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 programs properly according to our implementation.

KNN Classifier

To start with, the kNN classifier was implemented through the modification of existing code on the web [1], where we added the ability to loop through zip files, as opposed to the regular folder implementation that had already been present. The directory of the train and test files were provided by means of the test file, where argument 2 was the path. Once received on behalf of the kNNaccuracy() function through the classes’ local constructor, the read() method was then responsible for taking two parameters (created by adding file suffixes to the path argument) as input and counting the Spam and Ham within each of the two zip folders. This technique, though simplistic in nature, allows for the code to generate an idea as to how many files there are within each sub-category, in addition to where they are located in the file structure.

It is also within this domain that each email is transformed, for the purpose of making all whitespace more uniform and also for eliminating any non-alphanumeric/non-numeric characters from the body. The absence of any irregularities allows for the flawless combing of the string generated, where individual words are extracted, and added to a HashMap. Out of this collection, any words that occur less than 100 times are removed from the data structure. As an added detail, the occurrence of each of these words (whether they be in the training or testing sample) are counted and stored within structures of type List to be used in the calculation of Cosine Similarity.

Cosine Similarity, which is where the main classification event takes place, involves each List passed in being compared on an index to index basis, using dot product as the numerator and the length of both List 1 and List 2 as the denominator. Even though KNN’s use within this function can appear to be somewhat brief, its use extends to finding the greatest values in ‘finalList’, which can then be used accordingly to update the accuracy that will be needed later on. This is important, for the reason that this metric is later sent back to getAccuracy(), where the percentage value is then calculated/printed in a fashion that can be viewed by the user (via console window).

Naïve Bayes Classifier

The Naïve Bayes algorithm also involved actively seeking out repositories that were similar in nature, in which we decided to pick off from where Kine Johnsrud [2] had left off. To innovate upon his methodology, we altered his algorithm to read the training data from the zip folders (as opposed to regular folders),where we were then able to transform it by tokenizing the input (removing special characters and whitespaces), add spam/ham labels to each email entry, and then append each email entry to a new file ‘train.txt’. If we had 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) [3] and Random Forest Classifiers (RFC) [4]. 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 a few other methods (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* | 0% |
| *k-NN* | k=1 94.03%  k=3 96.02%  k=5 96.52%  k=19 97.01% |
| *Support Vector Machines* | C=2.0, iterations=1000 98.51%  C=3.0, iterations=1500 98.51%  C=5.0, iterations=2000 98.51%  C=10.0, iterations=3000 98.51% |
| *Random Forest Classifiers* | n-estimate=100, max-depth=2 92.59%  n-estimate=90, max-depth=3 96.30%  n-estimate=80, max-depth=4 98.52%  n-estimate=70, max-depth=5 95.56% |

Topping out at around 99%, Support Vector Machines appeared to be the most consistent in predicting the test files through the model given, coming very close to kNN’s prediction rate at higher ‘k’ values. The Random Forest Classifier methodology also proved to be effective in its own way, classifying up to 98.52% of the test results correctly to their respective categories of Ham or Spam. This metric was determined using the n-estimate, and the max depth of the tree to which their values yielded varied results (as shown above). Though kNN’s accuracy can also vary with regards to how high the ‘k’ value is specified in the parameters given, kNN is able make a small jump percentage-wise from k=1 to k=15, where the metric can be seen to have moved from 94% to 97%.

Naïve Bayes was unsuccessful in classifying email spam because of difficulties in measuring the probabilities precisely. While debugging the probabilities of each word, the error was not in calculating the m-estimate (using Laplace Smoothing) of each word, but because the logarithmic function doesn’t tolerate values close to or equivalent to 0, so it evaluated probabilities to the value infinity. So, to solve this problem, we attempted to use BigDecimal (which is a data type that is used in cases where high percentage of error cannot be tolerated), but it was not compatible with addition operators to sum up the probabilities. We feel like we understood the Naïve Bayes algorithm well from in-depth research, but unfortunately were not able to calculate the probabilities of ham/spam given each email due to this setback.

References

[1] Junshuai Feng, EmailSpamChecker, GitHub,

URL: <https://github.com/JunshuaiFeng/EmailSpamChecker/blob/master/KNN.java>

[2] Kine Johnsrud, Naïve Bayesian Spam Filter, GitHub,

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

[3] Machine Learning in Action, Email Spam Filtering and Implementation with Python and Scikit-Learn, KDNuggets,

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

[4] Savan Patel, Random Forest Classifier, Medium,

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