Kevin Nguyen
EID: kdn433
CS371r - Project 4

Project 4 Implementation

Project 4 was about text categorization; text are categorized based on various learning algorithms. As a result, this algorithm works similar to a sorting routine to decide if a text is relevant or not. There are four files named TestRocchio.java, Rocchio.java, TestKNN.java, and KNN.java that help implement categorization. It should be noted that NaiveBayes is already implemented and it was used as reference for the implementation. The implementation approach was to partially mimic the TestNaiveBayes.java structure and set up each needed function for Classifers and CVLearningCurves. Next, the train and test data functions were implemented last because those function were the main purpose of the project; everything else was trivial work. However, the goal was also to maintain the efficiency of the algorithm so the program doesn't slow down during computational times. We will also analyze some discussions and insight on the resulting data from the implementation.

The TestRocchio.java and Rocchio.java classes will implement the learning algorithm called Rocchio. TestRocchio.java merely sets up the directory, examples and routines for calling. This class will initiate the Rocchio algorithms. Rocchio.java will generate prototype vectors by summing up the document vectors by category and assigning a test vector to the closest prototype vector. Firstly, the train function is called at the beginning to decide whether normalTrainRocchio or modifiedTrainRocchio functions are called based on the "-neg" flag. Normal Rocchio provides the default approach described above, but modified Rocchio has an extra step that subtracts a document vector from all other prototype categories. Document vector summation is done by iterating in a loop over all example's hashmap vectors and adding them. However, the document vector's latest value per iteration is also normalized by the maximum frequency of a token in that document. Some test vector is then assigned to the closest prototype vector based on cosine similarity. The assigned prototype vector is matched with some category which becomes our predicted class; as a result, a comparison is made on the test example and the predicted data to see if the categories match or not. There is a special condition where an empty training set is given, then the algorithm will randomly choose a class to assign since there's no data to work with. These new java classes utilizes HashMapVector, lists, and hashmaps to hold data for future reference and to make the implementation manageable.

On the other hand, the TestKNN.java and KNN.java classes will implement the learning algorithm called K-Nearest Neighbors. TestKNN.java sets up the directory, example, and routines like TestNaiveBayes.java for calling. It's also the class that kickstarts the KNN algorithms. KNN.java will call InvertedIndex.java to help with the indexing when the train function is called. The test function will reference the indexed documents and to retrieve information from that index. This function takes an example document and calls the retrieval function from InvertedIndex to get the already computed similarity values. It should be noted that the retrieval array is already sorted; as a result, we collect the top “k” data from that list by an iterative loop. An iteration is used to determine which category occurs most frequently in the top k documents; an array, resultList[], is used to represent values per category. Those values in the array are incremented and when finished, at least one of the categories has a max value. As a result, the index of the max value is the predicted class. The retrieved category is the predicted class and a comparison is made with the example document category to see if they match. A boolean value is then returned as true or false. There’s a special condition where an empty training set is given, then the algorithm will randomly choose a class to assign since there’s no data to work with. It should be noted that the KNN.java class utilizes lists, arrays, and hashmap structures to help make the implementations easier.

To run the program, extract the files to a location. In the command line, set up your classpath as needed and cd into the directory with the files. Then run “javac *.java” to compile all java files. Then run either “java ir.classifiers.TestKNN -K *K*” (where K is some value) or “java ir.classifiers.TestRocchio -neg” to run the program. When the algorithms have finished executing, the learning curves will be outputted as a .gplot file to be analyzed on. The following discussion will be addressing the outputted results and to draw out further analysis.

1. **Comparative accuracy of the algorithms at different points on the learning curve for the training data.**

The plots on the graph showed varying results and the average range seems to be around 75% to 85% accuracy over a large training set. The Rocchio algorithm seem to initially start with lower at 75% accuracy, but continues to rise steadily towards 83%. On the other hand, ModifiedRocchio had a much higher accuracy for smaller sets of data. ModifiedRocchio dropped off by a large percentage over larger and longer training sets. The overall training set on ModifiedRocchio over large training data ended up with 76% and it was less than the normal Rocchio version. K-Nearest Neighbors where k is 3 has shown that the smaller data sets yield lower accuracy (lowest) at 70%; however, over large training sets, it became one of the highest accurate algorithms at 86% on average. This data is interesting because that algorithm started the lowest, but quickly rose as training sets got larger. If k is 5 on KNN, then that plot looked similar to Rocchio’s plot. However, it does show some varying alterations where one or the other is more accurate at various training set sizes. On average, KNN with k at 5 shows an accuracy around 81% and rising. NaiveBayes and KNN with k at 1 shows possibly the highest overfitting out of all the curves. In fact, those curves are too far at the top with 96% accuracy and higher. NaiveBayes uses prior data and computes probable distribution to predict accurate results. However, KNN with k at 1 shows the highest accuracy. The reason being is that KNN uses the Inverted Index. As a result, the Inverted Index retrievals are likely to hold the top document with the correct page. The most relevant document from Inverted Index is likely to be present at larger training sets. It could also be that the other plots (not Naive Bayes and KNN with k at 1) are underfitting the expected representation of the graph. However, it should be noted that the majority of the plotted points appear closer together and most of them have a steady increase in accuracy over large sets of training data.

2. **Comparative accuracy of the algorithms at different points on the learning curve for the testing data.**

The majority of the curves show a large improvement in the same general direction of increasing accuracy. Over large test datasets, the accuracies vary between 65% and 80% with steady increases. Rocchio algorithm starts with low accuracy over small test datasets, but rises quickly with larger datasets. As a result, the Rocchio plots become one of the highest with accuracy around 80%. Modified Rocchio looks similar, but seems to overfit in comparison to the Rocchio standard because over larger test datasets, Modified Rocchio steadily decreased to be around 72% accurate. And Modified Rocchio still ended up with lower accuracy than compared to normal Rocchio. Interestingly, the KNN starts with lower accuracy 0% with few datasets. However, the KNN plots rose quickly after a few datasets. As a result, KNN quickly rose to 50% accuracy and better after a few sets of test data. Initially, the KNN algorithm does better than Rocchio; however, as test sets get larger Rocchio and ModifiedRocchio are accurate. KNN could have been under fitted since the Inverted Index may have been unreliable. KNN with k at 5, appear to be the better fit with an accuracy around 78% out of the other KNN. However, it should be noted that KNN at k at 5 will eventually decrease and be around the other KNN plots. KNN with k at 3 and k at 1 appear to be the closest plots together. Both of them alters slightly over large datasets; however, the accuracy is around 65% and it’s the closest to being the most inaccurate out of the rest of the algorithms. The NaiveBayes had the highest initial accuracy and that accuracy continued rising over 73% over larger test datasets. However, Rocchio was shown to be the highest accurate with NaiveBayes as the second most accurate. These plots varying between each other, but all points have shown some improvements to accuracy over larger test datasets. Overall, Rocchio seems to give better results on larger data, but KNN gives better results for initial data sets.

3. **Comparative running times of the algorithms in training and testing phases. Include a summary table of training and testing times for each algorithm, as reported by CVLearningCurve.**

The UTCS machines were used to run the program.

- Rocchio Running running times:

```
Total Training time in seconds: 1.096
Testing time per example in milliseconds: 0.51
```

- Modified Rocchio running times:

```
Total Training time in seconds: 2.605
Testing time per example in milliseconds: 1.44
```

- <u>K-Nearest Neighbor, k = 1, running times:</u>

```
    Total Training time in seconds: 1.86
    Testing time per example in milliseconds: 0.17
```

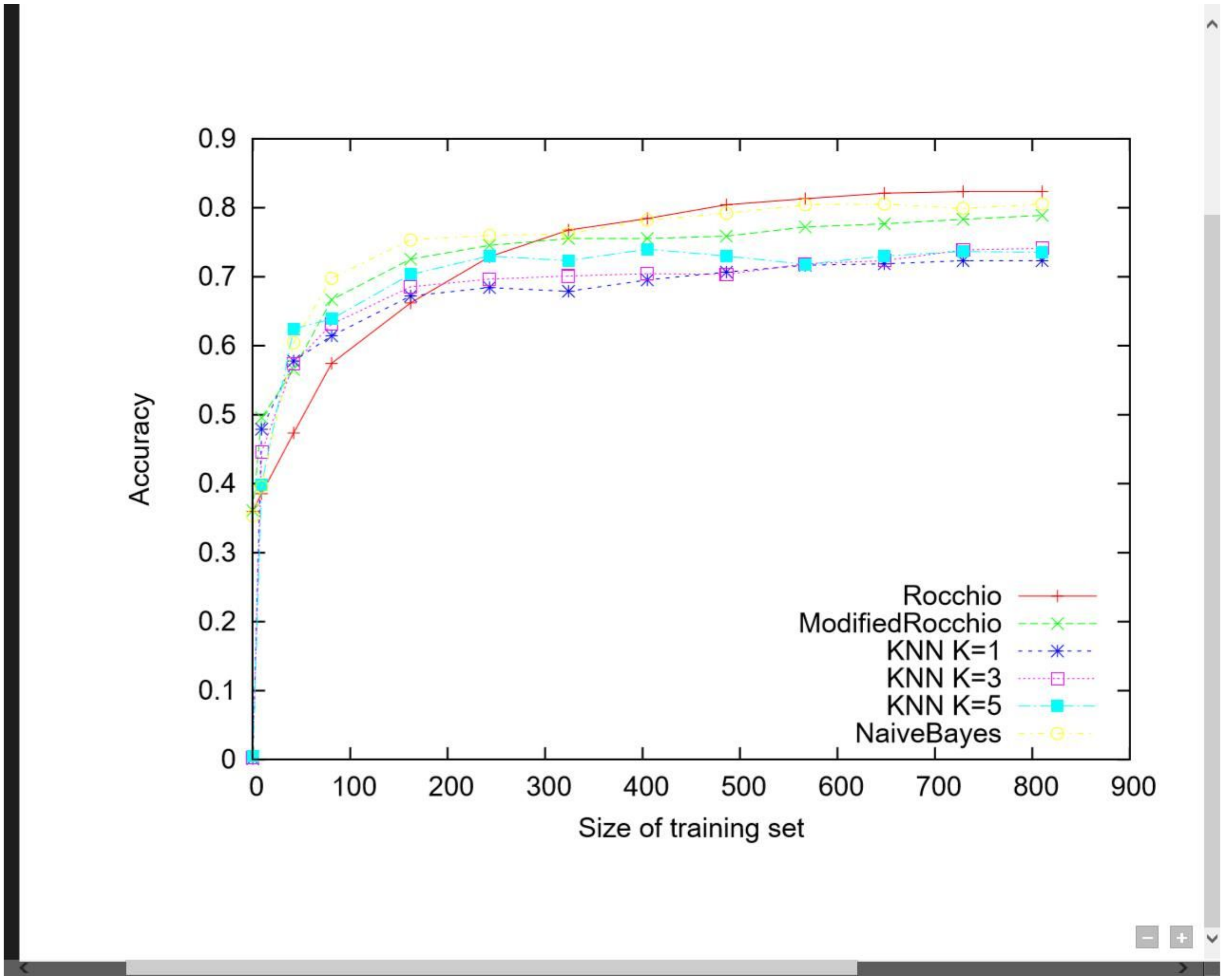
- <u>K-Nearest Neighbor, k = 3, running times:</u>

```
    Total Training time in seconds: 1.851
    Testing time per example in milliseconds: 0.18
```

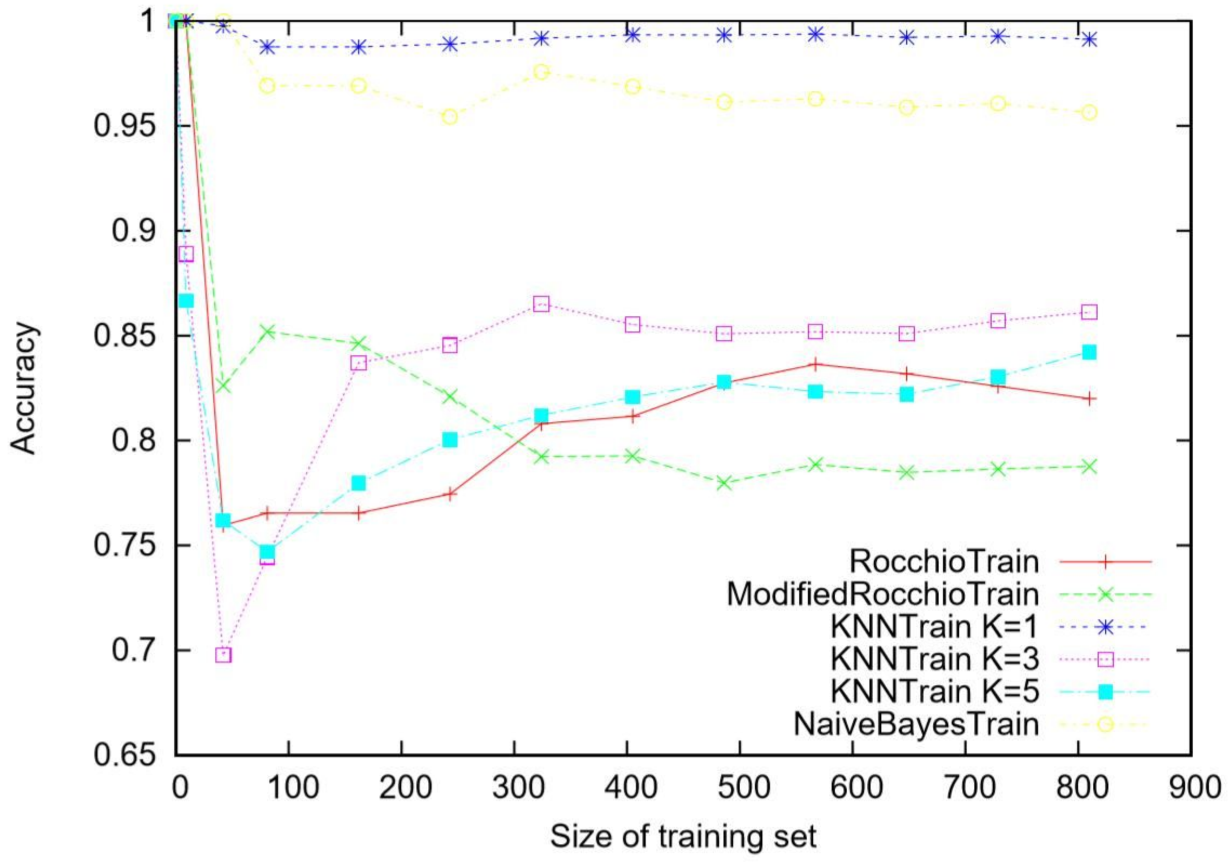
- <u>K-Nearest Neighbor, k = 5, running times:</u>

```
    Total Training time in seconds: 1.833
    Testing time per example in milliseconds: 0.17
```

<u>TEST DATA</u>



TRAINING DATA


Accuracy
Size of training set
RocchioTrain
ModifiedRocchioTrain
KNNTrain K=1
KNNTrain K=3
KNNTrain K=5
NaiveBayesTrain