Abstract

Code autocompletion is a useful tool that helps save developers’ time. Traditional autocompletion systems require source analysis and grammar rules that vary across language domains. Neural models such as GPT-2 have shown promising results in generative NLP tasks including text generation. When coupled with an almost limitless amount of training data in the form of open-source codebases, neural models are a candidate with a lot of potential for accurate code generation and autocompletion [1]. In this paper we draw upon Lee et al’s work[2] towards text autocompletion using a sequence to sequence LSTM and transformer model to tackle the problem of code autocomplete. We demonstrate that the models can indeed be used for code autocompletion effectively and that they outperform a naive baseline n-gram model. Furthermore we show that the communication game model with a transformer encoder outperforms the sequence to sequence model when taking into account percentage of tokens kept.

1 Introduction

NOTE: we had a teammate (alex wang) drop the class during the project.

The human brain reads and interprets the world in a systematic ways. From writing a scene in a book to delivering a legal argument in a courtroom, humans are able to translate the description that they have conjured in their mind at any point in time and map that to a probability of what they will say or write in the immediate future in their brains. This process of predicting future words based on what has been said so far comes naturally to humans but can be a challenge for machine learning models to achieve.

In this paper we attempt to solve the auto-complete problem in the coding domain. Code often has predictable routes that can be understood and mimicked for future prediction tasks. As well, another benefit of creating a auto-complete model for code is that code has a smaller lexicon of symbols and words than languages such as Chinese or even English, which are intentionally verbose. One difficulty of the coding problem however is that it is rigid in its design and not up to interpretation like the English language. In this paper we apply a baseline N-gram model, as well as a sequence to sequence LSTM and transformer model to this problem.

2 Related Work

Our paper builds off the work first achieved by Lee, et al[2]. In this paper, the authors explored methodologies for autocompleting text as a communication game between a human and machine. They aimed to come up with an unsupervised method to build a better autocomplete system. A secondary goal is to explore the trade-off inherent in the communication game between efficiency, accuracy and interpretability of keywords, and then to find a good method to optimize for these combined objectives. There are many approaches to autocomplete such as left-to-right. A limitation of left-to-right autocomplete is that it might be inefficient because oftentimes the prefix of sentences don’t capture the core message.
The approach in this paper constrains the problem by using keywords extracted from the original sentence as sentence summary. The keywords, being a subsequence of original sentence, may not be grammatically correct, but are very interpretable by humans. The task of optimizing for multiple objectives – efficiency and accuracy, requires balancing communication cost (efficiency measure) and reconstruction loss (accuracy measure). The naive methods involve linearly combining the cost and loss, and searching for a coefficient that minimizes the combined loss.

$$\min_{\alpha, \beta} E[\text{cost}(x, a)] + \lambda E[\text{loss}(x, \alpha, \beta)]$$

But this is shown to be unstable and suboptimal. To solve this, Lee, et al views the accuracy-efficiency tradeoff as a sequence of constrained optimization problems where efficiency is optimized subject to accuracy constraint, using this formula:

$$\min_{\alpha, \beta} E[\text{cost}(x, a)] + \lambda E[\text{loss}(x, \alpha, \beta)] \leq \epsilon$$

The optimization can be evaluated using the Lagrangian of the equation:

$$\min_{\alpha, \beta} \max_{\lambda \geq 0} J(\alpha, \beta, \lambda) \text{ where } J(\alpha, \beta, \lambda) = E[\text{cost}(x, a)] + \lambda (E[\text{loss}(x, \alpha, \beta)] - \epsilon)$$

The result is greatly improved stability in training and good efficiency accuracy performance.

Lee, et al showed robustness by comparing the decoder’s performance based on different encoder types with different parameters. However, the robustness with respect to other datasets remain unaddressed. For example, the training/test data are on short colloquial sentences with at most 16 tokens. This might make it easier to autocomplete given a set of keywords as there are not long distance dependencies in the sentence.

Another limitation in Lee, et al’s work is that the data is domain specific to the restaurant review corpus. Restaurant review might use certain word types (nouns, adjectives) with different retention ratio more often than other word types. Difference of word type distribution of dataset compared to general language dataset could impact portability of the model.

One final limitation is the assumption is that we know the encoder tends to retain some types of words more frequently, and the decoder will autocomplete based on those keywords. But some words with low retention rate (e.g. conjunction word) could change meaning significantly.

This paper utilized existing NLP models and algorithms such as word embedding, LSTM, and a seq2seq model with attention and a copy mechanism to build its encoder-decoder network for auto completion.

In terms of how this problem connects to broader NLP research, there are 3 specific domains that are worth mentioning:

1. Optimization strategy: Many machine learning tasks have multiple objectives. For example multi-task learning is inherently multi-objective because different tasks have different goals. Oftentimes a single metric does not represent our overall optimization objective. Thus multi-objective optimization needs to find a Pareto optimum. Many methods are proposed to solve this issue, such as multi-gradient descent algorithm[3] but this method does not scale well for deep neural networks. Another method proposed by Sener Koltun [4] developed a Frank-Wolfe-based optimizer that does scales to high-dimensional problems.

2. Modelling autocomplete task as a communication game: Wittgenstein [5] once proposed a language game that teaches agent language through purposeful usage of the language. Sida Wang, Percy Liang et. al. build a game (called SHRDLURN) that teaches a language by 2 parties collaborating to accomplish a single task [6]. This paper embodies this philosophy by having encoder-decoder collaborate on producing sentence summaries that can be reconstructed back to the original sentence with high accuracy. This paper does not need human input in the loop (because keywords are used as a proxy for human interpretability), and thus can be unsupervised. This makes the algorithm more scalable.

3. General sentence compression: sentence compression is an important part of this language game because the network needs to first generate a summary of the original sentence. There are many
ways to do sentence compression, as discussed in the background section of this paper. One is generating a grammatical summary from the original sentence[7]. Another is to generate a latent vector as a representation. These approaches either require large amounts of supervised data or are not interpretable by humans. This paper used an alternative approach of summarization using keywords.

A Google paper also demonstrated the feasibility and interpretability of keywords summarization where they arrived at condensed sentence by deleting words with LSTM while preserving meaning[8].

3 Approach

There are many approaches to autocompleting code. The simplest is using macros to replace words. Current IDEs use source analysis and rule based approach for autocompletion. Alon et al proposed using abstract syntax trees to represent code snippets and autocompletes by choosing the next node to maximize probability. In our approach, we used the communication game model as a starting point and trained on code data to see performance of autocomplete. This is viable because on the first order, autocomplete based on keywords is the same problem for both natural language and code.

For our baseline, we implemented a trigram, 4-gram and 5-gram model from scratch. The trigram model used \( p(w_i|w_{i-1}, w_{i-2}) \) to predict the next word - while the 4-gram model used the words up to i-3, and 5-gram up to i-4. That is, we provided the first N-1 tokens of each line, and has the N-gram model predict the rest of the line. When we could not find the previous two tokens in our training set, we sampled randomly.

For our neural model with an LSTM encoder, we modified the communication game model to work with tokenized source code[2]. In this approach, we view the objective to be: communicate a target sequence \((x_1, x_2...x_n)\) by passing in a series of keywords \((z_1, z_2...z_m)\). The neural model has an encoder that generates sentence (code) summarization as a sequence of keywords, and a decoder that generates back the full sentence (code) based on the keywords. Both the encoder and decoder train a 300-dimensional word embedding. The encoder is a unidirectional-LSTM with a linear layer to calculate keep probability used for sentence summarizing. The decoder phase has an encoder with bidirectional-LSTM that transforms sequences of keywords into context, which the decoder (300-dim unidirectional-LSTM) uses to generate full sentences (or in our case, code). The decoder also uses a copy generator which copies input [9] and global attention[10]. For this model, we consider both a communication and reconstruction loss where the communication loss is \( cost(x, \alpha) = E_{q_\alpha(z|x)}[\text{tokens}(z)] \) and the reconstruction loss is \( cost(x, \alpha, \beta) = E_{q_\alpha(z|x)}[-\log p_\beta(x|z)] \) which creates a trade off between efficiency and accuracy. Our objective is to minimize expected cost subject to expected loss being less than some \( \epsilon \).

For our neural model with a transformer, we further modified the communication game model. Instead of having a bidirectional LSTM for the encoder, we implemented a transformer’s encoder and encoder layers as described in [11]. More specifically, instead of an LSTM layer, our encoder now consisted 6 stacked encoder layers, where each encoder layer consisted of multthead attention and a feed-forward layer, followed by a residual connection and a layer norm, as shown in the appendix with the transformer figure.

4 Experiments

4.1 Data

We are using a subset of the dataset that the code2seq paper used[12]. Specifically, we are using the “java-small” dataset consisting of over 96k source files totalling 160MB in size. These source files came from a variety of open source projects (Cassandra, Hadoop, etc) and are pre-sorted into training and test sets - though we are creating our own validation set. For our baseline models, we trained on the Cassandra source code, and tested on the Hadoop source code. We preprocessed the data by first removing all comments (both in-line and block) using an open source java tool [13] and then splitting each line by white space, and finally pruning by keeping only lines with at least 3 tokens (or n tokens for larger ngram models than trigram).

More precisely, the input for the ngram model was the first n tokens of a line, seperated by whitespace. So, the input may look like this:

input: [("int"), ("x")]

3
output: "int x = 0;"

For the LSTM and transformer model, the input/output pairing looks a bit different. We are actually trying to reconstruct the individual sentence from taking in the sentence as input and producing it as output. We also see the model’s kept tokens as it passes from model to decoder, so the input/output would look something like this:

SRC : public enum Type {
    KEY: enum {
    OUT: public enum Type {
Where the SRC is the input, the KEY are the tokens that the model decided to keep to reconstruct the sentence, and the OUT is the reconstructed sentence.

4.2 Evaluation method

We used the average edit (Levenshtein) distance across all lines of code generated versus the actual "true" line of code we were trying to generate as our metric. The formula for the edit distance between two strings is

$$lev_{a,b}(i,j) = \begin{cases} 
\max(i,j) & \text{if } \min(i,j) = 0, \\
\min\left\{ 
\begin{array}{l}
lev_{a,b}(i-1,j) + 1 \\
lev_{a,b}(i,j-1) + 1 \\
lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)}
\end{array}
\right. & \text{otherwise.}
\end{cases}$$

Therefore, our metric can be expressed as

$$\sum_{a,b} lev(a,b) / \sum_{N} 1$$

for every line of code a in N that we generated line b for.

4.3 Experimental details

For our baseline, we used N=3, 4 and 5 for a trigrams, 4-grams and 5-grams model. The n-gram model would generate a STOP token when the line was complete. We randomly sampled the next token if we did not find the previous two tokens in our dataset. Each of these models took about 3 minutes to train on the Cassandra source code, and about 15 seconds to run over the Hadoop test set.

For the Neural Model with LSTM: We trained our Neural model for 150 epochs with batch size of 128. We used a sgd optimizer for encoder and adam for the decoder, with a 0.001 learning rate for both encoder and decoder. Learning rate decay is enabled for both. We used a lagrangian reward for the encoder and set the reconstruction error constraint to 0.2. The multiplier for reconstruction loss is set to 5 but has decay. The decoder has a beam search size of 1. We removed Java key words like “while”, “do” and “this” from the stopwords list in the model because these words actually matter in the language model. The training took 6 hours.

For the Neural Model with a transformer encoder: Many of the metrics are similar to the LSTM model, with a few differences. We trained our Neural model for 160 epochs with batch size of 8. We used a sgd optimizer for encoder and adam for the decoder, with a 0.001 learning rate for both encoder and decoder. Learning rate decay is enabled for both. We used a lagrangian reward for the encoder and set the reconstruction error constraint to 0.2. The multiplier for reconstruction loss is set to 5 but has decay. The decoder has a beam search size of 1. We removed Java key words like “while”, “do” and “this” from the stopwords list in the model because these words actually matter in the language model. The training took about 12 hours, double the LSTM model training time.

4.4 Results

We used the same average edit distance metric across all models generated across the same processed hadoop test set.
We also observed the percentage of tokens kept from our neural models to see how well they could generalize to incomplete lines of code with fewer tokens entered by a user.

<table>
<thead>
<tr>
<th>model</th>
<th>percent of tokens kept (no whitespace)</th>
</tr>
</thead>
<tbody>
<tr>
<td>neural LSTM</td>
<td>73.41%</td>
</tr>
<tr>
<td>neural Transformer</td>
<td>50.74%</td>
</tr>
</tbody>
</table>

It is clear that both forms of the neural model clearly outperformed the baseline model in terms of the average edit distance metric, but taking a look at the percentage of kept tokens shows a bit more nuance that will be discussed further in the analysis.

In addition to the metrics outlined above, we also used tensor board to gain insight into the training process of our neural model. We plotted the decoder loss for both the LSTM and Transformer model.

LSTM model:

Transformer Model:

Another metric we plotted was the kept percentage of the neural model. Note that these graphs are with white space, so the LSTM model has a higher kept percentage than the actual kept percentage of tokens.

LSTM model:
Interestingly, the decoder’s loss is far noisier when using a transformer encoder for the neural model, but the percentage of kept tokens goes down over time for this model. This is in sharp contrast to the LSTM model, where the percentage of kept tokens increases with time. It seems that from these graphs, the transformer model is actually learning to ignore more tokens as time goes on, while still decreasing the loss - implying that it is learning something about the structure of code, while the baseline neural model is just keeping more tokens to increase reconstruction accuracy.

5 Analysis

It appears that while both neural models (transformer and LSTM) outperform the ngram baseline significantly, there is an issue of over fitting for the LSTM baseline that is not present in the transformer architecture. This is apparent when looking at the output the LSTM model produces. Here are some examples with the LSTM model:

Example 1:
SRC : public String getHadoopVersion() {
KEY: publicstringgetHadoopversion()
OUT: public String getHadoop()

Example 2:
SRC : putCmd(path, "REMOVEACL", null, false);
KEY: putCmd(path,REMOVEACL","
OUT: path("REMOVEACL", REMOVEACL")

Clearly, especially in the first example, it is choosing to keep many of the tokens present - while this explains the rapid drop in edit distance from the ngram model, it does pose the problem of usefulness as an autocomplete tool if it needs all of the tokens a user types in a sentence to perform well. Luckily,
the transformer model does not suffer from these issues. While the average edit distance is higher for the lines of code that it creates, it seems to be leveraging actual coding patterns instead of keeping the entire line of text. This is show in in the following examples with the transformer model:

Example 1:
SRC: this.proto = proto;
KEY: proto=proto;
OUT: this.proto = proto;

Example 2
SRC if (numOffSwitchContainers == 0) {
KEY (numOffSwitchContainers==){
OUT if (numOffSwitchContainers == 0) {

In these examples, it is keeping far fewer tokens than the LSTM model and seems to actually be able to deduce output lines from common coding patterns. In the first example, it sees a property being set to itself and implicitly fills in the "this." token. In the second example, it sees an if statement without the "if" token or the token that it is being set equal to, and is able to fill those in automatically. Though this model performs less well in terms of the edit distance metric, it clearly greatly outperforms in terms of the kept token percentage, and is therefore more useful as a tool for code autocomplete, as the user will have to type far fewer tokens for the model to get an idea of how to complete the line of code.

6 Conclusion

We have successfully shown that by learning to autocomplete code as a communication game, we can significantly outperform a simple rule-based baseline of relatively large ngram length. While the baseline model often falters when exposed to new examples from a separate codebase that it has never seen before, The neural model is able to generalize well, leveraging the similar structure of code across a codebase used for training, and one we are evaluating on.

We also realized that the primary issue with using the neural model with a standard LSTM for an encoder was that the compression rate was relatively low. The kept percent of tokens was as high as 73.41%, and even higher with whitespace. This implies that, while the neural model had impressive accuracy, it was in danger of simply keeping every token and reconstructing the line of code from that. This was obviously an issue, as the point of the task was to generate lines of code off of few tokens as possible. To resolve this issue, we introduced a transformer encoder layer architecture in place of the LSTM and saw much better results. While the edit distance went up slightly, the number of kept tokens plummeted to about 50.74%. This implies that using the neural model with a transformer encoder layer is a much better fit for the problem, as we were still able to outperform the baseline significantly in terms of our metric of edit distance, while also optimizing for keeping as few tokens as possible.

In conclusion, we show that a neural model can significantly outperform an n-gram baseline in terms of the average edit distance between generated and true lines. We also demonstrate that while the neural model described is in danger of overfitting when using a traditional LSTM encoder, if we swap in a transformer encoder with encoder layers in place of the LSTM, the model learns the latent structure of the code much better as it keeps fewer tokens while still having significantly better metrics than the baseline. Future work for this research would include extending the transformer architecture in our model to modify the decoder layers, and training on a more diverse corpus to see if our neural model can better generalize to unseen codebases.

References


A Appendix (optional)

Baseline loss for Neural model:

Reward for neural model:

**LSTM:**

![LSTM Reward Graph]

Transformer:
Key log likelihood for neural model:

**LSTM:**

**Transformer:**

Transformer Architecture