Language agnostic Completion Engine

Felix Brei Anja Sieke

February 22, 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).

```
1        cleaned_text = re.sub(r'= =', '==', cleaned_text)
2        cleaned_text = re.sub(r'\+ =', '\+=', cleaned_text)
3
```

Figure 3: Fixing operators that got seperated

4 Networks used

5 Results

6 Outlook