Andrew Dircks (abd93)
Samuel Kantor (sk2467)
CS 4701
Final Report

Programming Language Classification

1. Introduction

Artificial intelligence, when utilized with written or spoken natural language, provides various interesting, useful technological applications. Natural language processing (NLP), this subset of AI, concerns itself with this understanding, manipulation, and application of human language in computer systems (Chowdhury, 2005). The NLP problem is structured by various levels of linguistic abstraction: lexical (word composition and independent meaning), syntactic (grammar of sentences), semantic (meaning of words in sentences), and pragmatic (outside context) (Liddy, 2005). With linguistic analysis and large amounts of data, software can perform powerful tasks on natural language.

Such tasks include sentiment analysis, text summarization, and text classification. Sentiment analysis concerns the computational task of understanding opinions and emotions from natural language, a popular problem with various commercial and psychological applications (Zhang et al., 2018). Text summarization simplifies text into shortened formats of important, relevant information; this area of research has implications in media, academia, and law (Munot and Govilkar, 2014). Text classification involves the grouping of texts into distinct categories based on data categorized by humans.

For our project, we focused on text classification, since there exists an extensive body of literature and software packages. We focused on an interesting classification problem: what programming language is an input text written in? As fervid software developers, we shared a motivation to make a code editor that uses NLP to predict the programming language of text and automatically adjust syntax highlighting! Instead of studying the interaction of *human* language with computer systems, we focused on *programming* languages, a recently developed branch of natural language used to communicate with computer systems themselves.

This classification problem is highly relevant for source control platforms. For example, GitHub determines the programming language of each file to generate repository metadata, aid in search, and alert security vulnerabilities (Ganesan, 2019). Furthermore, real-time classification of programming languages can automate syntax highlighting in text-editors, as successfully implemented through our web-based GUI. While language classification is typically done by mapping file extensions (e.g. *Python* is *.py*), certain extensions such as .h can be used by all variants of the C language including C, Objective C and C++. Discrepancies, mistakes, and absence of these extensions make language classification a non-trivial problem, well suited for AI, specifically NLP (Ganesan, 2019). This challenging AI problem is the basis of our project and allowed us to dive deep into the field of NLP, specifically with machine learning methods.

2. Learning Problem

We wish to predict the programming language of a code string, such that we can make a novel code editor with automated syntax highlighting. This problem can be formally stated as follows:

Let X be the set of all strings

Let Y be the set of all programming languages being considered

We define a function $F: X \rightarrow Y$ where

$\forall x \in X$, $F(x) = L$ s.t ***x*** *is written in language L.*

Given an n-sized training set $S = \{(Xi, Yi) \mid Xi \in X, Yi \in Y, F(Xi) = Yi, \; 1 \leq i \leq n\}$

We wish to learn a function $H: X \rightarrow Y$ that approximates F as well as possible. (1)

For training data, we elected to use the "Github Code Snippets" dataset on Kaggle, which contains 97,000,000 code string "snippets" labeled by one of the following languages: Bash, C, C++, CSV, DOTFILE, Go, HTML, JSON, Java, JavaScript, Jupyter, Markdown, PowerShell, Python, Ruby, Rust, Shell, TSV, Text, Yaml, and UNKNOWN (text corresponding to none of the following 20 languages) (Github). Figure 1 shows the distribution of programming languages in the entire dataset; note that this distribution is not a uniform distribution.

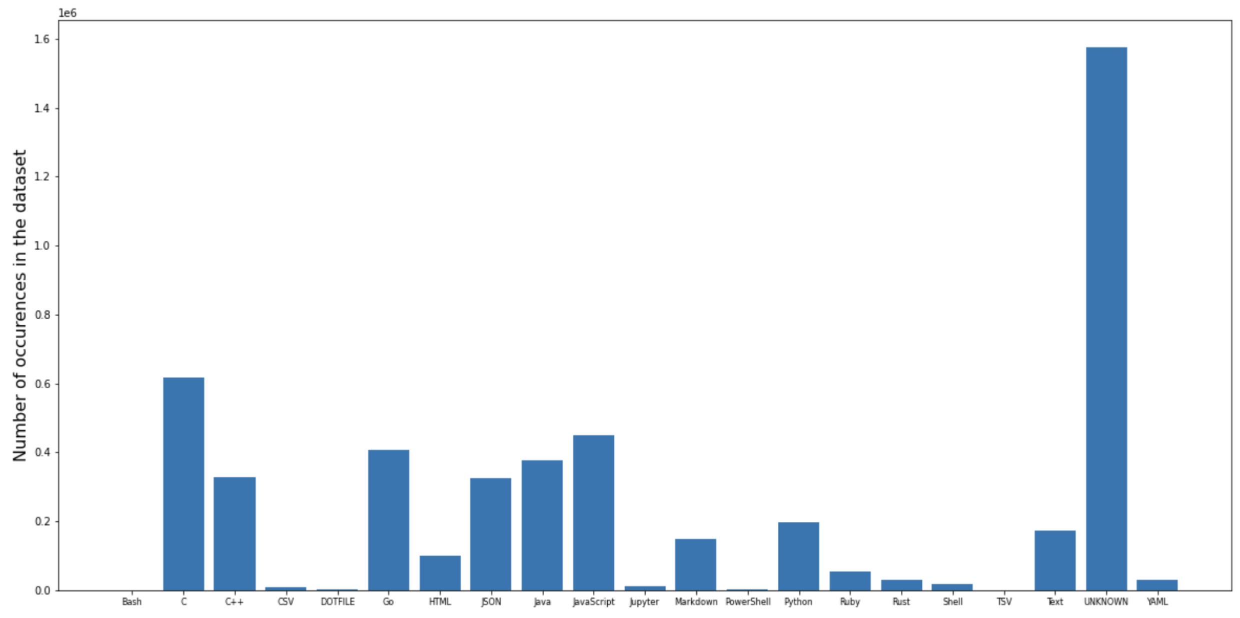

Figure 1: Distribution of programming languages.

This database encompasses most popular languages and the distribution of different languages is reflective of real-world environments. We chose this database for several reasons. Firstly, most popular languages are included in the classification set Y, which is appealing for our use case. Second, as the data is a random sample of GitHub repositories, we assume that the distribution approximates the mainstream popularity of each language. As each sampled repo has over 10000 stars we assume the code is of high quality and syntactically correct. Finally, the majority of instances correspond to UNKNOWN programming languages which is reflective of real-world environments; the majority of possible strings are arbitrary and will not match any of the programming languages in our language set. Learning with UNKNOWN text helps the model to discriminate between code and arbitrary text.

Given the size of the data and the large number of classes (languages), machine learning algorithms are a logical focus of our algorithmic methods. The structure of this large dataset is conducive to the standard machine learning formulations: train the data on a collection of (*x, y*) pairs, then generate a prediction *y** on a query *x** (Jordan and Mitchell, 2015). To accomplish this prediction, we employ various strategies for preprocessing, weight learning, and parameter tuning on various models, and we then analyze the results.

3. Preprocessing methods

A core part of all NLP systems is the mapping of raw data strings (sequences of bits), to meaningful linguistic units (Palmer, 2010). This conversion typically occurs before any model fitting or optimization, hence "preprocessing". Text preprocessing works primarily at the lexical level, where it breaks up strings into tokens (words, typically) based on predetermined delimiters. The programming languages we are mapping are all space-delimited languages, which makes breaking individual lines into words straightforward (Palmer, 2010). The first major discrepancy of our programming language data with typical, written language processing is the notion of "sentences". In code, there is no direct notion of sentences, but rather "lines of code"; in general, each line of code represents an individual instruction that is syntactically relevant to the respective programming language. Since each code snippet in the dataset has approximately 5 lines of code, it is impossible to look for cohesive, logical relationships between variables (words), as one may do when processing a paragraph of text. For this reason, we focus on utilizing simple, vector-based tokenizers to represent the lexical units of our data, then applying frequency transformations on each preprocessed snippet to normalize our data.

The first stage of our preprocessing and feature representation is generation of a "vocabulary", i.e. list of words/tokens that appear in our dataset. This procedure is relatively simple: iterate through each snippet in our *training* data, split the string by white-space and punctuation into individual words, and build up a full list of tokens that are found.

In addition to building the vocabulary with single tokens as elements, we introduce the concept of n-grams to represent the sequential nature of words in code. We focus only on unigrams and bigrams, since they are the most computationally tractable and effective in machine learning problems (Canvar and Trenkle, 1994). Given the line of Python code

```
return x + y
```

the bigram vocabulary would be: `(_ return)(return x) (x +) (+ y) (y _).`

Every pair of adjacent tokens becomes its own element of the vocabulary, increasing the size exponentially. With this large increase in size, computational complexity increases, but expressive power also increases (Canvar and Trenkle, 1994). For our experiments, we use a combination of unigrams and bigrams, i.e. a model's vocabulary may have single-world (unigram) tokens and two-word (bigram) tokens.

Next, we use our vocabulary to represent each snippet in vector form. Two popular encoding strategies are employed, Term Frequency (TF) and Term Frequency Inverse Document Frequency (TF-IDF), to vectorize each snippet. For a vocabulary $V = [w_1, ..., w_N]$ of size N with words/tokens w_i, each snippet $\boldsymbol{x} \in \boldsymbol{X}$ with word-count $|\boldsymbol{x}|$ is represented using TF transformation as

$$\left[\frac{count(w_1)}{|x|}, \frac{count(w_2)}{|x|}, \dots, \frac{count(w_N)}{|x|}\right] \tag{2}$$

where $count(w_i)$ is the number of times word i is found in $\boldsymbol{x}$ (Manning, 2018). This measure eliminates bias from length of snippet, that would be introduced if we created a vector of only *count()* values. Thus, short and long snippet representations are normalized by TF to allow for (relatively) even feature representation.

This vectorization can be further normalized by the TF-IDF algorithm, which reduces the weight of globally common words (e.g. "the" in English text) and values words shared in a small group of snippets (Ramos, 2003). For example, in our cas the "int" keyword is used in many languages (C, C++, Java, Python, etc.), so it's relative significance is low in determining which language we are in, thus it will be weighted less. Mathematically, we vectorize a solution $\boldsymbol{x}$ as

$$[count(w_1) * log(\frac{|X|}{totalCount(w_1)}), \dots, count(w_N) * log(\frac{|X|}{totalCount(w_N)})] \tag{3}$$

where $totalCount(w_i)$ is the number of times word i is found in all snippets (Ramos, 2003). This metric has proven successful in text querying, but it's effectiveness in analyzing programming languages is unclear. We analyze its effectiveness in the coming sections.

Note, we also attempted several other techniques such as skipgrams to skip over certain punctuation when forming bigrams (on the assumption this would yield more useful information). However, this surprisingly reduced the accuracy of models, which indicated to us that punctuation is important. Additionally, we tried to reduce the number of features by computing a correlation matrix using the pearson correlation coefficient to show the correlation between all the features and each unique language. This was achieved by appending 21 columns to each row (using one-hot encoding) where the new column I is 1 if instance is of Language I, and 0 otherwise. Correlation between each feature and language variable is computed to get the mean correlation between each feature and language choice. We then removed any features that had an abs(corr to a language l) < 0.03 for all languages l. Our data matrix of 4.5 million rows and millions of language features per row was unfortunately too large for our compute resources to calculate, so we calculated correlation by using stochastic methods of sampling rows to compute correlation, as well as by ignoring features below threshold counts. Unfortunately, none of these latter methods proved effective in increasing the accuracy (perhaps accidentally removing features that were important), so we concluded that removing uncorrelated values was not computationally tractable or beneficial for our purposes.

4. Naive Bayes approach

After feature representation and normalization, the processed code snippet data serves as training data for our machine learning model. The first, and simplest, algorithm we use to train our model is Naive Bayes. This classification algorithm assumes features are independent, given class; despite this largely simplifying assumption, Naive Bayes classifiers compete well with more sophisticated algorithms in practice (Rish, 2001). This simplifying assumption is expressed

$$P(x \mid C) = \prod_{i=1}^{N} P(x_i \mid C) \tag{4}$$

where $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]$ is a feature vector (i.e. snippet after preprocessing) and C is a class (i.e. programming language). Clearly, this is not an assumption we can make about our data, since, for a given programming language, features (tokens) are highly dependent on each other. For example, in Python, the token “`as`” will be dependent on the prevalence of token "`import`", due to the common expression "`import _ as _`". Regardless, Naive Bayes classifiers have proven to work well in datasets with features dependencies, despite the underlying assumption, since optimal classifiers are found if estimated and actual distributions match (Rish, 2001).

Let Ω represent the decision space of our problem, i.e. all possible TF-IDF vectors based on a given vocabulary. Let $c \in C$ be a class (programming language). Let $g : \Omega \rightarrow C$ be the "correct" mapping we are trying to learn, and let $h : \Omega \rightarrow C$ be the learned function. In Bayesian optimization, we define functions f_c for each class c, then predict the class with the highest respective function for a given input. Thus, $h(\boldsymbol{x}) = max_c\ f_c(\boldsymbol{x})$. To determine f, we use Bayes' rule, yielding the Bayes classifier (Rish, 2001)

$$h^*(x) = arg\,max_i P(X = x \mid C = i)\, P(C = i) \tag{5}$$

However, at high dimensions, this is computationally intractable, thus we use the "naive" assumption in (4) to approximate f with the Naive Bayes classifier (Rish, 2001)

$$f_i(x) = \prod_{j=1}^{N} P(X_j = x_j \mid C = i)\, P(C = i) \tag{6}$$

5. Support Vector Machine approach

We also experiment with linear support vector machines (SVM), a machine learning algorithm that searches for a (linearly) dividing hyperplane in the feature space to group data into classes (Pupale, 2019). SVMs have proven effective in high dimensional problems, which is valuable in our large decision space (n-gram vocabulary) and high number of classes

(Suthaharan, 2016). In the two-class case, we use the straight line equation, with $\boldsymbol{w}$ as weights and b as an intercept, to define each class C_i by

$$C_1 = \{wx + b \ \geq 0\} \tag{7}$$

$$C_2 = \{wx + b < 0\}$$

These class definitions can be parameterized by straight lines, in which case our objective is to maximize the distance between each line, i.e. each class (Suthaharan, 2016). This maximization of weights ensures the largest distinction of classes and can be equivalently represented as minimizing $w^T w$. We also want to penalize prediction error, which leads to the following SVM formulation

$$minimize_{w,c} \ w^T w \tag{8}$$

$$subject\ to\ c^*(wx + c) \ \geq 1$$

where c^* is the correct classification,$c$is the prediction of $\boldsymbol{x}$, and w is the weight vector of the dividing hyperplane. In *linear* SVMs, we minimize *hinge loss,* which is represented as the sum of $w^T w$and the max of the constraint value and zero (Suthaharan, 2016). This formulation allows for, while penalizing, misclassifications, since it is rare that data is perfectly linearly separable.

To minimize this loss function, we use *stochastic gradient descent (SGD)* a popular optimization algorithm for machine learning systems. SGD differs from regular gradient descent, as it computes an approximate gradient using a random sample of instances in the data instead of the entire dataset. In our implementation, at each epoch, observations O = {(Xi, Yi)} are randomly sampled from the training set, and we subtract each weight Wj by (the mean gradient of all losses in O with respect to Wj) * (learning rate η). Note that stochastic gradient descent is beneficial here as there is less processing time required. However, in other applications which may relate to non-linear learning, SGD helps to escape local minima using the noise and therefore increases the chance of finding global minima.

6. Perceptron approach

The next supervised machine learning algorithm we considered is perceptron learning. The perceptron algorithm is a famous algorithm, developed here at Cornell in 1957 by Frank Rosenblatt, designed to mimic the behavior of a neuron. A perceptron computes the weighted sum of its inputs by performing a dot product between the weight vector W and associated input vector X. A bias is additionally added to the weighted sum, but typically in courses such as CS4700 at Cornell, this bias is denoted as weight W0 and an X0 = 1 is prepended to all feature vectors within the training set. Then a step function is used that sets the output of the perceptron to 1 if wx >= 0, and 0 otherwise. We can then utilise the simple perceptron learning rule that does the following:

1. Initializes weights all to 0
2. For i = 1 to N (Where N is the number of items in training set)
    i. Compute hypothesis h for Xi which is +1 if wx >= 0, and 0 otherwise.
    ii. Update vector W to W + α Xi (Yi - h), where α is the learning rate

Step 2 can be repeated until a linear separation is achieved.

7. Passive Aggressive Classification approach

This algorithm differs from the previous supervised learning approaches in that it can be labeled an *online learning* algorithm (Lu et al., 2016). Perceptron-based algorithms update internal weights on misclassified training instances, while passive aggressive learning algorithms exploit correctly classified instances (Lu et al., 2016). In general, passive aggressive classification solves three variants of the hinge loss function on each training instance, namely

$$
\begin{aligned}
&\frac{1}{2}||w - w^T|| \text{ s.t. } c * (wx + c) = 0 \\
&\frac{1}{2}||w - w^T|| + c * (wx + c) \qquad (9) \\
&\frac{1}{2}||w - w^T|| + c * (wx + c)^2
\end{aligned}
$$

The minimum of these functions is taken, as weights *w* are optimized (Lu et al., 2016). In some cases, this adjusted loss function allows for continual, online learning of the model over time, as new data becomes available. In our case, however, we experiment with the new update rules in hopes of some performance gain.

8. Multi-Class Classification by Combining Many Binary Classifiers

As (purportedly) diligent students, we were originally under the wrong impression that the above 3 algorithms apply only to binary classification, as is typically suggested by undergraduate course material. In actuality, these classification algorithms - as in our use case - can be adapted to classify multiple classes effectively as well. We found that we could use the "One VS All" (OVA) scheme to combine many binary classifiers to accurately classify all the K different classes of programming languages. For each of the K languages, we learn a binary classifier drawing a boundary to discriminate between the other classes and itself. Then when classifying, we compute a confidence score for each language class boundary, based on an input point (which grows very positive as a point gets further from its boundary in the correct direction, and very negative as a point is further from boundary in incorrect direction). We return the language corresponding to the maximum confidence score. Thus, we achieve accurate multi-class classification using binary-classification techniques as a baseline.

9. Considering Nonlinear Methods and Hyperparameter Learning

We did consider more sophisticated methods than the above to try to learn a more accurate function. However, by increasing the cardinality of H, the set of possible learned hypothesis functions (as is the case for most non-linear methods) the likelihood of overfitting increases. Furthermore, non-linear learning methods typically require learning more parameters in training, adding computational complexity. For instance, when attempting to learn on a neural network with hidden layers, we had to learn all the weights to those hidden layers, as well as the ones that go from each neuron in the hidden layer to the output layer. Given the millions of features in each input vector, these added weights make computing gradients and learning an accurate function very expensive. We also experimented with XGBoost, which is an open-source library for regularized gradient boosting, based on our knowledge that these models typically outperform others for tabular data (such as our numeric pre-processed data). However, such an implementation proved computationally intractable given the number of features.

We additionally tried hyperparameter learning such as grid search and gaussian process minimization. For instance, in our XGBoost test implementation we sampled the learning rate from a log uniform range of e^-3 and e^-1, and made ranges for other XGBoost parameters. By training the model using those parameters on K = 4 different folds over training data, we produced a mean model accuracy, and the gaussian process uses this measurement to intelligently optimize and select a good combination of next parameters. However, once again, the added complexity of hyperparameter learning made our model infeasible. Ultimately, we could not utilise hyperparameter learning or non-linear learning methods. Quite surprisingly, we found better performance in linear learning methods in comparison to non-linear methods, which we believe is a consequence of our large feature set.

10. Methods

For ease of use, we utilized a random sample of 5% of the full Github Code Snippet dataset containing 4,850,000 code snippets labeled with the programming languages listed above. This smaller dataset, at 3.3 GB, is more manageable to work with than the original 60 GB dataset. We randomly split the data into 85% training and 15% test sets to avoid false accuracy and over-fitting. As stated prior, we fit the preprocessing models (n-grams vocabulary creation and TF-IDF vectorizer) with *only* the training data, to avoid "cheating" by representing the vocabularies and frequencies in the testing data in our feature mapping.

To implement the preprocessing and machine learning algorithms, we use *Scikit-learn,* a Python machine learning library (Pedregosa, 2019). The following library classes are used: *CountVectorizer* to create the vocabulary and implement n-grams; *TfidfTransformer* to apply TF and TF-IDF vectorization of data on the respective vocabulary; *MultinomialNB* to implement the Naive Bayes approach; *SDGClassifier* with hinge loss to implement support vector machines with stochastic gradient descent; *Perceptron* to implement the perceptron algorithm; *PassiveAggressiveClassifier* to implement passive-aggressive linear optimization.

We performed our computational experiments on "The Cube", a Cornell Center for Advanced Computing (CAC) cluster. This resource has 32 parallel compute nodes, each with dual 8-core Intel processors ("TheCube"). Best performing models were serialized, saved, and transferred to our local GUI.

We used a simple accuracy metric to quantitatively evaluate the performance of our model. Once trained, these models were evaluated on the test data, and the amount of correct predictions divided by the total number of test snippets was recorded. Qualitative evaluation was performed by the GUI, which we will describe in detail later.

11. Results

The best achieved test accuracies of the four model algorithms are shown in Table 1. In all cases, the optimal preprocessing strategy included a vocabulary of unigrams and bigrams and feature representation with TF-IDF vectorization. In general, the Naive Bayes, Perceptron, and Passive-Aggressive classifiers performed approximately the same on the testing data. We are relatively happy with these results; the 21 language classification is a difficult problem, and a lot of the discrepancies between snippets (for example, C and non-OOP C++) are barely distinguishable by humans.

Algorithm	**Best Test Accuracy**
Naive Bayes	82%
Support Vector Machine	62%
Perceptron	81%
Passive-Aggressive	83%

Table 1: Best model test accuracy.

Next, we analyze the performance of our model over time, during training, by breaking the dataset into batches and calculating test data accuracy after each batch. Figure 2 shows accuracy versus the number of training samples for each algorithm. For all algorithms, approximate convergence occurs when about 20% of the training data has been fitted. The SVM with stochastic gradient descent has notably lower accuracy convergence than the other three classification algorithms. Figure 3 shows a similar trend, but it plots accuracy versus execution time instead of training samples. The Naive Bayes classifier takes significantly longer (~4 times) to train than the other three algorithms.

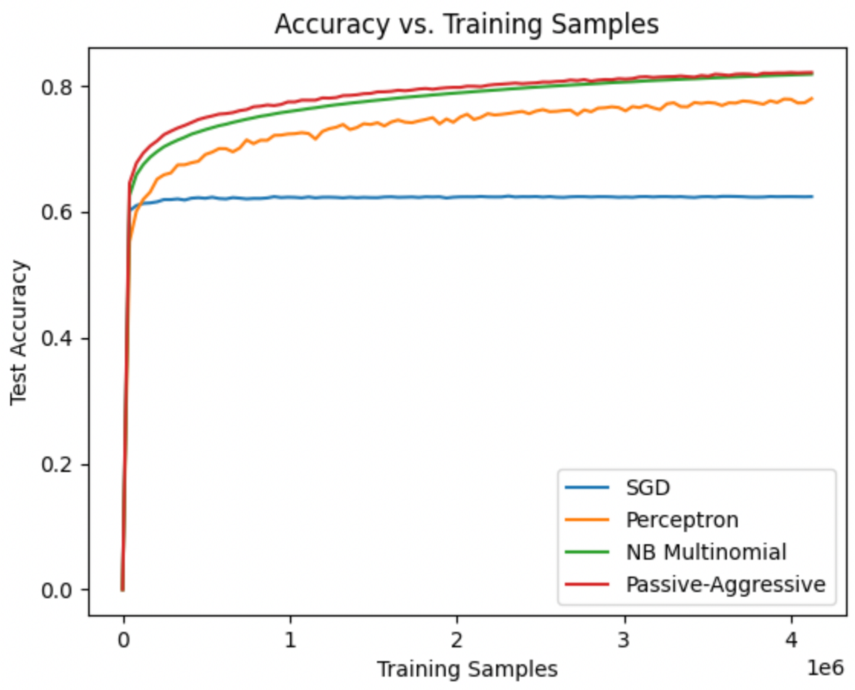

Figure 2: Accuracy vs. training samples for all four algorithms

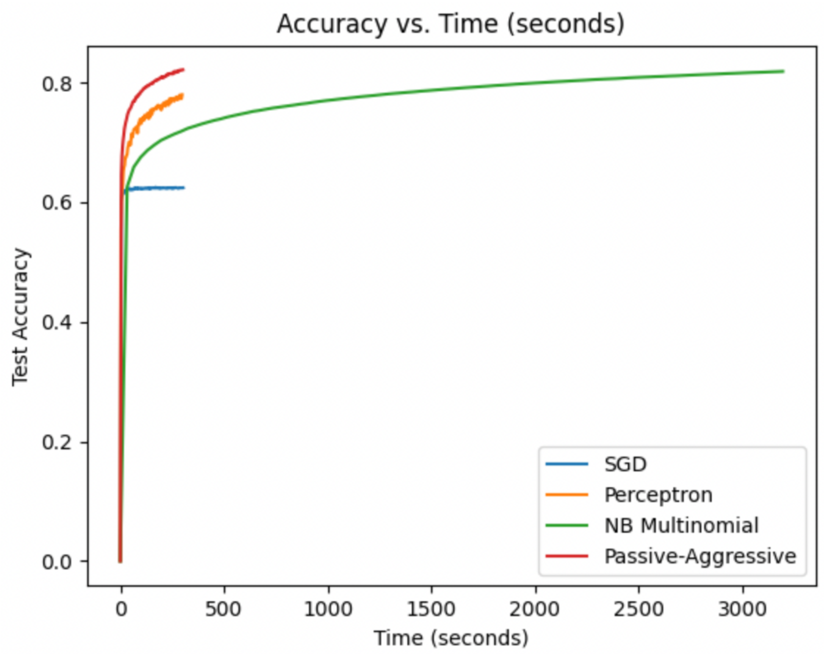

Figure 3: Accuracy vs. runtime for all four algorithms

To examine where these missed classifications are occurring, we test the models on the entire dataset, calculating the number of misses in each language. By Figure 1, since there is an uneven distribution of programming languages, we focus on the *proportion* of misclassifications per language class. For example, Bash has 574 total snippets in the dataset, and the naive bayes predicts incorrectly 107 times when Bash is correct, thus giving a "miss frequency" of 107/574, about 20%. Figure 4 shows this miss frequency metric for all languages and three classifiers: naive bayes, perceptron, and passive-aggressive.

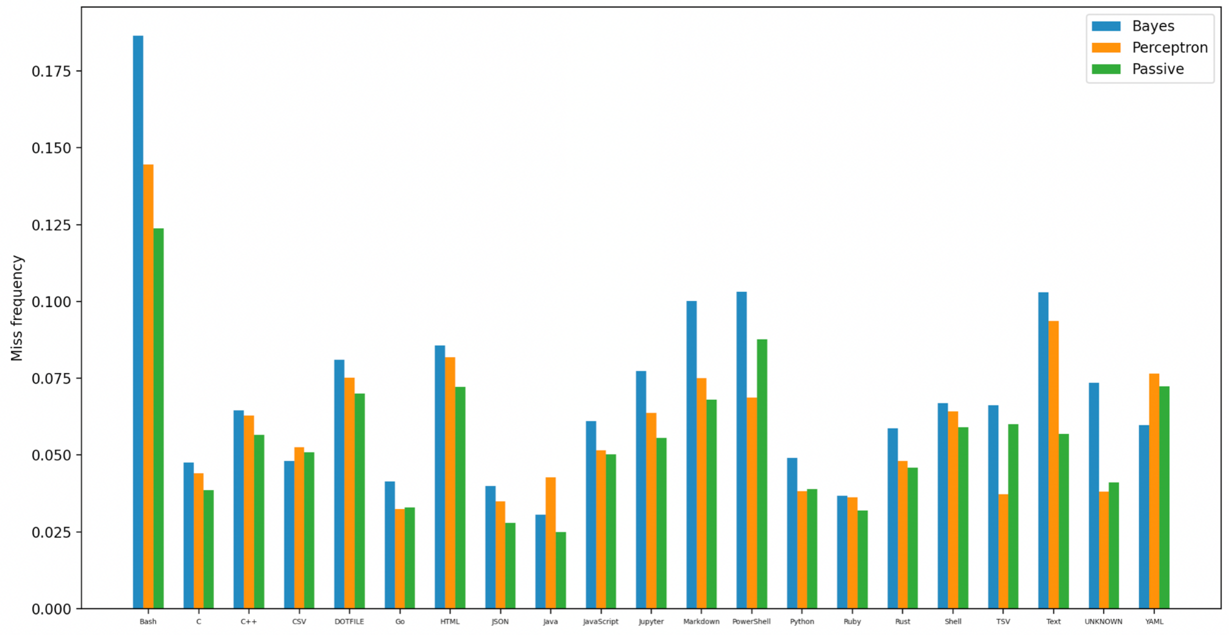

Figure 4: Miss frequencies for each language.

First, we notice the significantly high miss frequency of Bash; this may be explained by the low number of Bash snippets in the dataset, or other factors including Bash's similarity to other languages, like Shell. Similarly, other languages with relatively few snippets in the dataset (such as DOTFILE and PowerShell) have high miss frequencies. To visualize this relationship, we plot miss frequency as a function of the number of snippets in Figure 5.

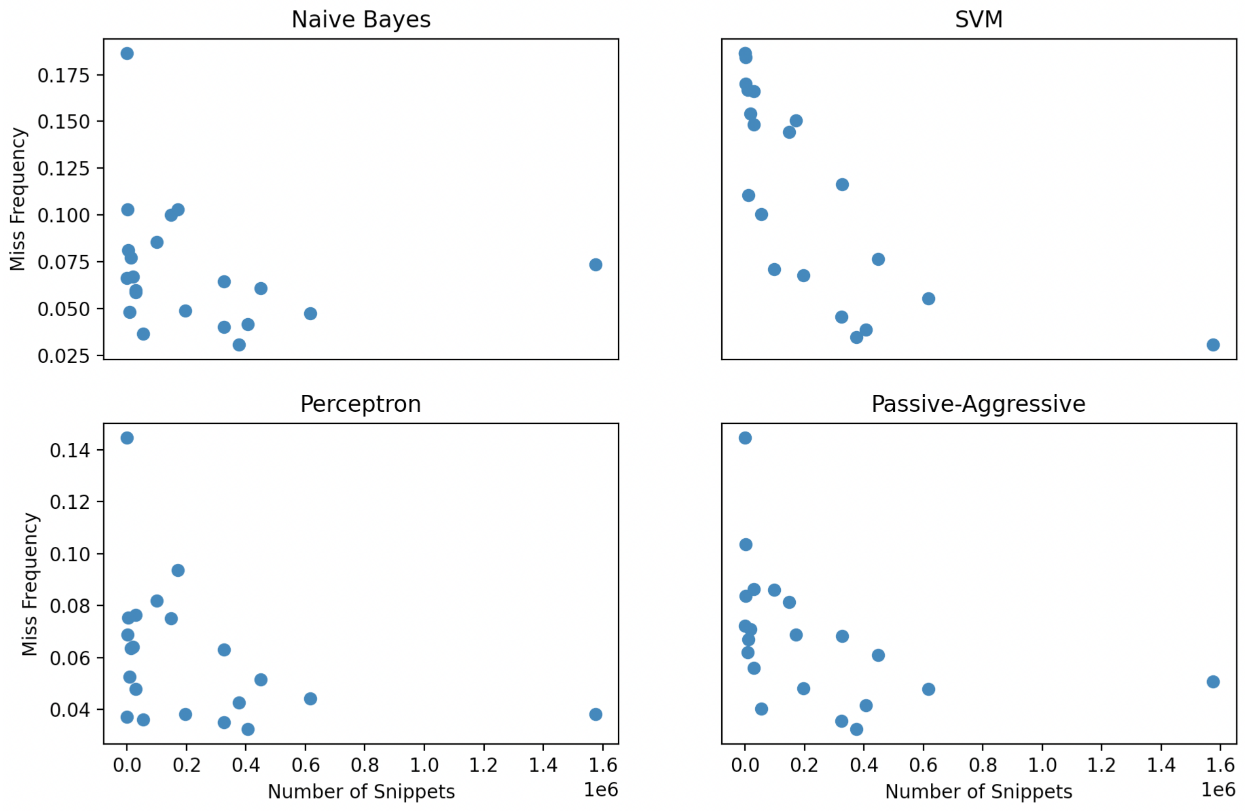

Figure 5: Miss frequency vs. number of occurrences. Each dot is an individual language.

There is a loose inverse relationship between miss frequency and total number of snippets in the dataset, per language. This result is aligned with our modeling strategy; we used non-weighted loss functions to maximize total accuracy, not weighted accuracy. As a result, classes with less training data are less understood by our machine learning models. This was a design choice in developing our model, since we expect users of our application to choose programming languages at approximately the same frequency they occur in the database.

On a smaller scale, we analyze individual code snippets that are predicted incorrectly by our models to get a sense of where our model goes wrong. In doing this, we did discover some inaccuracies in the dataset. For example, the following snippet was labeled as pure *C* code:

```
template <class T, class Cmp>
void TopN<T,Cmp>::ExtractUnsortedNondestructive(std::vector<T> *output) const {
	CHECK(output);
```

Clearly, this is *C++*, but the dataset was corrupted in some way. With this input, our models output *C,* the "correct" programming language in terms of the dataset, but for the end user, this is incorrect. It is difficult to tell how this bias has affected our model performance overall.

Here, we take an interesting example to demonstrate a specific behavior of our model. This code snippet, written in Python, appears in our source code for building the model itself:

```
from sklearn.naive_bayes import MultinomialNB, GaussianNB
from sklearn.linear_model import SGDClassifier
clf = MultinomialNB(alpha=0.001)
```

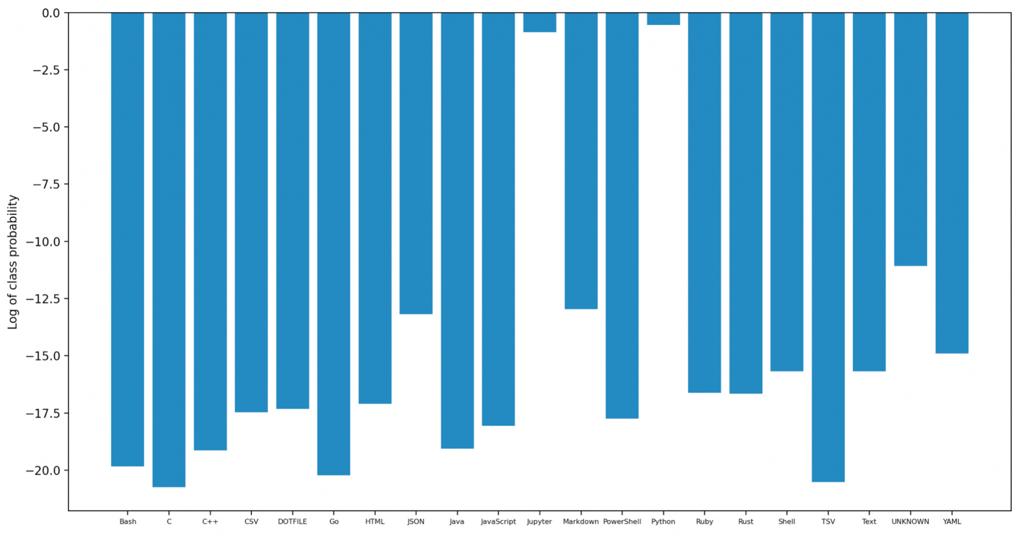

Figure 6

The Naive Bayes model predicts Python on this string. Figure 6 shows the log of each language's probability from the model output; the closer to 0 the more likely the programming language.

This figure shows Python as the most-likely class, with Jupyter following closely. With one (carefully crafted) variable name change, we change the output to something different. Take the snippet

```
from sklearn.naive_bayes import MultinomialNB, GaussianNB
from sklearn.linear_model import SGDClassifier
source = MultinomialNB(alpha=0.001)
```

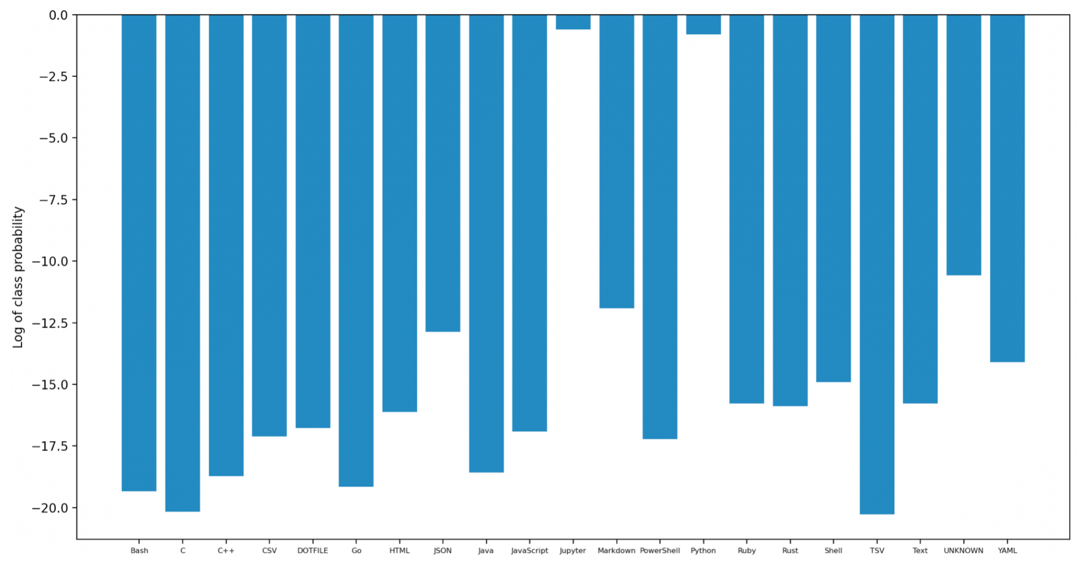

Figure 7

in which ‘clf’ is renamed to ‘source’. Our model now predicts Jupyter as the language … why? In the training data, there exist snippets of Jupyter notebook source code. As plain text, *.ipynb* files are essentially JSON encoded files with elements representing cells; the JSON key ‘source’ stores the raw text of a code block in a notebook. Additionally, Jupyter notebooks are popular for data science and machine learning applications, so the `sklearn` module, while Python code, is present in various Jupyter labeled code snippets. The token ‘source’ is the determining factor in whether or not the model predicts Python or Jupyter. Figures 6 and 7 show the difference in class probabilities that one word makes.

```
from nonsense_module import MultinomialNB, GaussianNB
from nonsense_module import SGDClassifier
source = MultinomialNB(alpha=0.001)
```

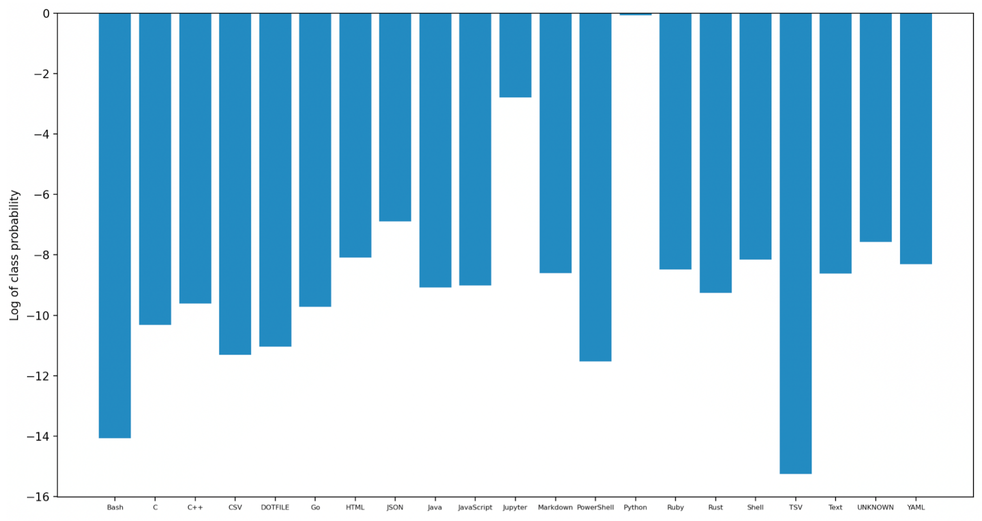

Figure 8

By replacing the ‘sklearn’ module with ‘nonsense_module’ in this code snippet, the probabilities change dramatically. No longer does the Naive Bayes module see ‘sklearn’ and associate this popular package with Jupyter and Python. It can now rely on normal syntax and tokens to make a more confident decision that the code snippet is Python.

12. Development of GUI Application

We created a web code editor that automatically adjusts the syntax highlighting of code to match the language predicted by our machine learning model. To achieve this, we utilized the popular open-source code editor ACE, which can be incorporated into websites and controlled using Javascript. As our machine learning models were all built on Python, we needed to devise a way of sending the raw code string in the code editor from Javascript to Python, and then sending the output of our model back to Javascript in order to highlight the syntax. The simplest way to communicate from Javascript to Python and also Python to Javascript was to set up a local web server using Flask, a simple, open-source Python backend framework.

First, we created the `index.html` file. This is a web page that embeds the ACE code editor into the page (where it is set to an exquisite dark theme with some sample text). Within this file we write our Javascript code, which calls an update function every 8 seconds that:

i) Calls our Flask server via a POST request on localhost, posting a JSON string representing the code in the editor and waits for our server (which is computing the language using the model) to respond.

ii) Once a response is received, the Javascript code calls the Flask server again using a GET request where it retrieves the predicted language mode for syntax highlighting. At this point, our Javascript code sets the mode of the editor to the predicted language which causes the syntax highlighting.

Next, we designed the methods for our local server on Python using Flask in `app.py`:

i) `index()` is called when the server is first accessed with a GET request. This loads our model and returns the main page, presenting the interface

ii) `postmethod()` handles the POST requests from Javascript code. It takes the form value for our code string, converts into plain text, and pre-processes the data in the same way we did originally for instances in our sample set. Then the formatted input is passed into our learning model to predict the matching output language.

ii) `getmethod()` handles the subsequent GET request from the Javascript. It simply reads the latest prediction and returns the jsdata.

By combining the power of Python, Javascript, and machine learning, we successfully created a fun, responsive and engaging code editor that seamlessly predicts the language as you type, changing the syntax on the fly! In many respects, this project has provided an exciting prelude to the future of code editors. We have shown it is indeed possible to use artificial intelligence to help developers automate and simplify their workflow, saving time and energy. Working on a novel, next-generation code editors has been a deeply inspiring and rewarding experience.

References

Cavnar, William B, and John M Trenkle. “N-Gram-Based Text Categorization.” 1994.

Chowdhury, Gobinda G. “Natural Language Processing.” *Annual Review of Information Science and Technology*, vol. 37, no. 1, 2005, pp. 51–89., doi:10.1002/aris.1440370103.

Ganesan, Kavita. “C# Or Java? TypeScript or JavaScript? Machine Learning Based Classification of Programming Languages.” *The GitHub Blog*, 2 July 2019, github.blog/2019-07-02-c-or-java-typescript-or-javascript-machine-learning-based-classification-of-programming-languages/.

“Github Code Snippets.” https://www.kaggle.com/simiotic/github-code-snippets

Jordan, M. I, and T. M Mitchell. “Machine Learning: Trends, Perspectives, and Prospects.” *Science*, Science, 17 July 2015, science.sciencemag.org/content/sci/349/6245/255.full.pdf.

Liddy, Elizabeth D. “Enhanced Text Retrieval Using Natural Language Processing.” *Bulletin of the American Society for Information Science and Technology*, vol. 24, no. 4, 2005, pp. 14–16., doi:10.1002/bult.91.

Lu, Jing, et al. “Online Passive-Aggressive Active Learning.” *Machine Learning*, vol. 103, no. 2, 2016, pp. 141–183., doi:10.1007/s10994-016-5555-y.

Manning, Christopher D., et al. *Introduction to Information Retrieval*. Cambridge University Press, 2018.

Munot, Nikita, and Sharvari S. Govilkar. “Comparative Study of Text Summarization Methods.” *International Journal of Computer Applications*, vol. 102, no. 12, 2014, pp. 33–37., doi:10.5120/17870-8810.

Palmer, David D. “Text Preprocessing.” *Handbook of Natural Language Processing*, by Frederick J. Damerau and Nitin Indurkhya, Taylor & Francis, 2010.

Pedregosa, F, et al. “Getting Started with Scikit-Learn for Machine Learning.” *Python® Machine Learning*, 2019, pp. 93–117., doi:10.1002/9781119557500.ch5.

Pupale, Rushikesh. “Support Vector Machines(SVM) - An Overview.” *Medium*, Towards Data Science, 11 Feb. 2019, towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989.

Ramos, Juan. “Using TF-IDF to Determine Word Relevance in Document Queries.” *Rutgers University*, 2003.

Rish, I. “An Empirical Study of the Naive Bayes Classifier.” *T.J. Watson Research Center*, 2001.

Suthaharan, Shan. *Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning*, by Shan Suthaharan, Springer, 2016.

“TheCube Cluster.” *Cornell Center for Advanced Computing*, 2021, www.cac.cornell.edu/wiki/index.php?title=THECUBE_Cluster.

Zhang, Lei, et al. “Deep Learning for Sentiment Analysis: A Survey.” *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, 2018, doi:10.1002/widm.1253.