Code Autocompletion: Research Report

University of Auckland Compsci 760, Semester 2 2020

Sean Zeng | Sahil Bahl | Vishal Thamizharasan | Josh Atwal

ABSTRACT

Code prediction is a subset of natural language processing that has attracted recent popularity for its use in assisting developers and streamlining program development. Unlike natural languages, code has an explicit and hierarchical structure that can be exploited for better results, making relying solely on traditional statistical-based approaches in natural language processing unsuitable. In this report we explore the research question of *how can recent deep-learning models be applied to the task of code prediction while incorporating structural and syntactic information from the code*. We address this by incrementally developing a recurrent neural network variant and tuning it to our large dataset of python source code files.

1 INTRODUCTION

Machine learning has recently been successfully applied to developer productivity tools and, in particular, code prediction [5]. Code prediction is a common feature in modern IDEs for an autocompletion functionality, whereby the current position of the cursor and the relevant source code is used to generate a list of likely next tokens. This has the benefit of saving time while reducing human error and the possibility of bugs. It also becomes an effective learning mechanism: a developer that cannot remember what should come next or that is unfamiliar with a coding language or library can receive meaningful suggestions rather than an alphabetically ordered list as was common in older IDEs [13].

Even gathering a list of type-compatible tokens becomes a challenge in a dynamically-typed language, but machine learning has proven to currently be the best method of learning the statistical properties of the code, and exploiting the "naturalness hypothesis": that software is just another form of human communication and software corpora should have similar statistical properties to natural language corpora [5]. Any good model should take into account the features of the data, and two of the major features of code [11] are *long-term dependencies* between tokens in different positions in the program, and *localisation*, whereby repeated patterns encountered in one source code file may differ from those in other files. Whereas many past approaches were unable to do so, the success of a model for code prediction is contingent on it being able to handle large context sizes while also being able to generalise well.

The recent popularity of deep learning in a range of applications (particularly image, sound, and text analysis) has spawned Recurrent Neural Networks (RNNs). These are specialised sequence processing models able to take a linear sequence of code and learn a language model through a complex process that allows them to better capture context and long-range dependencies than traditional methods. Long Short-Term Memory networks (LSTMs) are a variation of RNN models that introduce extra complexity in the way they calculate their hidden state using gates that allow models to combine information from past time-steps and the current time-step. Both RNNs and their variant LSTM models have been shown to be a viable option for language modelling [11].

Given the hierarchical nature of code, it makes less sense to treat the code as a linear series of tokens than in traditional natural language applications [15]. Sequence-processing models like RNNs can be improved upon by feeding sequences that retain some of the structured and hierarchical information from the code. *Abstract Syntax Trees* (ASTs) are tree representation of the source code generated from statements and expressions in the programming language, and linearly traversing ASTs in some pre-defined way allows them to be used as the input for a sequence-processing model. An example of the Abstract Syntax Tree representation for code can be seen in Figure 1.

Given the failures of previous approaches to effectively handle the nuances of code prediction and the novelty of deep-learning models such as RNNs within this application domain, we are now able to clearly define our research question: ***how can recent deep-learning models be applied to the task of code prediction while incorporating structural and syntactic information from the code?***

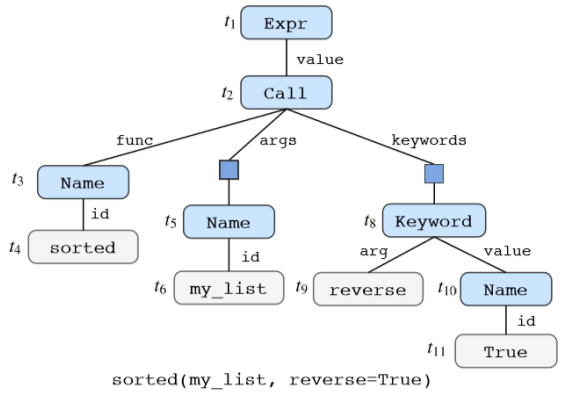
2 RELATED WORK

In this section we cover other work related to our paper that we reviewed.

2.1 RNN based approaches

White et al.[20] was the first to empirically demonstrate that a basic RNN model was able to outperform the n-gram model (upto 9 n-gram models) due to their increased ability to learn long-range dependencies in source code. More recently, Liu et al. [15] were able to achieve a high-standard of result by augmenting an LSTM network with a memory stack. While traversing the code, the hidden state can be PUSHed or POPped on/off the stack at each level of nesting (for example when entering and exiting an if statement block)–capturing within the stack memory the hierarchical structure and long-range dependencies between different positions in the code. However, in general it is difficult for RNNs to deal with very long-range dependencies, which are common in source code such as the class definition declared many lines before it is used. Using attention mechanisms, neural language models are able to learn to retrieve and make use of relevant previous hidden states, thereby increasing the model's memorization capability.

Bhoopchand et al. [7] state that neural models have fixed internal memory dimensions and can often only capture local phenomena (so-called hidden state bottleneck), e.g. they can fail to capture context information for functions declared at the start of a large file and are only referenced sparsely at the end, even though many



t	Frontier Field	Action
t_1	stmt root	Expr(expr value)
t_2	expr value	Call(expr func, expr* args, keyword* keywords)
t_3	expr func	Name(identifier id)
t_4	identifier id	GenToken[*sorted*]
t_5	expr* args	Name(identifier id)
t_6	identifier id	GenToken[*my_list*]
t_7	expr* args	Reduce (close the frontier field)
t_8	keyword* keywords	keyword(identifier arg, expr value)
t_9	identifier arg	GenToken[*reverse*]
t_{10}	expr value	Name(identifier id)
t_{11}	identifier id	GenToken[*True*]
t_{12}	keyword* keywords	Reduce (close the frontier field)

Figure 1: Abstract Syntax Description Language AST representation (*Left*) of code (*Right*) from StructVAE Yin et al. [21]

of the neural methods take in an entire code file as one training sample. They have instead shown that their sparse attention model is able to capture these longer dependencies.

Li et al. [14] incorporated the idea of attention into traditional LSTMs by adding context attention: additional hidden states added to remember context around given code block. Besides the traditional context attention, *parent attention* is also added for the AST-based code completion. In an AST, the parent node should have relevance to the child node. But when flattening the AST the parent-children information is lost. To keep this structure information, when flattening the AST, the parent location of each AST node is recorded and fed into the network.

2.2 Transformer based approaches

LSTM-based approaches have dominated the field of code prediction until recently, when state-of-the-art methods have tended to involve transformer networks which focus solely on attention mechanisms. Facebook's Kim et al. [13] achieved significant increases in accuracy over more conventional methods by exposing transformer models to the syntactic structure of the code: exploring path decompositions and traversals of the ASTs in the py150 dataset [16], with their best performing model incorporating a depth-first-search approach. This model was shown to be robust on a separate (Facebook internal) dataset, and an attribution study confirmed that the model focuses on relevant code locations for prediction. For future work the authors of this paper suggest copying mechanism and open-vocabulary models as ways of handling words that lie outside of the defined vocabulary (although the vocabulary size of 100k words already covered 90% of the dataset) as well as predicting beyond the next token and extending the model to languages other than Python.

Microsoft's Intellicode model [17] is a modified version of a GPT-2 transformer model that is trained from scratch on source code data. Like the GPT-2 and the transformer-decoder model from Peter J. Liu et al. [15], they did away with the encoder block so to as reduce the number of model parameters and to avoid redundancy like re-learning the same information about the language in both the encoder and decoder blocks. Their model learns representations for sub-tokens or combinations of Unicode characters instead of a whole token, thus allowing the model to tackle out-of-vocabulary words. With a simple modification of the training sample they are able to effectively train a multilingual model that gives a comparable performance to a monolingual model.

Further examples in practice of transformer models being used for code auto completion include TabNine [1] and Galois [2].

3 METHODOLOGY

Our experimental methodology was one of starting simple and establishing a baseline, before incrementally adding complexity to our models and approach.

3.1 Dataset

The dataset we worked with for the majority of the project was the Py150 dataset [16]. This is a large dataset of 150,000 python files collected from open-source repositories on GitHub, available both as the original source code files and as ASTs. The dataset has a pre-defined training/test split and keeps files from the same project together rather than splitting them across the two sets. The authors restrict the dataset to only containing files that result in ASTs with less than 30,000 nodes, as trees larger than that often involve JSON objects. Apart from this there is no other filtering applied to the dataset, and it reflects the distribution of code openly available on GitHub.

We performed some dataset analysis to try and understand the distribution of files for different coding purposes and styles in the dataset. In particular, we suspected that a mix of coding styles such as object-oriented versus inline code, or class-based versus function-based code may affect training accuracy. Code style is unique between developers and is not easily categorised, so we elected to pick these styles as the code differed the most between them. Class based modules references the special variable "*self*", while function based modules do not, inline code is not object oriented, while modular code is.

3.1.1 ***Classifying Code***. *Inline code* is defined as code that is run immediately by the python interpreter, this style of coding is often used by beginners, data analysts, and scripts. It is code that is not

usually inherited, and usually is "hacky" in that it is not object oriented.

We used the python token module to tokenise each code file, we then ran a filter that looked for inline code, specifically, root code lines that are not representations of "code executions". We looked for: comments, imports, variable and function declarations, function decorators ('@'), class declarations, and inline template HTML. This list was chosen after manually running an analysis of all root code lines and determining which lines are representations of "code executions" or "not correlated" (such as comments and imports). Any files that did not fall into the above set was marked as "inline".

On the other hand, *Modular code* is defined as code that needs to be inherited by other code to be run, for example the 'os' module is inherited and called 'os.listdir()' in order to be run. This style of coding is often used in team environments, open source projects, and generally, by more experienced developers. Modular code was simply defined as whatever was not inline.

We observed a distribution of 54% and 60.3% of modular code in the test set and training set respectively.

3.1.2 ***Further splitting the sets.*** Code written in python classes differ significantly from non-class code, specifically, python classes reference special variables such as "self" before the execution of sibling functions or access of sibling variables. It was important to ensure that the test and training sets had a significant representation of class code, we went further and looked at the distributions in both inline and modular code types. We looked for root class definitions and root function definitions within files and categorised them into 4 types:

- Class Only - contains root 'class' declarations, but not root 'def' declarations
- Function Only - contains root 'def' declarations, but not root 'class' declarations
- Both - contains both root 'def' declarations and root 'class' declarations
- None - contains no root declarations of either 'class' or 'def'

We analysed the dataset on these terms and found the different distributions to be balanced and similar between our training and test sets. We also looked at the file imports in the dataset and were able to ascertain popular libraries, and scrape the topic tag from GitHub to infer the purpose of each repository. We found that many of the repositories were related to web frameworks, but as mentioned, this is likely reflective of GitHub as a codebase overall. For more detailed results from this analysis, see Dataset Analysis under the Results section.

Overall, we feel confident in using this dataset given that the paper [16] introducing it has 78 citations at the time of writing and the authors are affiliated with ETH Zurich.

Given the size of the dataset we were working with and the computational resources required to train RNNs, we used a subset of the full dataset. Even with a reduced dataset, there was a necessity of cloud computing and so we set up an Azure instance. This allowed us to easily work collaboratively by accessing the same virtual machine that was able to train the models we investigated usually within 10 hours.

3.2 Character Based Model

We began by implementing an character based LSTM model based on a GitHub repository [3].

3.2.1 ***Vocabulary.*** To train a model, the first task is to convert a given code snippet into a list of numerical vectors. To enable this we need to define a vocabulary for the given model.

The input vocabulary for the model is mainly character based. However, since we are modelling code instead of plain English, we can use special words and characters holding a specific meaning by the python programming language i.e. keywords, operators, special characters etc.

The final vocabulary was a combination of 10 types of tokens:

(1) **Eof** : Denotes end of file

(2) **Indent** : Whenever indentation increases

(3) **Dedent** : Whenever indentation decreases

(4) **NewLine** Whenever there is change of line

(5) **Operator** : Various operators identified by the python programming language.

Examples: +, -, =, ==, :

(6) **KeyWord** : All the reserved keywords defined in python.

Examples: for, while, continue, break

(7) **Name** : List of all the lower and upper case characters.

Examples: a-z, A-Z

(8) **Number** : List of all the numeric characters. Example: 0 - 9

(9) **String** : This denotes all the text enclosed by quotes.

Examples: print("This is a string token type") In this case all the text within the print is treated as a single token of type String

(10) **Comment** : This is to identify comments in the code and these are removed as part of the pre processing step.

Examples: In python is used for single line comments and """ is used for multi line comments

This resulted in a fixed vocabulary of size **186 tokens** which can be used to encode every character in the source code.

3.2.2 ***Pre-processing.*** For the model input, source code is first converted into list of character tokens.

A custom parser is used to convert source code into the token list. Code is parsed at a character level and then characters are combined based on the token types to output a list of token types. This list is converted into integers by using the index of the token in the vocabulary. Below is an example of how the parser works.

Example input code:

```
import numpy as np
```

Parser Output: Will be a sequnce of intergers based on the table below [5, 78, 85, 77, 80, 89, 8, 78, 80]

Token	Token Type	Token integer ID
import	Keyword	5
n	Name	78
u	Name	85
m	Name	77
p	Name	80
y	Name	89
as	Keyword	8
n	Name	78
p	Name	80

3.2.3 ***Training - Model Input and Output.*** The model input is the previous token and the model tries to predict the next sequence based on the previous token. Each file in the training set is converted into an integer vector and then this vector is shifted by 1 to define a list training sequence where the first number is the input token and the second number is the output token. For the previous example:

Input Code: import numpy as np

Vector Sequence: [5, 78, 85, 77, 80, 89, 8, 78, 80]

Training Sequences: [[5, 78], [78, 85], [85, 77], [77, 80], [80, 89], [89, 8], [8, 78], [78, 80]]

1,157,890 input sequences are used as the training set and 204,372 input sequences are used as validation set.

3.2.4 ***Model Architecture.*** We trained a LSTM model with the following hyper parameters and architecture: we used 3 hidden layers, with 1024 hidden units per layer. Training was done over 100 epochs using a batch size of 32 using the cross-entropy loss function and Adam optimiser. These and the rest of the hyperparameters were taken from the original repository, though we did briefly experiment on this model by adding additional layers, adjusting hyper-parameters, and increasing the size of the data set.

3.2.5 ***Testing - Model Input and Output.*** When making predictions on the test file instead of just doing a prediction on the next character, we implemented a beam search of length 8 to be able to predict multiple tokens. The evaluation metric for this model was percentage of key strokes saved. The accuracy on the test set was **38.62**%.

3.2.6 ***Model Limitations.*** We identified various limitations in the character based model.

- The model was character based instead of word based requiring prediction to involve a performance intensive beam search
- The vocabulary was fixed to 186 tokens and could not be increased.
- Model only considered the previous token for making the prediction.
- There was limited exploitation of the structure of the code and user-defined constructs

3.3 Word-token model

Our first major improvement to the model was to, instead of the character based input from the previous model. use whole-word tokens for model input. We also addressed the fixed vocabulary size and the fact that the previous model just used the previous token to make predictions.

3.3.1 ***Vocabulary.*** While using character tokens meant that arbitrary strings could be represented in the input, switching to whole word tokens meant that a vocabulary had to be defined. We defined our vocabulary as the top 1000 tokens from the training data, and considered any other token as a special unknown token character. While a vocabulary size of 1000 may be small compared to other papers, there was a soft limit on how large a vocabulary we could have, given our computational resources. We still found that a vocab size of 1000 was sufficient to cover 82.6% of the tokens in the training set.

3.3.2 ***Pre-processing.*** For the model input, raw source code is first converted into the Abstract Syntax Tree using the python ast module. We trained 2 different models, one with the input sequence being text tokens while for the other model we used the AST nodes as input. AST nodes were converted into raw text nodes by using an AST unparser. This text token list is then converted into integers by using the index of the token in the top 1000 token vocabulary. All tokens not known in the vocab are unknown tokens.Another special token added to the vocab was the padded token. This is used when making predictions to pad the input to the model to ensure the same input length. So the final vocab size was 1002.

Example input code:
```
import numpy as np
```

Parser Output:

Assuming vocab has import, numpy and as on the first, second and third position this will be converted into the following sequence: [1, 2, 3, 1001]

3.3.3 ***Training - Model Input and Output.*** The second improvement over the character based model was to increase the context length: allowing the model to consider the last 1000 tokens when predicting the next token. This improvement allowed the model to better understand the semantics of the code. This model sequences are defined on a file-by-file basis and so the position or line number of tokens (such as having `import` statements be at the start) is implicit in the input sequence. We began with 10K files for training and validation in a 70:30 split. This results in 27k training sequences and 12k validation sequences. The model was trained for 10.6 hours on NC-series GPUs on Azure.

For the previous example, if we used lookback window size of length 2, thirst first two numbers denote the input and the last number denotes the output for each training sequence.

Input Code: import numpy as np

Vector Sequence: [1 , 2, 3, 1001]

Training Sequences: [[1, 2, 3], [2, 3, 1001]]

For our model we used lookback window size of length 1000, so the model input length was 1000 and the output was the next token.

3.3.4 ***Model Architecture.*** We trained a LSTM model with the following hyper parameters and architecture: We used 2 LSTM layers followed by 2 fully connected layers, with 50 hidden units per layer. Training was done over 20 epochs using the cross-entropy loss function and Adam optimiser. We used the window size of 1000. These and the rest of the hyperparameters were taken from [13] and their LSTM implementation for the same task of word token based code prediction. We experimented on this model by trying different vocab sizes and window sizes.

3.3.5 ***Testing - Model Input and Output.*** The model was tested on the tokenised version of the test data code files. The first token is padded to the window length and is used as the input sequence for the model to predict the next token. The comparison of this next token with the ground truth defines our accuracy metric. We continue by adding the next token to the input sequence and continue till the second last token in the code file.

The test set consists of 50 files, resulting in 53000 sequences. The discrepancy between the number of input sequences per file for the train set and the test set is because while training, we did not pad the input sequences and so training starts from the token at index 1000, while predictions start from the token at index 1. This is because we want to make a prediction after every new word typed. The evaluation metric for this model was percentage of correct next token predictions. In terms of accuracy we looked at the top 1 and top 5 accuracy. For top 1 accuracy we consider the prediction a success if the highest probability suggestion returned by the model matches the ground truth, while for top 5 if any of the top 5 suggestions match the ground truth the suggestion is considered a success.

The top 5 accuracy for the model was **0.720** and the top 1 accuracy for the model was **0.443**.

3.3.6 ***Model Limitations.*** There were a few limitations to this word based model:

- Tokenisation does not explicitly include end of line or change of indentation tokens
- The Look back window size of 1000 may be too large for some cases, since the code is block-based
- The vocabulary size could be larger (top 1000 tokens cover 82.6% of tokens in the training source code)

3.4 Final model

For our final iteration, we focused on two areas to improve our previous model: explicitly including indentation and end-of-line tokens, and optimising the length of our lookback window. Including indent, dedent, and new-line tokens increased the accuracy of the model noticeably as expected, as indentation is fundamental to syntax in Python. This is important for cases where the meaning of the code is ambiguous without knowing the position of the indents and dedents. For example, consider the following code snippet:

```
for i in range(n):       # Outer loop
    for j in range(m):   # Inner loop
        print(j)
    print(i)
```

Without an explicit character denoting the change in indentation, parsing this as a sequence does not provide the necessary information to tell whether the final `print(i)` statement is inside or outside of the inner loop.

To add new line tokens, we modified our AST unparser code to insert a `<newline>` token at the point in the sequence where the line number changed. Since we directly used line numbers to determine change of line defined on the AST nodes, we were able to handle files with different new-line characters. Indents and dedent tokens were added whenever there was a change in indentation level.

Varying the window size of context given to the model also yielded improvements in accuracy. Using a fixed vocabulary size of 1000, we experimented with window sizes of 25, 50, 100, 250, and found that the previous window size of 1000 was much larger than needed. The window size of 100 gave the best result. The full result of this experiment is available under Next Token Prediction in the Results section.

To summarise, our final model is:

- An LSTM network with 2 LSTM layers followed by 2 fully connected layers, with 50 hidden units per layer. We used the model hyper parameters as the ones defined in the Facebook's Kim et al. [13] paper.
- Word token based, including indentation and new-line tokens
- Has a context window length of 100
- Uses vocab size of top 1000 tokens

4 RESULTS

In this section we present the results of our dataset analysis, and our models for next-token prediction.

4.1 Dataset Analysis

4.1.1 ***Code Style Distribution.*** The goal of our dataset analysis was to ensure a balanced distribution of coding styles across the training and the test sets. For example, a mix of coding styles such as object-oriented versus inline code, or class-based versus function-based code may affect the accuracy and generalisability of our approach. We analysed the dataset in these terms and found a balanced distribution of in-line versus modular coding formats. We then split the inline and modular sets further into class only, function only, both, and other, respectively and found balanced distributions in both the test and train sets. These results can be seen in Table 1 This reassures us that there are no major class imbalances in the dataset, and that while there may be a preference for object-oriented code, it is unlikely to be skewing the results of our models by too much.

As our analysis only considers the dataset on a file by file basis, it is possible for some files to have more lines of code - therefore have more representation than other types of code styles. Analysis

	Inline	Modular	Class Only	Function Only	Both	None
Training Set Inline	100%	0%	22.6%	36.9%	19.6%	21.0%
Training Set Modular	0%	100%	51.6%	21.2%	13.7%	13.4%
Test Set Inline	100%	0%	47.8%	30.4%	17.4%	4.3%
Test Set Modular	0%	100%	48.1%	22.2%	22.2%	7.4%
Training Set	39.7%	60.3%	40.1%	27.4%	16.0%	16.4%
Test Set	46.0%	54.0%	48.0%	26.0%	20.0%	6.0%

Table 1: Dataset Analysis

of the code line count distribution for both test and training set, yields a normal distribution of code line counts. We therefore feel comfortable that our file by file analysis is still a good approximation in representing the different coding styles in our dataset. We did encounter an outlier file containing over 20K lines of JSON string. However, as our model tokenisation converts the JSON strings into unknown tokens, there should be no major negative affect on training.

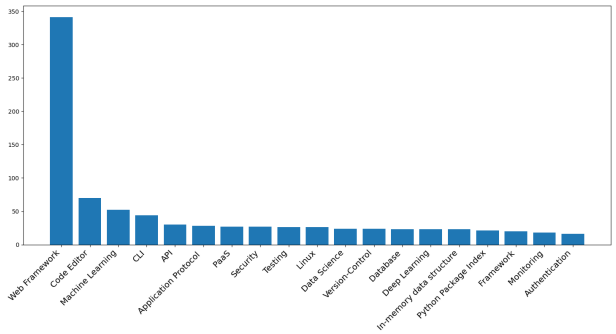


Figure 2: Code topic distribution

4.1.2 ***File purpose***. We also briefly looked at the import statements in each file to determine the popular libraries. We found that libraries like `os` and `sys` were by far the most common, followed by other libraries like `logging`, `re` (regular expressions), `time`, `unittest`, and `numpy`. By scraping the topic tag on GitHub, we also were able to infer the purpose of each repository. The largest category was Web Frameworks by far. Other popular categories included Machine Learning, CLI, API, PaaS (Platform as a Service), Security, and Testing. While the style of code may be relatively evenly distributed, there is a definite skew in the purpose of the code itself. We therefore keep in mind as we interpret the results of our models that a significant amount of the code we have trained on has been to do with web frameworks.The distribution of libraries used both in the training and test dataset are almost the same.

4.2 Next token prediction

Table 2 shows the main results of our approach using a standard vocab size of 1000. Without the indent and new line characters and using a window length of 1000, the accuracy of the model in recommending the correct next token within the top 5 predictions is 72%. Including the indentation and new line characters increases this to 74.3%. However, after varying the window length, we found the best results came with a window length of 100. Our initial intuition was that the bigger the lookback window, the more context the model has. But based on the experiments we found that that longer context makes it harder for the model to learn the relationship with previous tokens.The accuracy increases initially as we increase the window size from 25 to 100 and then starts to fall when increased further. Decreasing the window size and the shorter sequences that arise have the additional benefit of a decreased number of tunable parameters to train as well.

4.3 Vocabulary Size Analysis

Table 3 demonstrates that the smaller the vocab size, the better the accuracy in both the top 1 and 5 results. When the ground truth is an unknown token, the model cannot predict anything meaningful and so this is not counted towards the accuracy. Obviously, when the model has less options to pick a prediction from, it becomes easier to choose the right one. This accuracy metric does not take into account real-world applicability however, as even a highly accurate model with a small vocabulary size will have less opportunities to offer a prediction. Considering another accuracy metric like keystrokes saved may help in this instance, although there is the consideration that this can be biased towards longer tokens.

Table 4 shows the training and test set coverage for each level of vocabulary size. One can expect the frequency of any kind of counts in a large enough setting to follow a Zipf distribution, and we see that increasing the vocabulary size leads to quickly diminishing returns. This can also be used to interpret the results in Table 3. For example, there seems to be a quick drop-off in accuracy between vocab size 125 and 250, and this is echoed by a sharp increase in dataset coverage. The first 125 vocabularies have a larger presence in the training dataset than the next 125 vocabularies, and due to being trained on more number of samples, it has a higher accuracy. For the 250 vocabulary example, the bottom 125 having a lower frequency count in the dataset and therefore brings down the overall accuracy when compared to the first 125 vocabulary example. There does seem to be a relationship between vocab size and top 5 accuracy that is approximately linear in $\log_2$–at least for this range of vocab sizes. If anything, these results show that one needs to take care in managing the trade-off between model generalisability and accuracy caused by varying the vocabulary size.

5 DISCUSSION

In this section we discuss problems and shortcomings with the project, and describe future steps with this project.

Indents and New lines	Lookback Window	Top 5 Accuracy	Top 1 Accuracy
×	1000	0.720	0.443
✓	1000	0.743	0.478
✓	25	0.774	0.484
✓	50	0.780	0.487
✓	100	0.811	0.532
✓	250	0.731	0.448

Table 2: Model Results for vocab size 1000 for 53,328 test sequences

Vocab Size	Top 5 Accuracy	Top 1 Accuracy
125	0.893	0.535
250	0.852	0.514
500	0.840	0.540
1000	0.798	0.51
2000	0.770	0.493

Table 3: Vocab size accuracy comparison, lookback Window = 100 for 43,616 test sequences

Vocab Size	Training Set
125	74.22
250	77.35
500	80.16
1000	82.64
2000	84.12
5000	87.70
10000	89.71
12500	90.34
15000	90.85

Table 4: Percentage coverage for vocab size over the training and test data

5.1 Challenges

Overall, our aims for this project were set quite high. A lot of time was required in the initial stages of the project to learn the necessary background information regarding RNNs, LSTMs, and ASTs. More time was then spent gaining practical familiarity with PyTorch and the process of training and evaluating models. Given the amount of effort we spent on both understanding and improving it, our initial baseline model may have been overly simplistic. We already suspected that it would not perform well on the principle alone that it worked at a character level rather than a word token level–and we may have ended up making more progress had we upgraded to a word token model earlier. This also meant that implementing a transformer model was too much of an orthogonal effort that we did not have enough time or resources to investigate properly: we ended up requiring all but one of the members working on the LSTM models. The novelty of transformers also meant that there wasn't much easily adaptable open-source code available for this specific problem domain in particular.

We also found ASTs were difficult to exploit, especially in Python where variables are dynamically typed. They did not yield an instant improvement in accuracy by merely using them as an input as there is no additional information for the model to learn. Svyatkovskiy et al. [19] used enhanced ASTs as input by adding data type information into the AST by running the training code at run time and capturing the data types. We did not try this as including this information is not trivial and time consuming. Kim et al. 2020 used ASTs [13] successfully, but did so with a transformer-based approach and with a vocabulary size of 100,000 compared to our 1000. While ASTs should theoretically be better able to convey the structural information of the code, by traversing the code as a tree there are inherently long range dependencies introduced between the different branches. Transformer models excel at this due to their focus on attention mechanisms because if the model is unable to appropriately capture these dependencies then parsing the code as a linear sequence may end up being better

One of the biggest problems was that the training of our models was computationally very expensive even on a fraction of the full dataset, and took about 10 hours each run. Even using cloud computing, we lacked the proper scale of resources necessary for a project like this, and the time required for training and model inference limited our ability to iterate on our approach and run experiments for an overall very slow development cycle.

5.2 Future Work

Currently, the vocabulary size is limited to 1000, causing the model to suffer from having many out-of-vocabulary tokens. However, increasing the vocabulary size also incurs an additional computational cost, and our results demonstrated the trade-off between model generalisability and accuracy caused by the vocabulary size. Further experiments could be conducted that look at interpreting these results through different metrics such as keystrokes saved, as the accuracy alone does not take into account how often a meaningful prediction can be made by the model. We even found that by making the vocabulary size too large, we started encountering things such as strings that may not be desirable in a vocabulary. Rather than using a general vocabulary, for the best results within a particular software project, training the model to have a project specific vocabulary is recommended. This kind of transfer learning approach is seen in [1, 17], enabled by the use of transformer based models. Further experiments could also be conducted to gain a better understanding of the weaknesses and strengths of the model, such as breaking the results down by code style, or testing the sensitivity of the results with respect to particular pieces of vocabulary.

One challenge with training a model on Python code is that objects have no explicit data types. These could try to be inferred and would likely improve the accuracy–the model would likely perform better on a statically-typed language like Java. While it is rather trivial to predict multiple next tokens by using the output of the previous prediction as the input of the next for the next n tokens, better results would be yielded from modifying the architecture of the network to natively handle multiple predictions. There is a lot to investigate still regarding ASTs and better exploiting the structure of the code, such as using graph neural networks to transform the input.

Other model architectures could also be explored. In particular, recent work highlights transformer models [13, 17]. More powerful approaches like this would be better suited to more complicated tasks like a model that is able to work on multiple languages. However, transformers do have the benefit of being able to benefit from transfer learning and so with a large enough model and training corpus already established, it would be easier to work backwards from there. For example, large transformer models like BERT [9] have been trained on english corpuses with billions of words. Many NLP-based approaches are then able to take this model and fine-tune it to their particular example. The problem in this case is that there is no pre-trained model with a vocabulary based on code,

Future work could also go towards handling the presence of duplicate code blocks in code bases that are used for training and evaluation (for example, many programs may include a line like `import os`). As machine learning makes an important assumption that each of the data points need to be independent and identically distributed in a dataset, not removing duplicate code blocks from the training and evaluation sets would lead to the model over-fitting. Allamanis [4] has further stated that not removing the duplicates could lead to accuracy metrics being inflated by 100% for tasks that involve modelling code bases like code completion, classification, clone detection. During our dataset analysis we also found that many of the files were to do with web frameworks, and that should be kept in mind while using this dataset going forwards. It may be worthwhile investigating the development of a larger and more varied corpus of code.

Evaluating the accuracy metrics of the different implementations is also challenge. We encountered this when trying to compare our baseline character model to our improved word token models. Since the character based model was predicting single characters at a time, it was working on an accuracy metric of individual keystrokes saved. In comparison, our word token models were evaluated as an accuracy in recommending the correct next token within the first n suggestions. In general, some models might predict all tokens including punctuation, literals like Intellicode [17] and Karampatsis et al. [12]. Others [13, 16, 19, 20] do not. Another issue is that the implementations have different accuracy metrics like top-1 or top-5 next tokens accuracy in [6, 13, 18, 19], accuracy with respect to GT [8] and Levenshtein edit similarity [17]. The methods have also been tested on different code bases and as such it becomes a challenge to compare the performance metrics of the various code completion methods. This may motivate the need for a cross comparison study in-which models are tested on the same dataset and evaluated using the same metrics. Such a study could also include the different testing environments introduced by [10] for papers where only a static environment may have been used.

While the project focused on model development, deployment into an IDE or similar is a likely end-goal application for code prediction. Future work could also involve deploying a model like this appropriately.

6 CONCLUSION

In conclusion, we have performed a robust analysis of our dataset and incrementally developed an LSTM model for code prediction. Beginning with a simple character-based model, we improved the method of input to handle complete word tokens, included a look-back context, and tuned hyperparameters. We were overall limited by our short time-frame and computational resources, but the model serves as a robust baseline for future work and improvement and is able to offer reasonable code suggestions. We see transformers as the way forward in the field of code prediction, and they are best poised to exploit the structure of ASTs and benefit from transfer learning.

REFERENCES

[1] [n.d.]. AI Smart Compose for your Code | TabNine. https://www.tabnine.com. Accessed: 2020-09-20.

[2] [n.d.]. Galois Autocompleter. https://github.com/galois-autocompleter/galois-autocompleter. Accessed: 2020-09-20.

[3] [n.d.]. Python Autocomplete. https://github.com/vpj/python_autocomplete. Accessed: 2020-10-19.

[4] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 143–153.

[5] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[6] Gareth Ari Aye and Gail E. Kaiser. 2020. Sequence Model Design for Code Completion in the Modern IDE. arXiv:2004.05249 [cs.SE]

[7] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307* (2016).

[8] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from Examples to Improve Code Completion Systems. (2009). https://doi.org/10.1145/1595696.1595728

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[10] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code? *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017). https://dl.acm.org/doi/10.1145/3106237.3106290

[11] Truyen Tran Hoa Khanh Dam and Trang Pham. 2016. A deep language model for software code. arXiv:1608.02715, archivePrefix=arXiv, primaryClass=cs.SE

[12] Rafael-Michael Karampatsis and Charles A. Sutton. 2019. Maybe Deep Neural Networks are the Best Choice for Modeling Source Code. *CoRR* abs/1903.05734 (2019). arXiv:1903.05734 http://arxiv.org/abs/1903.05734

[13] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2020. Code Prediction by Feeding Trees to Transformers. arXiv:2003.13848 [cs.SE]

[14] Jian Li, Yue Wang, Irwin King, and Michael R. Lyu. 2017. Code Completion with Neural Attention and Pointer Networks. *CoRR* abs/1711.09573 (2017). arXiv:1711.09573 http://arxiv.org/abs/1711.09573

[15] Fang Liu, Lu Zhang, and Zhi Jin. 2020. Modeling programs hierarchically with stack-augmented LSTM. *Journal of Systems and Software* 164 (2020), 110547. https://doi.org/10.1016/j.jss.2020.110547

[16] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices* 51 (10 2016), 731–747. https://doi.org/10.1145/3022671.2984041

[17] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode Compose: Code Generation Using Transformer. *arXiv preprint arXiv:2005.08025* (2020).

[18] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Franco, and Miltiadis Allamanis. 2020. Fast and Memory-Efficient Neural Code Completion. arXiv:2004.13651 [cs.SE]

[19] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted Code Completion System. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Jul 2019). https://doi.org/10.1145/3292500.3330699

[20] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 334–345.

[21] Pengcheng Yin, Chunting Zhou, Junxian He, and Graham Neubig. 2018. StructVAE: Tree-structured Latent Variable Models for Semi-supervised Semantic Parsing. *CoRR* abs/1806.07832 (2018). arXiv:1806.07832 http://arxiv.org/abs/1806.07832