Programming Language Predictor

Jacob Kulik and David Pogrebitskiy

Northeastern University

DS 2500: Intermediate Programming with Data

Dr. Laney Strange

April 27, 2022

Problem Statement and Background

Have you ever seen a snippet of code and wondered what programming language it was written in?

While hoping to delve deeper into the field of Natural Language Processing, we noticed the interesting intersection of supervised learning and word vectorizing. Relating these topics to our own interest in the syntactical differences between different programming languages was the obvious next step. Further, we hoped to create a tangible final product through a web application, rather than a collection of visualizations.

With the vast variety of skill sets among computer programmers, it could be really helpful to be able to paste a snippet of code into our model and figure out what languages it could be. Also, with the extreme similarities between certain languages, such as Java and C#, a fast way to decipher a given script could be useful for inexperienced programmers. Further, due to the nature of a large script, we hope that larger scripts could be quickly and accurately interpreted.

Introduction to Data

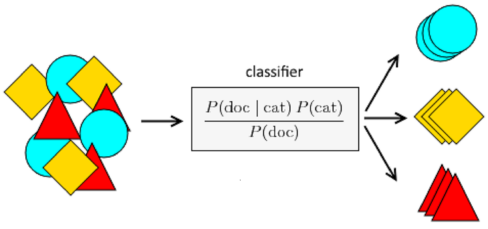
Our group utilized a Kaggle database, https://www.kaggle.com/code/amalhasni/creating-labeled-code-snippets-dataset/notebook, with sample code snippets across a vast variety of programming languages. More specifically, the dataset was created by performing a search query using a BigQueryHelper object on a GitHub repository dataset. Our kaggle data contains only two columns, with each value in the first being an entire programming script, while the second contains the language it is written in. Currently,

there are 131604 snippets of code written in 34 different programming languages, making the file size 1.35 gigabytes.

Data Science Approach

Given that our dataset was made up of scripts, features, and the related programming language, the target, supervised learning was our go-to data science method. Right away, we encountered some problems given that in regular natural language processing, getting rid of punctuation is an obvious first step. However, in our case, punctuation is an important part of certain programming languages and can help predict the language a given string is written in. Using regular expressions in a string-searching algorithm, we were able to account for punctuation during tokenization.

Next, in order to transfer these tokens into vector values, we utilized the machine learning library scikit-learn to perform TFIDF, an algorithm that filters out common tokens that wouldn't help in differentiating between languages. This is extremely important, since we use unique syntax to influence our algorithm, rather than comments, variable names, or other structures that are shared across the languages. After coming up with the most important vector values for each programming language, we used multinomial naive bayes, an algorithm commonly used for text classification, to further associate certain tokens with certain programming languages. The image to the right highlights the classifier's use of conditional probabilities derived from Bayes' Theorem to differentiate between different objects.



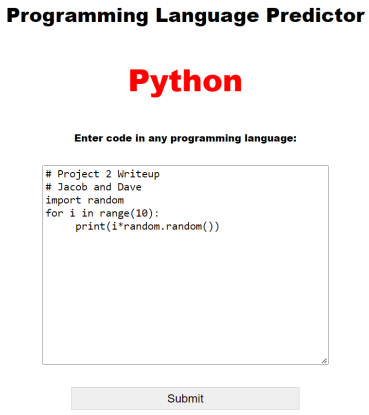
Once our algorithm was complete, we began working on a local host of an application, where it would simply take a user inputted string and return the predicted function. Using Flask, a framework written in python, our application worked locally through the setting up of a virtual environment. Once we confirmed that the application worked locally, we worked to host it on the web through a free hosting service, Heroku.

Results and Conclusions

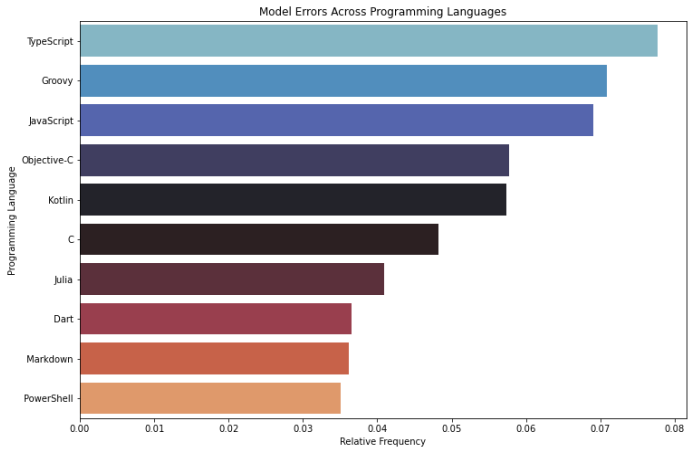
After many iterations and tweaks in both our algorithm and our web application, we have launched the final product using Heroku: https://program-predictor.herokuapp.com/. After starting out at a precision of around 0.80, we enjoyed testing the various ways to increase this score. With a combination of increasing vectorized features and grid-searching for the optimal parameters for the Naive Bayes' mode, our final precision score was 0.90 and our recall score was 0.82. It is important to note, however, that we are confident that with a long script rather than a few lines of code our algorithm will be even more accurate with its prediction because of the presence of more discriminatory tokens.

Applying our machine learning topics from lectures to our interest in the field of natural language processing was extremely valuable and a great learning experience. Although our algorithm is meant for programming script syntax, we realized over the course of the project that with a few tweaks, it could be applied to written language as well. Further, the use of regular expression in our algorithm to deal with the syntactical nature of programming languages was a great learning experience and helped us overcome our first major hurdle.

Launching our web application was also a great learning experience and helped us show a tangible, interactive side to our project. Below are two images, one showing how our app looks before a user input and the other showing how it looks after user input.



Additionally, a great learning experience was analyzing the source of errors in our algorithm. The graph shows the relative frequency of the most common programming languages that are incorrectly guessed. After further research, we came to the conclusion that the similarity between syntax in certain languages is the source of this error. For example, in Java and the C family, a semicolon is required after every statement, serving as a terminator. Also, certain languages such as TypeScript and JavaScript are extremely similar in nature. However, we found that with a longer input, our algorithm is able to eliminate the possibility of these interfering languages.



Future Work

With more time, we can continue to make small improvements and expand the scope of our project. Most notably, we hope to improve the design of our site to make it more friendly for all devices. We also hope to push Heroku's limits by showing graphics that display our algorithm's thought process as well as 2nd and 3rd choice programming language guesses. Further, our web application pushes the limits of Heroku's free memory allocation so coming up with a faster, more efficient solution could be a next step.

In regards to the algorithm, we know that in the future we can make minor changes that will continue to improve our accuracy and recall. We noticed that an increase in vectors from 3000 to 4000 or 5000 has a minor change, but other algorithms may produce more accurate results with a longer runtime. Additionally, we can look into dealing with user error and typos, since our program doesn't have a way of dealing with these and they may hinder the ability to guess the right language. Finally, including some sort of text pre-processing to the pipeline before tokenization could benefit the algorithm because it could remove blocks of code, like comments, that wouldn't be helpful in classifying the language.