CSE 5243 Lab 2 Report

Roee Ebenstein, Arathi Mani

Unix Pointer to Files

/home/1/ebenstei/5243/hw2

Work Split

We teamed together after Roee's teammate dropped the class, so we had to choose which first homework to use. We concluded that since there is a convenient library for doing the things we want to in Java, we should rewrite the code we wrote for the first homework in Java, and then to implement the second homework.
Roee did the migration work, and created the file in the format required.
Arathi did the KNN and Naive Bayes classifier (using the Weka library).
We both did code review and debugging.
We both worked on the same code simultaneously, and we used GitHub as a source control.
The report was written together.
It is important to mention that although only one person programmed each module - the ideas behind of them are made after mutual discussions.

Lab 1 Improvements

As mentioned, the code for HW1 was written again, in Java (not including the migration of SGM's to XML's from Roee's first assignment, which was left as is in Python. The Java code creates two vectors - Word count, and Word list. In the word count - we added screening words by the linux English dictionary (in addition to the stemming and stop words, which are

now being done by using Lucene - because of the platform change).
In addition, we added statistical reduction of the number of words using the mean and variance of the word counts (we are using the same statistics for both of the vectors).
We currently do it as if the distribution of words is normal, we'd might change it in the future to fit the actual distribution better.

File Format

Because of Weka requirements - the file format have been changed.
The word count format will look like that:
List,Of,Words.
DocumentId,word count(0,0,1,0..),Topic
There can be repetitions for documents with multiple topics.
The word list format will look like that:
DocumentId,word1,word2,...,Topic

Library

We used the Weka library to build the KNN and Naive Bayes classifier. Originally, the code for Lab 1 was built using Python, but we chose to switch to Java in order to use the existing Weka library instead of implementing fresh (and possibly buggy) versions of the classifiers. This decision was proven to be a good one.

K-Nearest Neighbor Classifier Implementation

Most of the code that perform the training and testing of the KNN Model lies in KNNPrediction.java. The CSV files that are outputted by the WordCountVector.java and WordListVector.java are inputted into a DataSource Weka object. This object extracts each row of the CSV file as an Instance object and the last attribute (last column of the CSV file) is set to be the class attribute. These Instances are then split up into a training set and testing set based on a percentage that is specified as a member variable. The LinearNNSearch object is used as the model that accepts the training set and finds the N nearest Instances to each of the target Instances from the testing set. We used the KNN model to find the top 3 nearest documents and aggregated the topics from these closest documents to output the predicted topics. We determined correctness of classification by doing a set intersection between the predicted topics and the actual topics of the document.

Naive Bayes Classifier Implementation

Most of the code that perform the training and testing of the Naive Bayes Model lies in NaiveBayesPrediction.java.

Cost (Estimated)

Time to build the KNN Model: 4.5 seconds for 1000 documents (so about 45 seconds for ~10k documents).
Time to classify tuples for KNN: 11.2 minutes for 1000 documents (so about 1 hour, 52 minutes for ~10k documents).

Time to build the Naive Bayes Model: 39.2 seconds (for about half size so approximately 1 minute to build entire model).
Time to classify all tuples for Naive Bayes: 9.44 minutes (for about half size so approximately 19 minutes to classify the entirety).

Results

K-Nearest Neighbor

We did 3 different types of evaluations for KNN. The first measured the number of documents that had at least one topics predicted correctly. The second measured the number of documents that were perfectly predicted. The final measurement said what percentage of the topics were erroneously categorized. However, these measurements are based off of 1000 documents since running nearly 10k documents took far too long. We would expect the percentages to be significantly higher because of the larger corpus of topics to predict from.

60-40 Split

41.5% accuracy for correct classification.
None of the documents were classified perfectly.
85% of topics did not match.

80-20 Split

44% accuracy for correct classification.
None of the documents were classified perfectly.
84% of topics did not match.

Naive Bayes

The Weka implementation of Naive Bayes used more memory than available on RAM so in order to accommodate this, the classification algorithm was run on smaller sets with ~5k documents. The results were then extrapolated to predict the accuracy for ~10,000 documents. The results were also lower than expected because Weka does not handle multi-class classification. This meant that topics had to be perfectly classified into one topic and therefore, some tuples would never be classified correctly. This skewed the results down.

60-40 Split

49.5% accuracy.

80-20 Split

45.7% accuracy.

Future Improvements

In the future, Naive Bayes might have to be re-implemented using our own algorithm since Weka does not handle multi dimensional classes well. The low accuracy might have been partly attributed to this fact.

Other areas for improvement could be determining more methods to evaluate whether an object was classified correctly or not. For Naive Bayes, the method provided (the evaluateModel method of the Evaluation) may not have been the ideal evaluator since the Weka Naive Bayes implementation was not designed for multi-classes.

For KNN, there were multiple ways to determine correctness (i.e. whether the topics perfectly matched the actual topics, whether at least one of the topics predicted was included in the actual topics, or whether a percentage, maybe 50%, of the predicted topics matched the actual topics). We chose the most lenient of the three evaluations which was to determine if there was at least one topic that matched the actual topic list. The other methods could be tested in the future to determine whether they have a better evaluation rate.

All of that assumes we didn't remove words that are important, a concept which we will probably have to tackle in the future.

Finally, the time complexity of the implemented KNN model was extremely high. For each entity in the testing set, the Euclidean distance was calculated to every entity of the training set. The Euclidean distance took into account every single attribute (i.e. about 3k). This resulted in an extremely slow algorithm for predicting the topics of the testing set, about Big-O of (n*m*b) where n is the size of the testing set, m is the size of the training set, and b is number of attributes. For the future, this running time should be faster, by perhaps reducing the number of attributes (dimensions) of the prediction or something similar.

References

- http://weka.wikispaces.com/Programmatic+Use
- http://weka.8497.n7.nabble.com/multiclass-classification-td23322.html
- http://weka.sourceforge.net/doc.dev/weka/classifiers/Evaluation.html
- http://weka.wikispaces.com/file/view/IncrementalClassifier.java/82917091/IncrementalClassifier.java
- http://crdd.osdd.net/man/wiki/weka/core/converters/ConverterUtils.DataSource.html
- http://weka.sourceforge.net/doc.dev/weka/core/neighboursearch/LinearNNSearch.html
- http://weka.sourceforge.net/doc.dev/weka/core/Instances.html
- http://stackoverflow.com/questions/6554796/how-to-actually-get-the-nearest-neighbor-in-weka-using-java
- http://ianma.wordpress.com/2010/01/16/weka-with-java-eclipse-getting-started/
- http://stackoverflow.com/questions/16034008/using-weka-for-k-nearest-neighbors-search