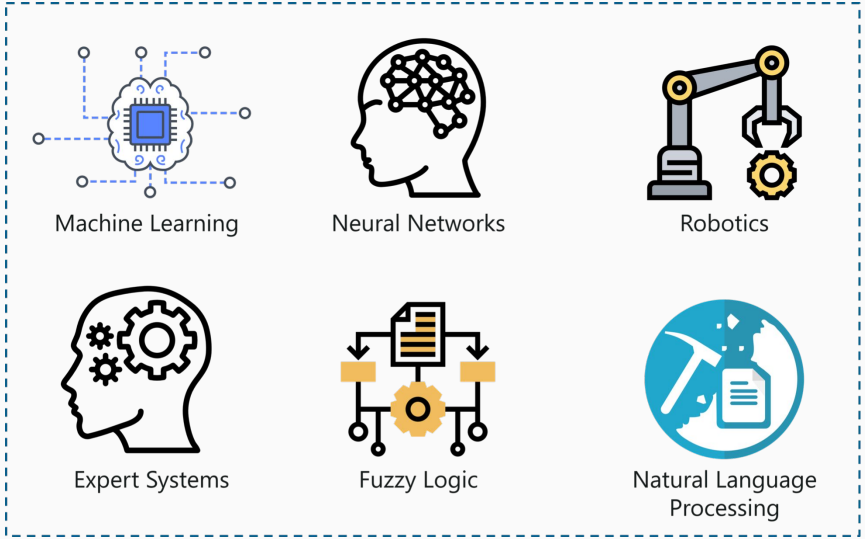
Programming Language Classification

Andrew Dircks & Samuel Kantor

CS 4701

Introduction

Where does our project fall within AI?

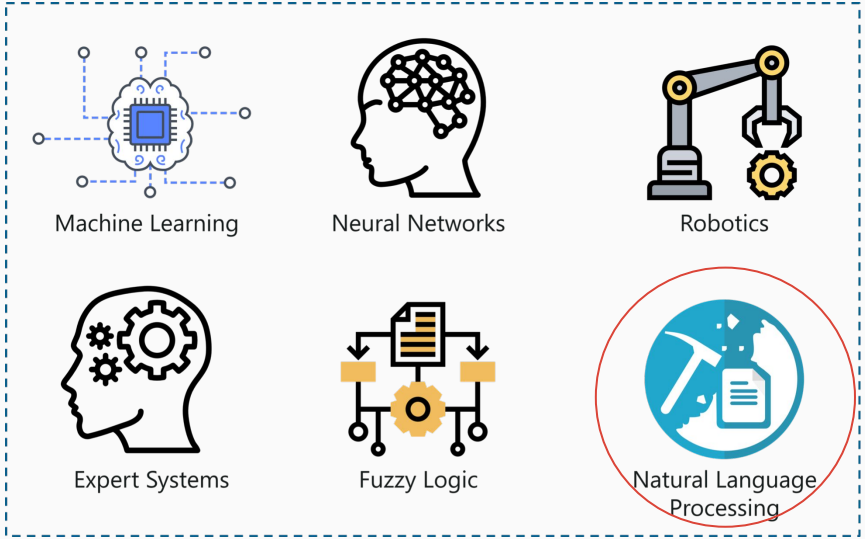


Introduction

Where does our project fall within AI?

NLP

- Working with written (programming) language



Introduction

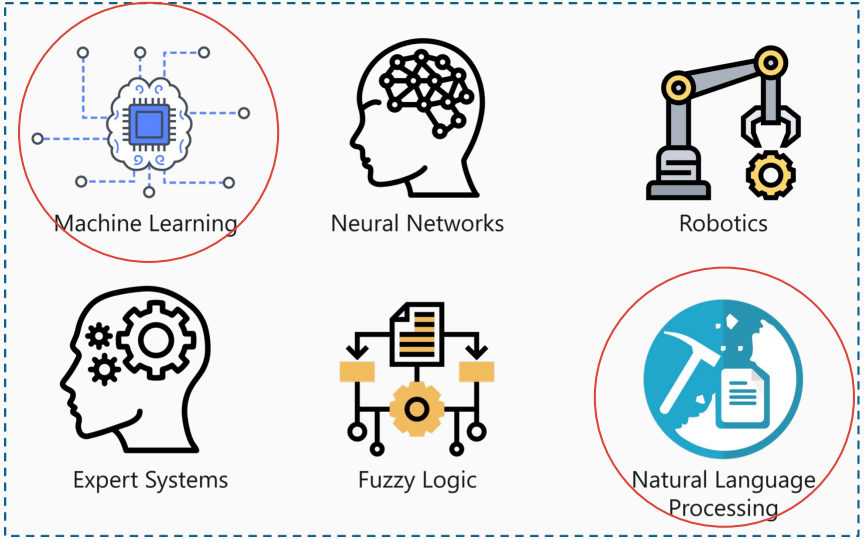
Where does our project fall within AI?

NLP

- Working with written (programming) language

Machine Learning

- Making strong predictions with ML



Natural Language Processing

“Understanding, manipulating, and applying human language in computer systems”

- Lexical, syntactic, semantic, and programmatic analysis
- Sentiment analysis, text summarization, **text classification**

Why text classification?

- Extensive body of literature and software
- Lends itself to interesting, useful user applications

Classification Problem

What programming language is an input text written in?

- Newly developed branch of natural language
- Relevance in source control platforms, like GitHub
  - Repo metadata, search functionalities, security vulnerability analysis
  - Non-trivial problem: cannot rely on file extensions only due to user discrepancies
- Practicality in text editors
  - Automatic syntax highlighting and formatting

Problem Formulation

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.*

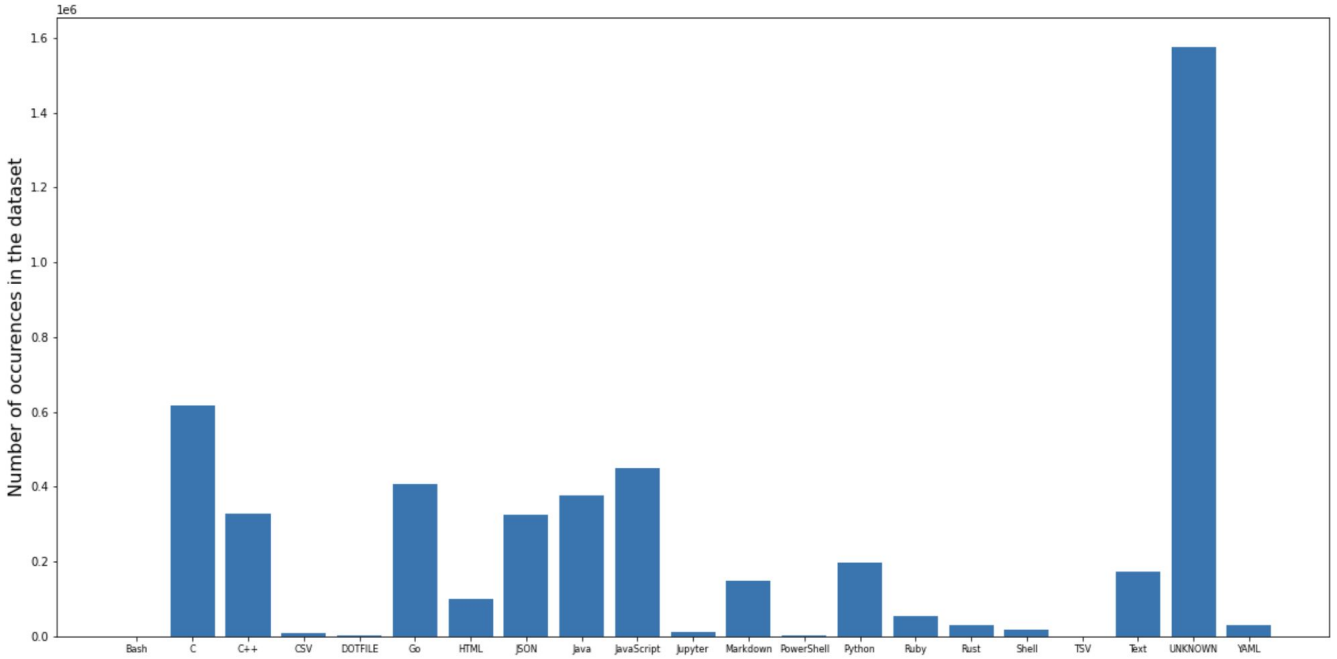
Given an n-sized training set $S = \{(Xi, Yi) \mid Xi \in X,\ Yi \in Y,\ F(Xi) = Yi,\ 1 \leq i \leq n\}$

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

Data

- “Github Code Snippets” on Kaggle
  - 97,000,000 code snippets labeled by programming language
  - Bash, C, C++, CSV, DOTFILE, Go, HTML, JSON, Java, JavaScript, Jupyter, Markdown, PowerShell, Python, Ruby, Rust, Shell, TSV, Text, Yaml, and UNKNOWN
- UNKNOWN categorizes all other programming languages

Data Distribution



- Non-uniform

- Reflective of real-world preference/popularity

- Random sample of popular Github repositories

Preprocessing & Feature Representation

- *Mapping of arbitrary length, raw data strings (sequences of bytes) to meaningful linguistic units*

Preprocessing & Feature Representation

- *Mapping of arbitrary length, raw data strings (sequences of bytes) to meaningful linguistic units*
- Tokenization and vocabulary generation

Tokenization and Vocabulary Generation

- “Tokens” as sequences of characters separated by white-space and punctuation
  - e.g. “int”, “def”, “class”
- Iterate through all samples, splitting string by predetermined delimiters, create a list of all tokens (vocabulary)

Preprocessing & Feature Representation

- *Mapping of arbitrary length, raw data strings (sequences of bytes) to meaningful linguistic units*
- Tokenization and vocabulary generation
- N-grams

N-grams

- Sequential nature of words in language is significant in understanding and classification
- *Bigrams:* every adjacent pair of tokens becomes its own element in the vocabulary

return x + y

`(_ return)(return x) (x +) (+ y) (y _)`

N-grams implementation

- Our vocabularies are composed of *unigrams* and *bigrams*
  - Anything more is computationally intractable, as including (n+1)-grams increases the size of the vocabulary
- Including *bigrams* in preprocessing greatly increases the expressive power of our models

Preprocessing & Feature Representation

- *Mapping of arbitrary length, raw data strings (sequences of bytes) to meaningful linguistic units*
- Tokenization and vocabulary generation
- N-grams
- TF and TF-IDF vectorization

Term Frequency (TF)

Vocabulary $V = [w_1, \ldots, w_N]$ of size N, where w_i is th i-th word/token in the vocabulary.

$count(w_i)$ is the number of times word i appears in $\boldsymbol{x}$.

Snippets $\boldsymbol{x} \in \boldsymbol{X}$, each with word count $|\boldsymbol{x}|$ represented as a vector by TF as

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

- Eliminates weighting bias of long vs. short strings, compared to term-count vectorization

Term Frequency Inverse Document Frequency (TF-IDF)

$totalCount(w_i)$ is the number of times word i is found in every snippet.

A snippet x is vectorized by TF-IDF according to

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

- Reduces the weight of globally common words in feature representation
  - e.g. “the” in English text

Preprocessing & Feature Representation

- *Mapping of arbitrary length, raw data strings (sequences of bytes) to meaningful linguistic units*
- Tokenization and vocabulary generation
- N-grams
- TF and TF-IDF vectorization

Machine Learning Approaches

- Naive Bayes

Naive Bayes

- *Assumes features are independent, given class*
  - We cannot make this assumption about our dataset, since tokens are highly interdependent within a programming language
  - However, NB is proven to work in datasets with feature dependencies
- Bayes' rule yields this formulation
  - Max Bayesian probability over all classes
- Applying our simplifying assumption

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

$$h^*(x) = arg\ max_i\ P(X = x \mid C = i)\ P(C = i)$$

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

Machine Learning Approaches

- Naive Bayes
- Support Vector Machines

Support Vector Machines

- Searches for linearly dividing hyperplane in the feature space to optimally separate classes
  - Minimize the distance between weight vector *w* distance, subject to all (feature, class) pairs on the correct side of the hyperplane
- ‘Hinge loss’ loosens this constraint by penalizing for misclassified data
- Stochastic gradient descent (SDG) minimizes this loss function with random batch sample instances to approximate the gradient

$$minimize_{w,c} \; w^T w$$

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

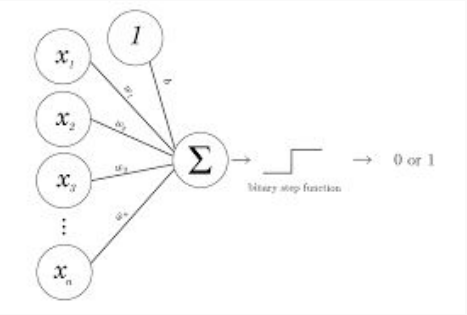
Machine Learning Approaches

- Naive Bayes
- Support Vector Machines
- Perceptron

Perceptron

- Designed at Cornell to *mimic the behavior of a neuron*
- Optimize a set of weights *w* to minimize classification error by iteratively updating via the learning rule

1. Initialize weights to 0
2. For all samples x_i in the training set
   a. Compute hypothesis *h* for x_i : 1 if *wx >= 0*; 0 otherwise
   b. $w \leftarrow w + \alpha x_i (y_i - h)$, for learning rate α



Machine Learning Approaches

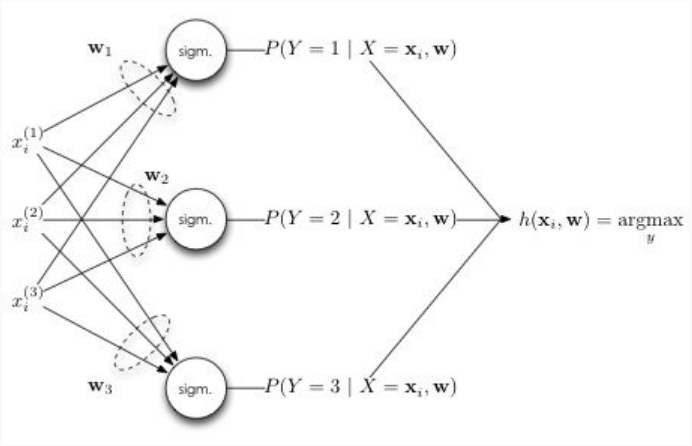
- Naive Bayes
- Support Vector Machines
- Perceptron
- Passive-Aggressive

Passive-Aggressive

- *Online learning* approach
  - Typically used for continually trained models
  - Here, we experiment with the more aggressive update rule
- Typical supervised learning algorithms (SVM, Perceptron) update weight vectors only during mis-classifications during training
- Passive-Aggressive uses an update rule that is *aggressive*
  - Updates weights with random probability on correctly classified data during training
  - Risk of over-fitting

Multi-Class Classification

- Our supervised learning models (SVM, Perceptron, Passive-Aggressive) are defined for *binary classification*
- To extend these algorithms to our 21 language classification problem, we use the One vs. All (OVA) scheme
    - For each of the 21 languages, we learn a binary classifier to draw the distinction between this class and all others
    - To predict data, we return the class associated with the highest probability



Machine Learning Approaches

- Naive Bayes
- Support Vector Machines
- Perceptron
- Passive-Aggressive
- Other attempts

Other (failed) Approaches

- XGBoost, Recurrent neural networks, Gaussian processes
- Complex, nonlinear approaches did not do nearly as well
  - Huge feature space makes learning extremely expensive for these algorithms
- The multi-class linear formulation had a balance of simplicity (for computation sake) and expressivity to yield good performance

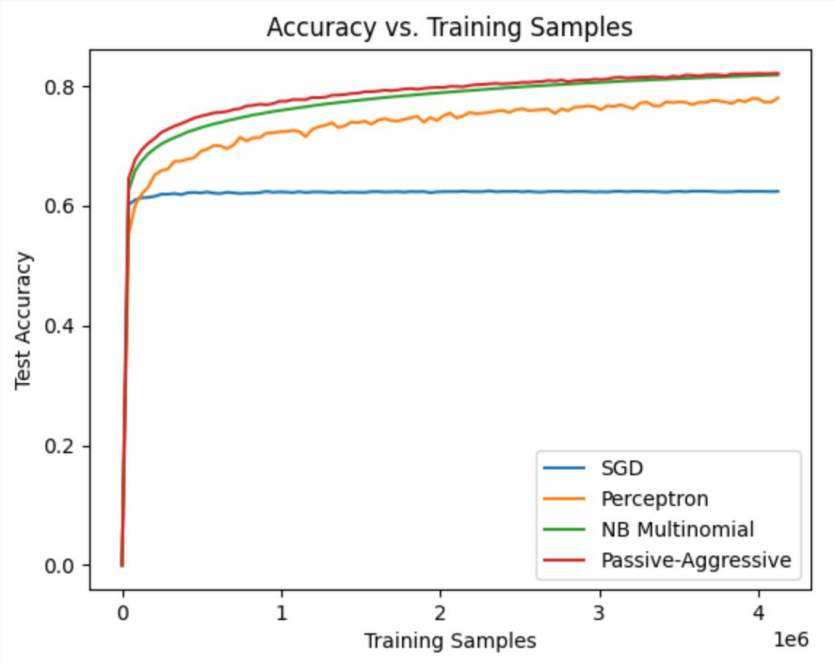
Methods

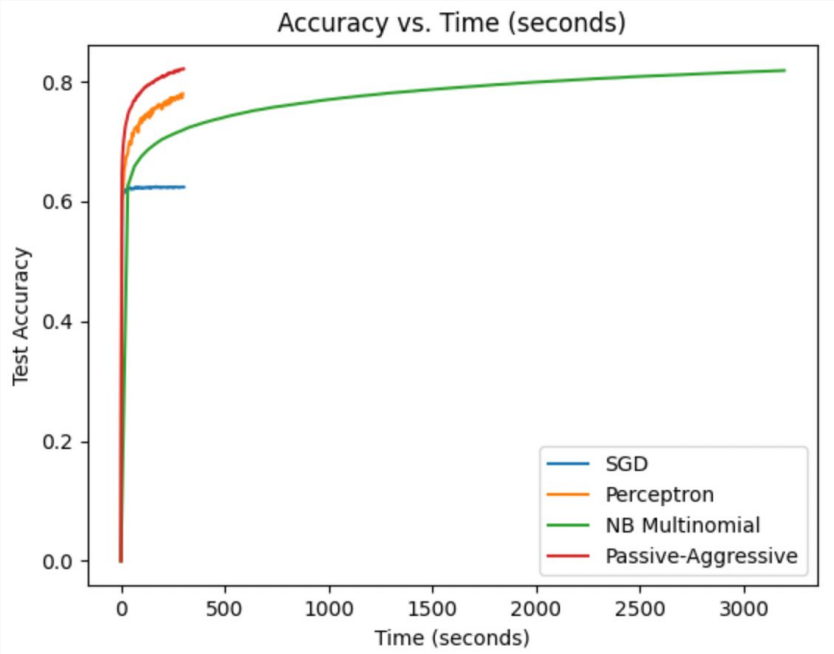
- Utilized 5% of the training data
  - 4,850,000 snippets
- Random split of 85% training and 15% testing data
- Pre-processing on *training data only* to prevent mapping of vocabularies in test data (cheating)
- Scikit Learn, a Python library for machine learning
  - *CountVectorizer, TfidfTransformer, MultinomalNB, SDGClassifier, Perceptron, PassiveAggressiveClassifier*
- Computational experiments on “The Cube” - Cornell CAC cluster
- Evaluation by test data accuracy

Results - Test Accuracy

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

Results - Runtime Performance



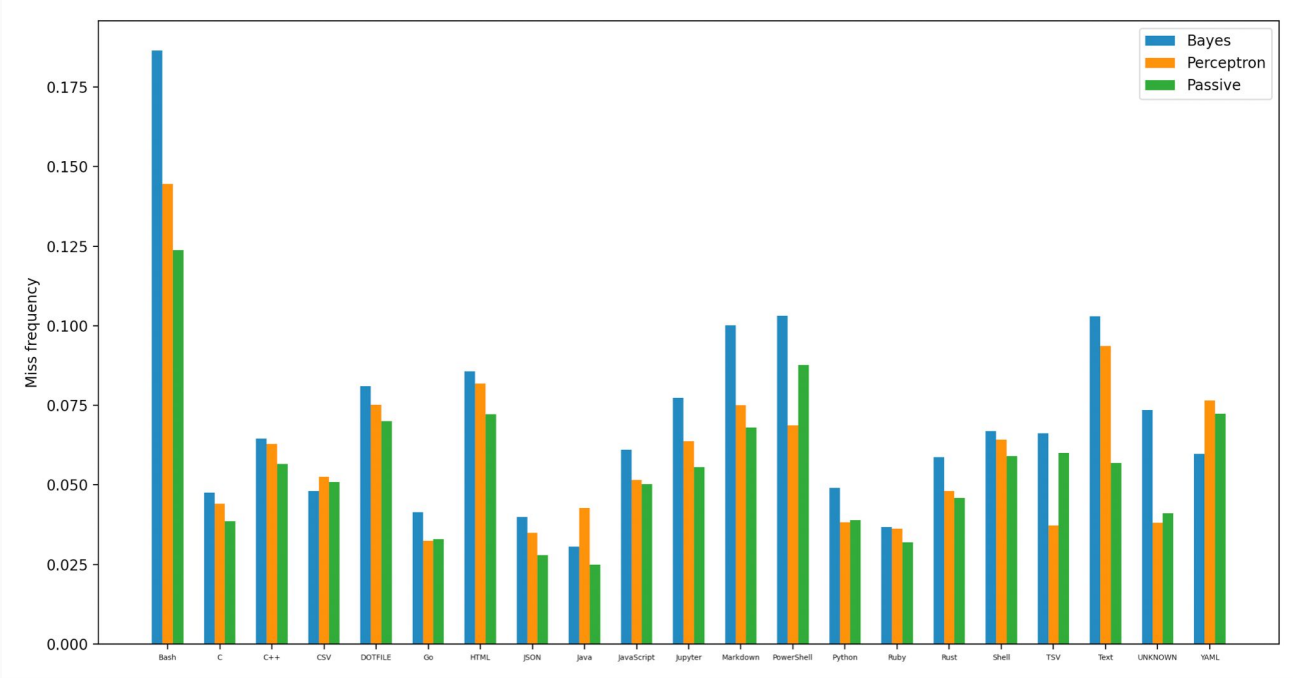


Results - Miss-classifications

- Test models on *entire* dataset

- Calculate `# misses / # snippets` for each language

- Higher miss frequency for languages with less representation in the dataset?

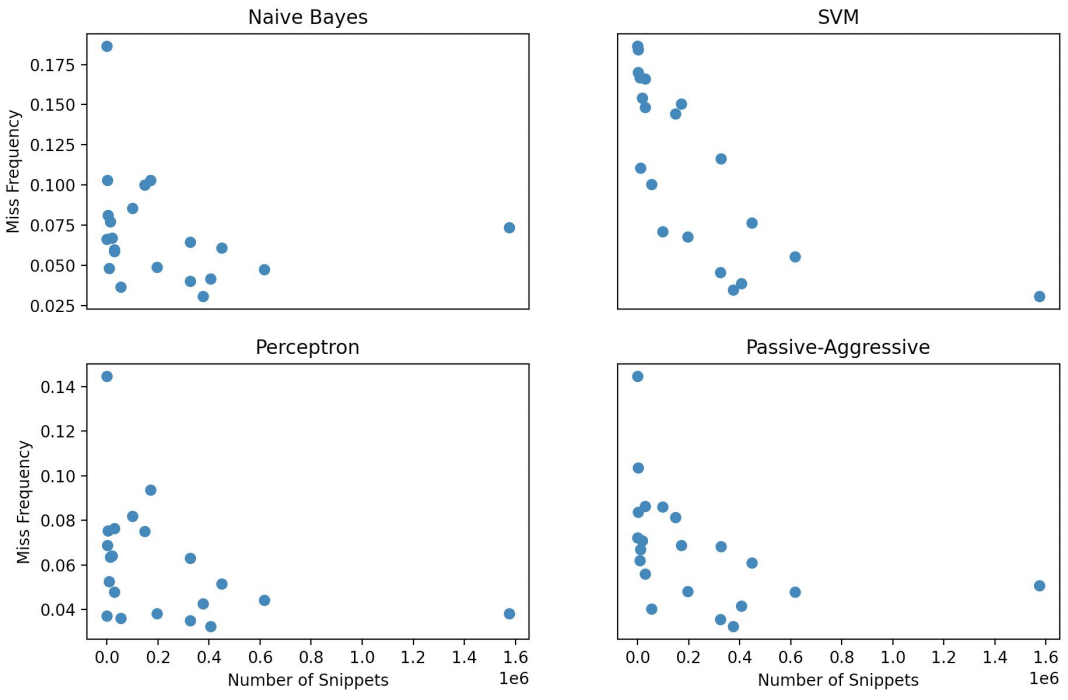


Results - Miss Frequency

- General inverse relationship between number of snippets and miss frequency

- Consistent with our formulation: we are optimizing for accuracy, not *weighted* accuracy

- This aligns with our end user's preferences: popular languages will be chosen more often



Data Bias

- Slight inaccuracies found in the data
  - Difficult to tell how this bias will affect our model is affected

Ex/ *C++* snippet labeled incorrectly

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

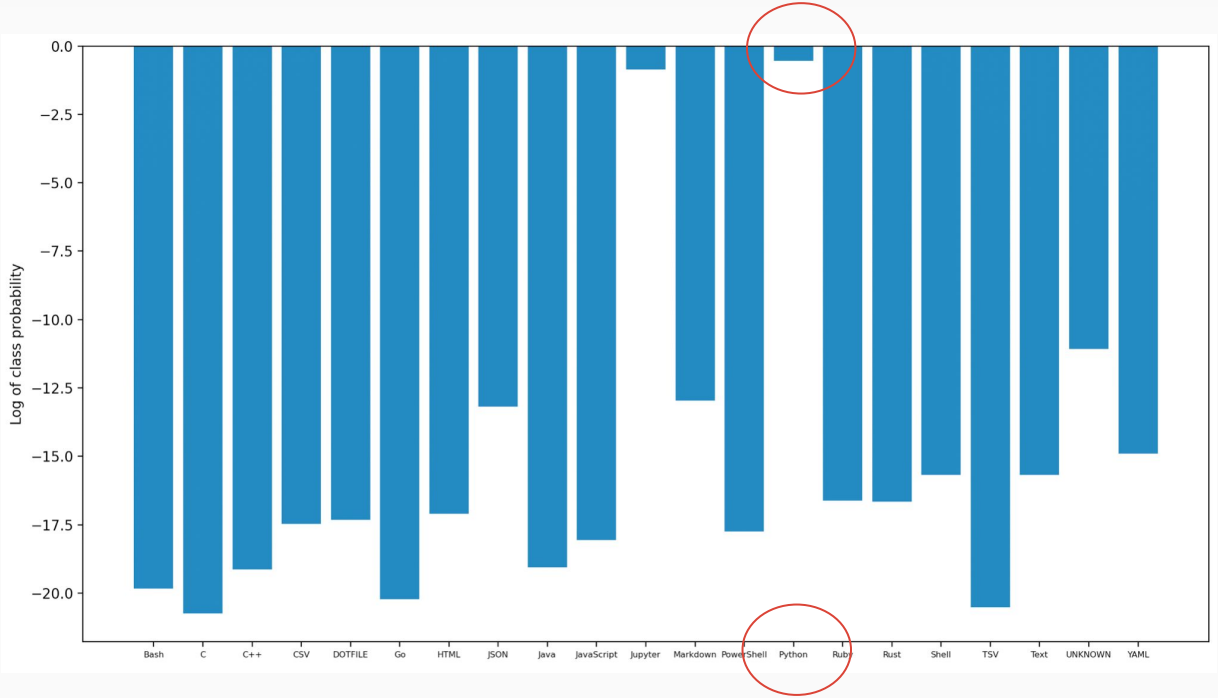
Dataset Label

Example Output

```
from sklearn.naive_bayes import MultinomialNB, GaussianNB

from sklearn.linear_model import SGDClassifier

clf = MultinomialNB(alpha=0.001)
```



Model prediction:

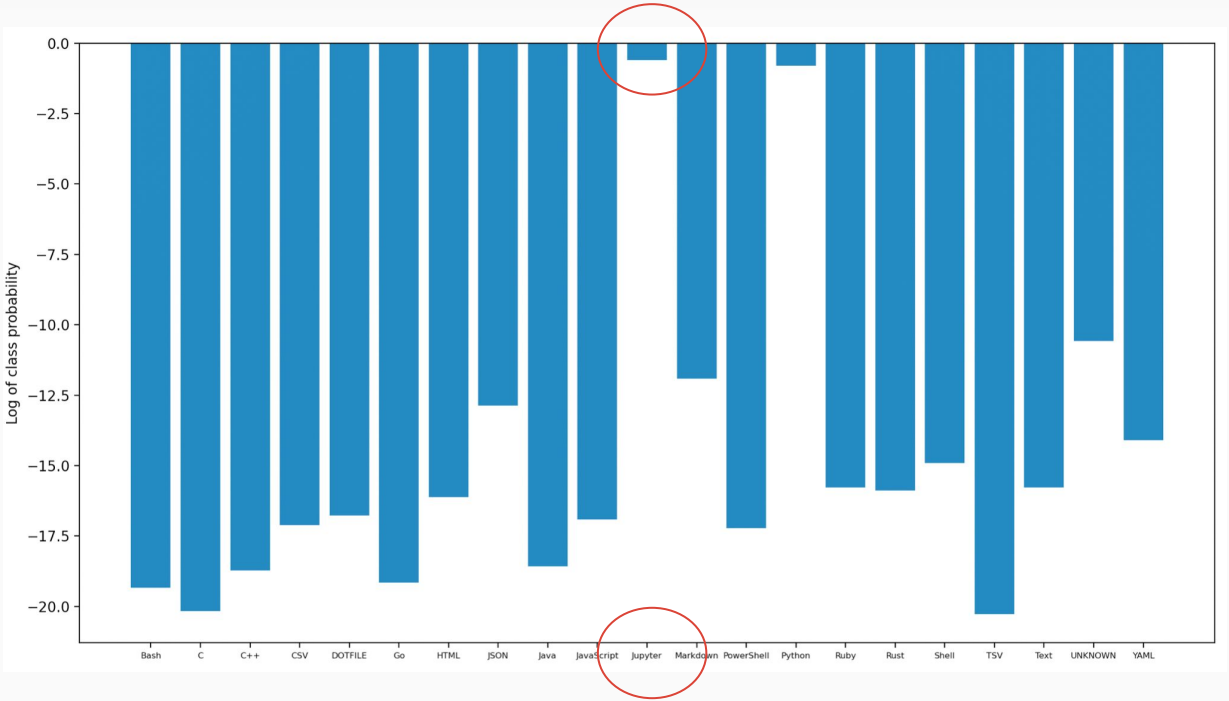
Python

Example Output

```
from sklearn.naive_bayes import MultinomialNB, GaussianNB

from sklearn.linear_model import SGDClassifier

source = MultinomialNB(alpha=0.001)
```



Model prediction:

Jupyter

Example Output - Analysis

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

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

Changing the variable name (cleverly) to “source” augments the probabilities enough to make *Jupyter* the most likely class

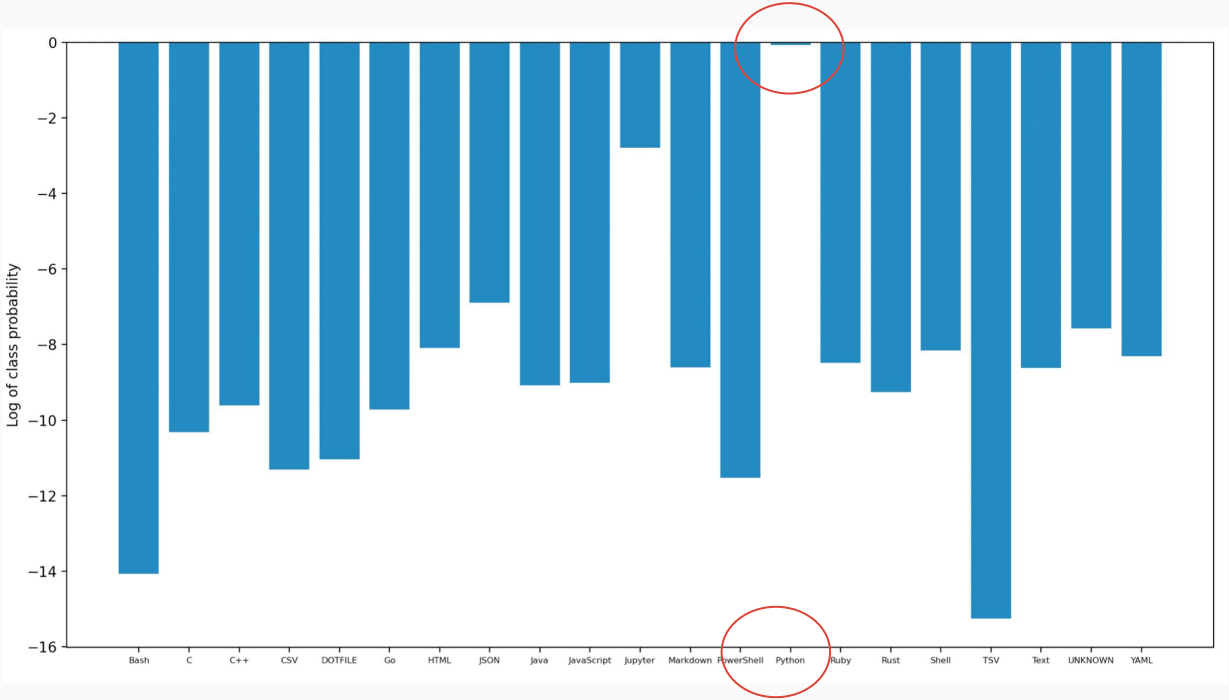
- Jupyter notebooks often contain ML experiments/code (e.g. Python code with sklearn)
- Jupyter notebooks are structured as JSON files, and the “source” key holds the source code of a cell

Example Output

```
from nonsense_module import MultinomialNB, GaussianNB

from nonsense_module import SGDClassifier

source = MultinomialNB(alpha=0.001)
```



Model prediction:

Python

GUI Application

- Ace text editor fork with automated language predictor
- As the user types, we send a POST request to the Python backend, which predicts the programming language with a trained model
- Once predicted, we update the syntax highlighting of the text editor automatically

Demo

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.