the code written. Sorting Jedi completions further increases the likelihood of the desired completion occurring in the top 3 places in the list of completions by over 26%, if it is present. Most importantly, the likelihood of the correct completion being first in the list, if present, was increased by 23%. We have shown how our model can potentially save more than 30 hours per year over using Jedi, along with reducing wrist strain.
Finally, pre-processing and evaluation pipelines are now open source so researchers interested in this topic can easily get started with comparing their models against each other.
While working on this project, it had become apparent to us that apart from the model's architecture, the steps taken during pre-processing of text data are crucial as they often have a very large impact on model performance.

Deepnote

The model we developed is now being tested with a subset of users in Deepnote. The results are not yet available at the time of writing this report. The main concern would be the extra load on the infrastructure caused by running the model and increased latency of the system. We are going to be evaluating this trade-off based on real user data and feedback as a part of future work.

7.2 Further work

Since pre-processing large amounts of data and training machine learning models is time consuming, there are some approaches that we thought about but did not quite manage to

evaluate during the duration of this project. Here, we would like to list some of these ideas on how smart code completion can be pushed to save even more keystrokes.

- **GPT-2 + syntax mechanism** - As we mentioned in Section 3.0.1, relying on a language model alone to provide syntactically correct completions is not recommended. Instead, it may be worth investigating whether it is possible to write a simple mechanism that would take outputs from GPT-2 (or any language model) and extract useful information from what the LM predicted.
- **AutoKeras experiments** - Unfortunately, during the time frame of this project, AutoKeras was not in a state of development where the parts that we needed to experiment with it were usable. After many tries with all available pre-release versions of Tensorflow and attempts at fixing the issue with saving generated models, we have not managed to get it running by the deadline of this project. In terms of future work, we will continue trying to find a fix for this problem. Alternatively, when the required version of Tensorflow gets released and AutoKeras gets updated, resuming this experiment may be worthwhile as AutoKeras generated some interesting neural architectures.
- **Enhancing our predictor** - One potential improvement we thought about is enhancing our predictor by adding a mechanism that would rename variables into fixed names like `VAR1` and `VAR2`, similarly to what is done in [25]. If we would then feed this modified code into our model. This would potentially improve our model's performance as it could sort Jedi predictions even in cases where it currently does not (since variable names often occur infrequently and we filter out any tokens that occur less than 5 times, most of them are not present during training our model). During testing of our model we could do the same by parsing incomplete Python code with Parso [3] and renaming the variables.
- **Optimization** - There are a number of improvements that could be done to speed up our code completion system. For instance, the code that was already written could be stored so we don't have to re-tokenize and re-encode the whole sequence every time we query for completions.
- **Re-training on current project** - Since training time of our model is (at under one hour) relatively fast and after performing some optimizations on the pre-processing pipeline, it might be possible to build a mechanism which would train on demand a model on a Deepnote user's code base, which would allow for improved precision and relevance of compeltion sorting.

Appendix A

Example Predictor Implementation

Below we give sample implementations of a custom Predictor (in our case, one that uses Jedi) and then we will show how to the custom predictor in the evaluation tool. All that needs to be implemented is the `.predict()` function, which takes the following arguments: (Please note if your model or tool requires more information to predict code, feel free to let us know or submit a pull request to the repository containing these files.

- **line** - Text written in the edited file so far.
- **lineno** - The line on which the cursor currently lies. The evaluator uses zero-indexed lines.
- **column** - The column in which the cursor currently lies. Again, the evaluator uses zero-indexed columns.
- **limit** - A limit on how many predictions should be returned.
- **path** - This input is required by some predictors - it is important to pass a project's path to Jedi as this will enable it to see relative imports.

```
import jedi
from .predictor import Predictor


class JediPredictor(Predictor):
    def predict(self, line, lineno, column, limit=None, path=None):
        if path is not None:
            path = os.path.abspath(os.path.split(path)[0])

        # Jedi uses 1-indexed lines and columns
        # our script uses 0-indexed lines and columns
        script = jedi.Script(line, lineno + 1, column + 1, sys_path=path)

        completions = []
        for c in script.completions():
            completions.append(c.complete)

        return completions[:limit]
```

After we implemented our predictor, we can use Python's standard library Glob tool to list files in the directory where we would like to run our evaluation. The code snippet below describes how we could run our evaluator on all Python files in the `tornado` library.

```
from lib import JediPredictor, Counter
import glob

# list .py files in the tornado directory
files = glob.glob("tornado/**/*.py")
# instantiate the counter and custom predictor
counter = Counter()
predictor = JediPredictor()

# save result per-file
saved_keystrokes = []
for file in files:
    saved_keystrokes.append(counter.count_keystrokes(file, predictor))

avg_keystrokes = sum(saved_keystrokes) / len(saved_keystrokes)

print("Our predictor saved", 1 - avg_keystrokes, "keystrokes")
```

Please note that this is a very simple use of our Evaluator which only counts the average % keystrokes saved. We could also return the keystrokes saved per file and measure the position of the desired completion each time it appears like in our evaluation..

Appendix B

Sorting Jedi Predictions

Here we give the implementation of the `.predict()` function from our code completion system that sorts Jedi predictions. For conciseness, we omitted the initialization code.

```
    def predict(self, line, lineno, column, limit=None, path=None):

        if path is not None:
            path = os.path.abspath(os.path.split(path)[0])

        encoded_line = []

        if len(line) != 0:
            g = line.split(" ")
            # g[-1] is the keyword currently written, so it is incomplete
            tokenized_line = [token for token in g[:-1]]
            # remove infrequent words (according to the train set)
            filtered_line = list(
                filter(lambda x: x in self.vocab, tokenized_line))
            if len(filtered_line) == 0:
                # zero-pad the sequence to length 40
                encoded_line = np.array([np.zeros(len(self.encodings)) for
                    _ in range(40)])
            else:
                # one-hot encode and pad line
                for item in filtered_line:
                    temp = np.zeros(len(self.encodings))
                    temp[self.encodings[item]] = 1
                    encoded_line.append(temp)

                encoded_line = keras.preprocessing.sequence.pad_sequences(
                    [encoded_line], maxlen=40)
        else:
            encoded_line = [np.zeros(len(self.encodings)) for _ in range(40)]
        # query Jedi
        script = jedi.Script(line, lineno + 1, column + 1, sys_path=path)

        oov_completions = []
```

```
# one-hot encode predictions that are in the vocabulary
enc_completions = {}
for c in script.completions():
    if c.name in self.vocab:
        tmp = np.zeros(len(self.encodings))
        tmp[self.encodings[c.name]] = 1
        enc_completions[c.name] = (c.complete, tmp)
    else:
        oov_completions.append(c.complete)

prediction_scores = []
# feed values in enc_completions to model and get their probabilities
model_input_seq = encoded_line.reshape(1, 40, 1991)

for e in enc_completions:
    complete, encoding = enc_completions[e]
    model_input_compl = [encoding]
    score = self.model.predict(
        [np.array(model_input_seq), np.array(model_input_compl)])

    prediction_scores.append((complete, score[0][0]))

# order on probabilities returned from model
prediction_scores = sorted(
    prediction_scores, key=lambda x: x[1], reverse=True)
# print(line, prediction_scores)

preds = [pred for (pred, score) in prediction_scores]
preds = preds + oov_completions


if limit is None:
    return preds
else:
    return preds[:limit]
```

Appendix C

Model Hyperparameters

In this appendix we list the hyperparameters of our best-performing model for sorting Jedi predictions.

Parameter	Value
LSTM Units	256
LSTM Activation	sigmoid
Dense layer units	256
Dense layer activation	ReLU
Dropout regularization	0.2
Last layer activation	sigmoid
Input sequence length	40 tokens
Minimal token count	5
One-hot encoding size	1991

Here we also give the hyper-parameters used for fine-tuning the GPT-2 model that generated sample text for Chapter 3.

Parameter	Value
dataset	CodeSearchNet
Training steps	1000
Batch size	1
Learning rate	0.0001
Model size	355M

Bibliography

[1] Autocompletion with deep learning — tabnine. https://tabnine.com/blog/deep/. (Accessed on 01/19/2020). pages 17, 21

[2] borisbarath/code-completion-gym: A toolkit for evaluating code completion systems on real code. https://github.com/borisbarath/code-completion-gym. (Accessed on 06/07/2020). pages 4

[3] davidhalter/parso: A python parser. https://github.com/davidhalter/parso. (Accessed on 05/31/2020). pages 40

[4] Deepnote - data science notebook for teams. https://deepnote.com/. (Accessed on 06/07/2020). pages 2

[5] Github - davidhalter/jedi: Awesome autocompletion and static analysis library for python. https://github.com/davidhalter/jedi. (Accessed on 01/19/2020). pages 17, 23

[6] Github - github/codesearchnet: Datasets, tools, and benchmarks for representation learning of code. https://github.com/github/CodeSearchNet. (Accessed on 05/05/2020). pages iv, 3, 19, 33

[7] The import system — python 3.8.3 documentation. https://docs.python.org/3/reference/import.html. (Accessed on 05/31/2020). pages 33

[8] Intellisense in visual studio code. https://code.visualstudio.com/docs/editor/intellisense. (Accessed on 01/19/2020). pages 17

[9] Langserver.org. https://langserver.org/. (Accessed on 05/22/2020). pages 28

[10] microsoft/python-language-server: Microsoft language server for python. https://github.com/Microsoft/python-language-server. (Accessed on 05/22/2020). pages 22

[11] minimaxir/gpt-2-simple: Python package to easily retrain openai's gpt-2 text-generating model on new texts. https://github.com/minimaxir/gpt-2-simple. (Accessed on 05/23/2020). pages 19

[12] Official page for language server protocol. https://microsoft.github.io/language-server-protocol/. (Accessed on 05/22/2020). pages iv, 1, 22, 28

[13] openai/gym: A toolkit for developing and comparing reinforcement learning algorithms. https://github.com/openai/gym. (Accessed on 06/08/2020). pages 37

[14] palantir/python-language-server: An implementation of the language server protocol for python. https://github.com/palantir/python-language-server. (Accessed on 06/12/2020). pages 29

[15] Papers with code : Language modelling. https://paperswithcode.com/task/language-modelling. (Accessed on 01/19/2020). pages 1, 17, 18

[16] py_compile — compile python source files — python 3.8.3 documentation. https://docs.python.org/3/library/py_compile.html. (Accessed on 05/20/2020). pages 24

[17] tornadoweb/tornado: Tornado is a python web framework and asynchronous networking library, originally developed at friendfeed. https://github.com/tornadoweb/tornado. (Accessed on 05/20/2020). pages iv, 33

[18] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. pages 23

[19] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–15, 2015. pages 14

[20] Richard Bellman. *Dynamic Programming*. Dover Publications, 1957. pages 7

[21] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. *Advances in Neural Information Processing Systems*, 3:1137–1155, 2001. pages 8

[22] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016. pages 4, 37

[23] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pages 1724–1734, 2014. pages 14

[24] François Chollet et al. Keras. https://keras.io, 2015. pages 23

[25] Subhasis Das. Contextual code completion using machine learning. 2015. pages 40

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR*, abs/1810.04805(Mlm), 2018. pages 15

[27] Eclipse-Foundation. *Eclipse IDE*, 2020 (accessed January 16, 2020). https://www.eclipse.org/ide/. pages 7

[28] Kawin Ethayarajh. How contextual are contextualized word representations? comparing the geometry of bert, elmo, and gpt-2 embeddings. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *EMNLP/IJCNLP (1)*, pages 55–65. Association for Computational Linguistics, 2019. pages 15

[29] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. CACHECA: A Cache Language Model Based Code Suggestion Tool. *Proceedings - International Conference on Software Engineering*, 2:705–708, 2015. pages 7

[30] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000. pages 13

[31] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997. pages 12, 13

[32] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956. ACM, 2019. pages 27

[33] Daniel Jurafsky and James H. Martin. N-Gram Language Models N-Gram Language Models, available at https://web.stanford.edu/~jurafsky/slp3/3.pdf. 2009. pages 5, 6

[34] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *CoRR*, abs/1404.2188, 2014. pages 9

[35] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics. pages 9

[36] Ryan Kiros, Ruslan Salakhutdinov, and Richard Zemel. Multimodal neural language models. *31st International Conference on Machine Learning, ICML 2014*, 3:2012–2025, 2014. pages 7

[37] R. Kneser and H. Ney. Improved backing-off for m-gram language modeling. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 181–184 vol.1, May 1995. pages 6

[38] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, Contour and Grouping in Computer Vision*, page 319, Berlin, Heidelberg, 1999. Springer-Verlag. pages 9

[39] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. *IJCAI International Joint Conference on Artificial Intelligence*, 2018-July:4159–4165, 2018. pages 1, 16, 17

[40] Ping Yeap Loh, Wen Liang Yeoh, Hiroki Nakashima, and Satoshi Muraki. Impact of keyboard typing on the morphological changes of the median nerve. *Journal of occupational health*, pages 17–0058, 2017. pages 35

[41] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and Optimizing LSTM Language Models. *CoRR*, abs/1708.02182, 2017. pages 13

[42] Microsoft. Documentation for visual studio code. https://code.visualstudio.com/docs. (Accessed on 06/12/2020). pages 22, 28

[43] T. Mikolov, S. Kombrink, L. Burget, J. Černocký, and S. Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5528–5531, May 2011. pages 1, 11

[44] T. Mikolov and G. Zweig. Context dependent recurrent neural network language model. In *2012 IEEE Spoken Language Technology Workshop (SLT)*, pages 234–239, Dec 2012. pages 1, 11, 12

[45] Tomáš Mikolov, Anoop Deoras, Daniel Povey, Lukáš Burget, and Jan Černocký. Strategies for training large scale neural network language models. *2011 IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU 2011, Proceedings*, pages 196–201, 2011. pages 1, 6, 7

[46] Tomaš Mikolov, Martin Karafiát, Lukaš Burget, Cernocky Jan, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association, INTERSPEECH 2010*, number September, pages 1045–1048, 2010. pages 11

[47] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *ArXiv e-prints*, 11 2015. pages 9

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. pages 23

[49] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep Contextualized Word Representations. *CoRR*, abs/1802.05365:2227–2237, 2018. pages 15

[50] Ngoc Quan Pham, German Kruszewski, and Gemma Boleda. Convolutional neural network language models. *EMNLP 2016 - Conference on Empirical Methods in Natural Language Processing, Proceedings*, pages 1153–1162, 2016. pages 1, 9, 10, 11, 12

[51] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019. pages 3, 16, 19

[52] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. *ACM SIGPLAN Notices*, 49(6):419–428, 2014. pages 6, 7

[53] Ronald Rosenfeld. Two Decades of Statistical Language Modeling. *Proceedings of the IEEE*, 88(8):1270–1278, 2000. pages 5

[54] Holger Schwenk. Continuous space language models. *Computer Speech and Language*, 21(3):492–518, 2007. pages 1, 9

[55] Holger Schwenk and Jean-luc Gauvain. Training Neural Network Language Models. *Proceedings of the Conference on Empirical Methods in Natural Language Processing: Human Language Technologies (EMNLP/HLT'05)*, pages 201–208, 2005. pages 8

[56] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314, 2018. pages 10

[57] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *CoRR*, abs/1505.00387, 2015. pages 10

[58] Martin Sundermeyer, Hermann Ney, and Ralf Schluter. From feedforward to recurrent LSTM neural networks for language modeling. *IEEE Transactions on Audio, Speech and Language Processing*, 23(3):517–529, 2015. pages 1, 13

[59] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language processing. *Interspeech 2012*, pages 194–197, 2012. pages 13

[60] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems*, 4(January):3104–3112, 2014. pages 1, 13, 14

[61] Jalaj Thanaki. *Python Natural Language Processing: Advanced Machine Learning and Deep Learning Techniques for Natural Language Processing*. Packt Publishing, 2017. pages 25

[62] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 16-21-November-2014:269–280, 2014. pages 7, 33

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017-Decem(Nips):5999–6009, 2017. pages 1, 14, 15

[64] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer Networks. In C Cortes, N D Lawrence, D D Lee, M Sugiyama, and R Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015. pages 1, 16

[65] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing [Review Article]. *IEEE Computational Intelligence Magazine*, 13(3):55–75, 2018. pages 1, 8, 9, 10