Introduction

For the first portion of the project, we decided to implement Naïve Bayes and kNN within one test driver to make it more convenient during runtime within the command prompt. The command line arguments, including algorithm type (kNN/NB), k number (if applicable), and the relative file path to the folder containing the train and test zip folders (i.e.: C:\Users\chris\Documents\data), were all provided via args [] within the test class ‘classify’. In response to the data received, the algorithm of choice is able to run accordingly, provided that the parameters have been specified correctly to it via constructor. The Naïve Bayes and kNN algorithms both receive input by looping through each of the zip folders, so before the program is ran, the contents of each the zip folders do not need to be extracted to a regular folder, but rather have to be stored within zip folders. Also, note that the directory structure of one of the zip folders (test.zip) was slightly modified to remove an internal file folder (test) which was causing the program to loop through the zip entries twice for a total of 402 entries instead of 201 entries. Therefore, the contents of both zip folders (test and train) consist of the initial spam and ham files and excludes the extra file folder that was present originally within the test zip folder. We have attached the zip folders along with the source code in the project submission to run the program properly according to our implementation.

Naïve Bayes Classifier

For the Naïve Bayes algorithm, we read the training data from the zip folders and transformed it by tokenizing the input (removing special characters and whitespaces), adding spam/ham labels to each email entry, and then appending each email entry to a new file ‘train.txt’. If we wanted to improve the accuracy of the Naïve Bayes classifier further, we could have also removed stop words (words that do not have any significance in evaluation of the Naïve Bayes classifier) during tokenization of the training data. For instance, we could have removed the Subject header from each email, since every email typically has a Subject header. Next, using the ‘train.txt’ file generated from the transformation of the training data, we store the type (ham/spam) and the contents of each email entry.

Looping through each entry, we store each word occurrence in a HashMap (in which the HashMap key is the type(ham/spam) of the current word and the value is an instance of the class Word). If the word occurrence already exists within the HashMap data structure, we return the corresponding word. Then, we examine whether the current word occurrence is of type ‘ham’ or ‘spam’. If the current word occurrence is of type ‘ham’, then we increment the ham count for the word and increment the ham count for the word in all emails. The vice versa also applies to spam words. Once all words in the training set have been accounted for, the probability (P (Word | Ham) or P (Word | Ham) of each word will be calculated with respect to ‘ham’ or ‘spam’. These probabilities will later be added up to calculate the probability of email given spam (P (Email | Spam)) and the probability of email given ham (P (Email | Ham)).

The methodology of reading the test data follows a similar direction to that of the training data, by first transforming the data and appending it to another text file ‘test.txt’. As the case was before with the training data, the test data will also rely on email labels, but this time to count the total number of ham and spam emails. Then, we created an ArrayList to store each of the words from the test data. We call calculateBayes () to compare each of the words from the test data to the unique words found in the training data. This is used to compute the correct classifications and the incorrect classifications, which are ultimately used in measuring the classifier’s accuracy.

Experimental Section

The second task, the experimental portion, was completed using the Sci-kit learn libraries, in which we experimented with Support Vector Machines (SVM) [1] and Random Forest Classifiers (RFC) [2]. The entirety of this section was contained within one file of type .py, where one could choose what algorithm they wanted to employ on the data, as well as the source to where the data was coming from. Though most of the syntax for both algorithms had already been provided for us online (as shown in Works Cited), majority of the methods had needed heavy modification in order to be able to receive a zip file as input and loop through the collection of files within them. The functions, which had consisted of a dictionary builder, and various forms of a feature extractor, were used hand in hand to provide the model something to work with within both algorithms. To accomplish this, the test labels within the SVM condition had to be resized in order to meet the demands of our dataset, as opposed to that which the original authors had used (which was 360 data points in size). In addition, a few libraries which had not made their way into the code before had to be brought to light, for the reason that accuracy\_score and confusion\_matrix (which the tutorials had originally used) were undefined in their own nature.

Once such had been transformed to our liking, we were then able to use the dictionary builder function within the SVM code to receive input from the train directory, and create a train matrix based on the features extracted from those files. In turn, this would allow a linear SVC model to be built, which would prove useful when testing against the features in the test matrix and its associated data points. On the other hand, RFC creates a tuple of both labels and a features matrix, which is then able to store double the amount of information that the previous extract function in SVM would have rendered possible. The resulting variables (which had been instantiated to the extract function built specifically for RFC) were then placed in a model, to be used against the test feature matrix to calculate the algorithm’s overall accuracy.

Efficiency

As far as the efficiency of these algorithms go, we found SVM to be the slowest (followed by RFC), due to the fact that larger datasets typically have a negative impact on how quickly the classification tasks can be performed (having had four ongoing for loops at the same time). The fastest algorithm, Naïve Bayes, was largely owed to it being much more adept to larger bodies of data, as opposed to kNN, which is generally quicker if and only if the dataset is minute in scale.

Results

With regards to how these models print the result, both SVM and RFC display this metric in accuracy score format, with the value being anywhere on an interval from 0 to 100 respectively. Using the built-in round function, the percentage amount was rounded to two decimal points. These results, which were discovered to be consistent across the board, can be found as follows:

|  |  |
| --- | --- |
| *Algorithm* | *Accuracy on the Test Set* |
| *Naïve Bayes* | \_\_\_\_ % |
| *k-NN* | k=1 \_\_\_\_ %  k=3 \_\_\_\_ %  k=5 \_\_\_\_ %  k=19 \_\_\_\_ % |
| *Support Vector Machines* | 98.51% |
| *Random Forest Classifiers* | 95.54% |

Works Cited

[1] KDNuggets, Email Spam Filtering and Implementation with Python and Scikit-Learn

URL: <https://www.kdnuggets.com/2017/03/email-spam-filtering-an-implementation-with-python-and-scikit-learn.html>

[2] Medium, Random Forest Classifier

URL: <https://medium.com/machine-learning-101/chapter-5-random-forest-classifier-56dc7425c3e1>

[3] GitHub, kinejohnsrud

URL: <https://github.com/kinejohnsrud/naive-bayesian-spam-filter>