Language agnostic Completion Engine

Felix Brei Anja Sieke

February 23, 2018

Abstract

More students than ever are faced with the challenge of becoming good programmers, but this process takes time and requires learning from a lot of examples. We will take a brief look at the possibilities of automating this process using artificial intelligence, to create a kind of virtual mentor that can make suggestions to aspiring programmers based on the patterns and best practices that it has found in a collection of examples.

1 Introduction

Machine learning has recently become more popular than ever before and is being applied in research disciplines far beyond computer science. This has resulted in the necessaty for many students to learn how to program. Being pressured with assignments and deadlines, students tend to be satisfied with programs that 'get the work done', but often suffer from a low quality. It is common to avoid error checking, copy and paste code from the internet or stick to very low level language constructs instead of utilizing more advanced techniques that get the job done more efficiently and with actually less hassle for the one that is writing it (think of list comprehensions in Python for example).

Becoming a good programmer is a process that takes a lot of time and devotion and ideally a mentor that can help get things done 'the right way', but often students (especially in the humanities) may be devoted enough, but lack the time and a mentor to become proficient, which is very unfortunate.

Our goal is to provide such a mentor for aspiring programmers in the form of a program that utilizes artificial intelligence to find best practices employed in a programming language and make suggestions based on its own findings. In theory it should be possible to apply machine learning techniques on a big collection of source code that is considered high quality and find patterns, just as companies like Netflix or Amazon find patterns in their customers preferences. In this paper, we will not just report our findings, but will also emphasize the steps that had to be taken in order to get there.

2 Prerequisites

In this section, we will give a brief overview of the terms and definitions that we will use throughout this report. These definitions should not be considered complete or even absolutely accurate, as they are a personal choice of words with the goal of giving the reader an impression of what we are talking about.

2.1 Neural networks

A *Neuron* is a programmatic structure (or function) that is based on neurons inside the brain. In its most basic form, it takes a vector of input variables, calculates a weighted sum and applies an activation function (typically a sigmoid function like *tanh*).

A *neural network* is a collection of neurons which are typically arranged in layers, where the input of a layer is the output of the preceeding one. The output of the last layer is called the output of the network.

2.2 Recurrent Neural Networks

A *recurrent neural network* is a neural network where the output of one or more layers is fed back as input to the network or a selection of units. This allows the network to make decisions based on previous results / events.

2.3 LSTM and GRU

TODO

2.4 Embeddings

TODO

2.5 Keras

Keras is a Python module that contains functions and classes to easily create and manipulate neural networks. It contains not only basic neurons, but also more involved units like LSTM-Cells or Embedding-Layers (plus many more).

3 Data preparation

Data preparation is one of the most important aspects in machine learning. While it is nice to have huge amounts of data at your disposal, there is also a high chance that it contains a lot of noise.

3.1 Dealing with comments

One kind of noise is introduced by comments within the source code. Although these may be informative to the reader, they will probably just distract our algorithm from learning the things that it should.

We addressed the issue about the comments with a simple substitution using regular expressions, after reading every single source file into a single string variable called `blob`. The relevant part of our source code can be seen in (1).

```
        import re

        comments = re.compile(r'#.*')
        cleaned_text = comments.sub('', blob)
```

Figure 1: Removing all comments in Python source code

This may seem specific to Python, but it is easy to add another line that deletes every substring starting with something like `//` or that is wrapped between `/*...*/`, covering a great deal of programming languages.

3.2 Punctuation and Operators

Another potential source of trouble is the fact that it is absolutely common to append something like a dot or comma directly to a name without adding whitespace in between. While perfectly clear to humans, a machine does not recognize `a,` and `a␣,` as the same thing.

We can easily remedy this by adding said whitespace, again using regular expressions. Example (2) deals with dots and commas, but we have also used this on the characters `(,),[,],{,}` and `=`.

```
        commas = re.compile(r',')
        cleaned_text = commas.sub(' , ', cleaned_text)

        dots = re.compile(r'\.')
        cleaned_text = dots.sub(' . ', cleaned_text)
```

Figure 2: Seperating punctuation

This obviously introduces a new problem as operators like `==` or `+=` are torn apart. This will again be fixed using regular expressions, as seen in (3).

```
cleaned_text = re.sub(r'= =', '==', cleaned_text)
cleaned_text = re.sub(r'\+ =', '\+=', cleaned_text)
```

Figure 3: Fixing operators that got seperated

3.3 Turning words into numbers

Since neural networks can only deal with numeric values, we have to turn all the words into numbers. This can easily be achieved by assigning sequential numbers to all the words, as seen in (4). At first, we make sure that there are no words consisting only of whitespace. Then we add a special token that will be used later in all places that we think are variable names, constants or other kinds of identifiers. After that we make all the items so far unique. If this was omitted, the length of the dictionary and the highest index number would no longer correspondend, which would cause a lot of overhead when we create a one hot encoding later. Then we use dictionary comprehension to create the dictionaries.

It is beneficial for the performance to create dictionaries for both directions of the lookup.

```
cleaned_word_list = [ w.strip() for w in word_list if w.strip()
  != '' ]
cleaned_word_list += ['<ID>']
cleaned_word_list = set(cleaned_word_list)
word_idx_pairs = list(enumerate(cleaned_word_list))

w2n_dict = { w: i for i, w in word_idx_pairs}
n2w_dict = { i: w for i, w in word_idx_pairs}
```

Figure 4: Turning words into numerical values

Just like any other language, a programming language follows the rule that there is a small number of words that appear extremely often, while the majority of words appears far less frequently. We use this to find out which words are keywords (like `if`) or just common enough to be considered part of the language (like the string `'__name__'`) and which words are just noise. The tokens can then mentally be split into two categories, those that carry meaning to the language and those that are just identifiers and will all be treated as equal later on.

So after splitting the whole collection of source code based on whitespace, we decided that a word needs the appear in at least a certain percentage of all documents to be considered meaningful (5).

```
from collections import Counter
all_words = cleaned_text.split(" ")
c = Counter(all_words)

min_fraction = 0.2
min_count = num_files * min_fraction

reduced_words = [ w for w in all_words if c[w] >= min_count ]
```

Figure 5: Cutting off uncommon tokens

Of course `min_fraction` is another parameter that can be tuned. We experimented with numbers from 0.01 up to and including 0.2 and received equally good results. We decided to go with 0.2 because this reduces the dictionary size from 151441 to 214 and thereby greatly reducing the amount of RAM needed to store the training data. Plus, if there are too many tokens in the dictionary the network might try to learn to differentiate between objects that actually mean the same.

Finally, the sequences of words can be turned into sequences of numbers, keeping all the ones that appear in our dictionary natively and replacing everything else with the special token `<ID>`. Translating these numbers back into words than gives us lines like (6).

```
['def', '<ID>', '(', '<ID>', ')', ':']
```

Figure 6: Sample line after translating the words into numbers

This is exactly what we are looking for. A scriptable text editor can use this type of output to make smart suggestions and just ask the programmer to fill out the `<ID>` placholders.

3.4 Creating the training data

Creating samples for training from these sequences is now rather trivial. Given an input length `w`, we need to create input sequences from a starting index `i` up to `i+w-1` and define the word at position `i+w` as the target value (7). Since we replaced the line breaks in our input with `<EOL>` tags and split the text based on these, we have to reintroduce them to make sure that the network can also learn when to break a line.

After that, the input data is turned into a numpy-array, while the labels are turned into one hot encoded vectors. This is crucial in classification tasks, as a small deviation around the correct target value could lead to wrong predictions.

```
    ls = sentence + [ reduced_w2n_dict['<EOL>'] ]

    x_vals = [ ls[i:i+window_size] for i in range(len(ls) -
    window_size) ]
    y_vals = [ ls[i+window_size] for i in range(len(ls) -
    window_size) ]
```

Figure 7: Turning the sequences into input label pairs

4 Networks used

4.1 Layout

Based on the success of the infamous Word2Vec model, we decided to add an embedding layer as the first part of our network. This has two advantages. First, embedding layers can be trained to group similar words closer together in a given vector space. This helps structuring the data for later layers and makes it easier to group words that have some kind of connection to each other. The second advantage is more technical, as the subsequent recurrent layer needs vectors as input and the embedding layer turns single values into vectors in a very natural way.

We then decided to go for a single recurrent layer of GRU cells to keep the number of parameters low. This gives us the opportunity to explore the parameter space a little bit to find a good network regarding size and effectiveness. All GRU cells use the `tanh` activation function.

The last layer consists of densely connected neurons using a softmax activation function, so we can access the index of the token that the network predicts using an `argmax` function. The size of this layer is predetermined by the size of our dictionary which means that this layer adds no additional hyperparameters.

4.2 Parameters

Thanks to the low number of parameters we have the rare opportunity to sample from the parameter space and find out how the network behaves. The two most influential parameters in our network are the size of the embedding and the number GRU cells.

We ran each combination of parameters once and waited five epochs for the loss to settle. We set aside a fraction of ten percent of our data to calculate the validation loss, as this is far more accurate than the training loss when assessing the quality of the network. To train the network we used the adam optimizer and the loss is calculated using the categorical crossentropy function, which is best practice in these types of machine learning.

As expected, a higher number in both dimension leads to a reduced validation loss. The best results were achieved with relatively high numbers in

Embedding Dimension	16	32	64	128
GRU Cells				
16	1.04725	1.0305	1.02196	1.01758
32	1.0316	1.02255	1.01265	1.00873
64	1.0206	1.01366	1.00847	1.00286
128	1.01684	1.01002	1.00372	1

Table 1: Relative training losses for predictions based on single inputs

both dimensions and we had hoped for the loss to reach a minimum with lower numbers.

Eventually we settled with $(128, 128)$ as our parameter pair and trained both networks with this combination.

Embedding Dimension	16	32	64	128
GRU Cells				
16	1.12393	1.12226	1.07682	1.05991
32	1.08078	1.04828	1.03805	1.02983
64	1.04889	1.03574	1.01911	1.00842
128	1.03613	1.02123	1.01179	1

Table 2: Relative training losses for predictions based on two input values

5 Results

5.1 Single Input Network

Using the aforementioned parameter pair, we trained another network to make predictions based on a single input token. The loss settled after three epochs so we decided to abort early.

We then ran a selection of input words through the network and sorted the output vector indices based on their corresponding values. Then we selected the first five of them, meaning the five best guesses the network would make. The results are depicted in (5.1).

Input	`if`	`import`	`def`	`(`
Prediction	`<ID>`	`<ID>`	`<ID>`	`<ID>`
	`not`	`sys`	`__init__`	`self`
	`self`	`os`	`main`	`0`
	`__name__`	`time`	`get`	`1`
	`len`	`random`	`close`	`x`

Table 3: Predictions made by the single input network

These results look very promising as it makes indeed a lot of sense that words like `if` or `import` are usually followed directly by an identifier. We can also see that common Python imports like `import sys` or `import os` are important enough to the network that it considers them helpful suggestions. This is exactly the kind of behavior that we were looking for.

5.2 Dual Input Network

Learning to predict the correct next word based on a pair of input words is a bit more trickier because there are much more possible combinations of inputs that the network has to learn. On top of that, there are much fewer samples per possible input than with single tokens.

Using the same architecture and parameters as before, we trained another network to make predictions based on tuples. The validation loss stopped dropping during the fourth epoch so we aborted training at that point.

Again we ran some common combinations of words through the network and recorded the top five outputs as recorded below.

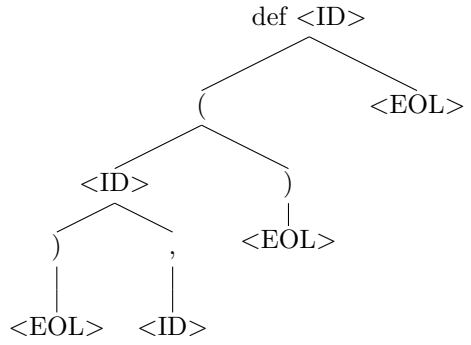
Input	`if <ID>`	`def <ID>`	`main (`	`( <ID>`
Prediction	`<EOL>`	`(`	`)`	`)`
	`==`	`<EOL>`	`<ID>`	`,`
	`.`	`.`	`self`	`.`
	`[`	`[`	`\*`	`(`

Table 4: Predictions made by the dual input network

These results look again very promising; almost every suggestion makes sense immediately and we can construct working lines of code for each of the suggestions.

5.3 Consecutive Output

We took this even further and fed back the output of the dual input network together with the second input token to create a new input pair. We repeated this process until we hit an `<EOL>` tag. The results are depicted below, starting with a mere `def <ID>` string.



As one can see, the suggestions already make some sense (except for the immediate `<EOL>`). Unfortunately, once the network starts suggestions multiple comma separated identifiers, it is trapped in an endless loop due to the small lookbehind window.

6 Outlook