



Naive Bayes Document Classifier

By: Matthew McChesney and Denver Quane

Introduction

In a variety of scenarios as Computer and Data Scientists, it can be important to be able to classify large amounts of data without looking at the data and determining these classifications manually: in most instances, this is flat-out impossible. But with the use of Naive Bayes classifiers, and the assistance of MLE and MAP estimates and techniques, it can be a trivial task for even modest or slower computers.

The goal of this project was to implement the Naive Bayes algorithm for document classification. This classifier looks at the frequencies of certain words in numerous documents, and attempts to guess which class the document belongs to, based on classes and word frequencies provided in a training dataset.

In this report, we aim to explore how a Naive Bayes classifier can help to sort documents into their appropriate conceptual topics and genres, purely by using the frequencies of certain words found within these documents. In addition, we will discuss what we learned in terms of programming language considerations, code optimizations, and general knowledge of Naive Bayes document classifying.

High Level Description of Code Functionality

While we initially attempted this project using Python (NumPy, Pandas, SciPy, etc.), we ran into issues where, even using Panda’s optimized “DataFrames” and Sparse Matrices, the program would take upwards of 10 minutes just to organize the input .csv data into a sparse matrix, before even operating on the entries within. By contrast, a simple Java implementation was able to load the same data in less than 60 seconds, and is able to train and classify in less than 3 minutes. Our Java implementation of this algorithm was drastically faster, and easier to rationalize (as both group members are very familiar with Java code, compared to Python) going forward in its development and improvement.

The code starts out by looping through the CSV file, and extracting the data into an array. The array contains twenty rows to store the number of occurrences of each word per class. As we are filling the number of occurrences, we are also filling an array to keep track of how many times each class occurs to be used for calculating the priori. These two steps get rid of redundant looping of the whole dataset, which is instrumental to the fast speed of our program. Our last bit of preprocessing of the data is finding the total number of word instances from each news group.

After our data is stored, we begin calculations to help begin classifying our test data. First, we calculate the Maximum Likelihood Estimate (MLE) by looking at the number of total words per newsgroup. From this we can calculate the conditional probabilities of a given word belonging to a specific news group.

Now that we have calculated the MLEs of the newsgroups, we can begin to classify our testing data. This can be done by reading in the data (which is comprised of word frequencies and document IDs, but no listed newsgroup classification), and analyzing the most likely class that it belongs to -based on the likelihood array that was calculated using the training data- and using our Maximum A Posteriori (MAP) knowledge.

Accuracies Obtained Under Various Settings

With the most basic implementation of the algorithm using MAP, MLE, and priori we obtained an accuracy of approximately 82% using submissions on Kaggle for reference. We attempted to utilize smaller amounts of the training data to try and avoid overfitting, but this did not give us any advantages over the previous accuracies. In one case using half of the training data we lost 4% of our accuracy.

However, we did discover weaknesses and missteps in our algorithm implementation that affected the overall classification accuracy and were able to be modified and improved. For example, we were initially confused in regards to counting the *total* instances of a word per class as opposed to counting the *unique* instances of the word. When fixing this incorrect implementation of Naive Bayes, our accuracy increased by roughly 4% when submitting the Kaggle testing data.

Further improvements were made when ignoring words that occurred only once in a document (indicating that the word is potentially a typo, or at the very least, incredibly uncommon or irrelevant), and by adjusting the MLE likelihoods for words that occur very commonly across the entire training dataset (words with high entropy for MLE). These smaller improvements netted us approximately 3% additional accuracy when testing with Kaggle submissions of the classification testing data.

Below is a table detailing the classification accuracy when training and testing with the same dataset (training.csv):

|  |  |
| --- | --- |
| **Percentage of 12,000 Documents used for Training** | **Accuracy in Classifying Remaining Elements** |
| 10% | 68.611% |
| 25% | 76.788% |
| 50% | 83.083% |
| 75% | 86.0% |
| 100% | 98.875%  (tested on same docs used for Training) |

Our final result, and highest accuracy (87.392%) on Kaggle (testing.csv):https://lh3.googleusercontent.com/S5TulvOUZeP2cDT5J6q1xz5jMC6gE8sQcMZuJYESpi8VPgO7I0W1jhddc7t5NIRnC6fuz5DatDsUMGi8ZhFV3NcBsX6Wl3Npazy4hNStApkLTRgQM3pcAoIsk1PkF4RvO0T9fTeY

And

https://lh3.googleusercontent.com/v_Gz53ba-g3nXfn_e0OKWKxoFhxrQIhmzTrtsj_eMsCADCFD8WoOyXUb3Zfn4yQYX2K0aM8DNGlXol9GQUUppn5MHfw46u_G7ramRh9c5z39G50jka1ULxqxzni_W7YXGinA2pLo

Lessons Learned

The first thing we learned was how to deal with large amounts of data. We initially looked into using Python, but the tools available to read in csv files were far too slow. This is even when considering Python libraries optimized for handling large csv files, such as Pandas and its “dataframes”. In addition to the slow csv parser, Python does not have the fastest for loops. With a total classification/training time of ~20 minutes we decided to move to Java, where we could obtain results in ~2 minutes. Having created the algorithm in these two different languages we learned the bottlenecks of Python.

When trying to learn how to use entropy to increase accuracy, we also learned how to work with datasets to get rid of misleading trends. For example, we observed when training and classifying that words such as “the”, “to”, etc. can be found in almost all of the documents, and thus impacts the efficacy and accuracy when using these words to classify an input to a specific newsgroup. By filtering these types of words (using entropy calculations) from our MLE estimates, we can reduce the impact of these sorts of words (as well as words that are extremely rare or insignificant, such as typos). Further explanation of this process and method can be found in the answer to Question 5.

Conclusion

Throughout the development of this Naive Bayes Document Classifier, our team collectively learned an incredible amount about not only Naive Bayes classifiers themselves, but also the intricacies and complexities involved in extracting greater accuracy and precision from the data used to classify them. From realizing an error in our code that only identified the unique instances of words -and whose revision increased our accuracy almost 4%- to the implementation of more advanced entropy calculation and filtering -which increased accuracy another 3%-, our group was surprised at the improvements that could be made when considering the intrinsic qualities of the data used to classify future data. In addition, the process of answering the provided seven questions greatly enhanced our knowledge of the Naive Bayes’ inner workings, and how we could go in depth to see how the classifier arrived at answers for our input data.