Mark Bryk
Robert Gruener
Elliot Rappaport

Hadoop Text Categorization

For our Cloud Computing final project, we decided to use hadoop in order to exploit the parallelization of featuring text and the algorithm K-Nearest Neighbors by performing text categorization. Therefore given a training set of news articles, each having a specific category such as politics, crime, or disaster, a set of test articles can be classified within one of the categories. The motivation behind using hadoop with text categorization is to greatly speed up the computationally intensive process of featurizing large amounts of text, as well as performing the K-Nearest Neighbors (KNN) algorithm in parallel to actually classify text categories. Our source code can be found at https://github.com/mbryk/HadoopTextCategorization

The project utilizes maven for building as well as running the source. The details on running the project can be found in the github's readme. The project allows to run the text categorization on a set of training and testing articles, or only perform either training or testing separately in order to allow for more flexibility when testing on different data sets. The goal of the different run options is to reduce unnecessary computation caused by re-running mapreduce jobs on identical data.

Within the source code, the featurizing and categorizing are performed in two separate mapreduce jobs which are chained together when necessary. Setting up these jobs can be found in the Main.java file. The featurizing job takes a directory of training articles as input and performs a word count on each article. The mapping class can be found in MapClassWC.java which maps the value of one to the key of the article name as well as the word within that article number. This allows the same words in different articles to be separated properly. The reducer for this featurizing job is simply the built-in IntSumReducer.

The other mapreduce job actually performs KNN and outputs the K nearest training articles for every test article. The mapper can be found in the file MapClassKNN.java which takes the output of the word count job as input. The input is formatted as a string containing the counts for all words within a single training article. Therefore each mapper gets a different subset of the total number of training feature vectors. In order to perform KNN, each mapper needs to compare every single test feature vector to each training feature vector. Therefore since the mappers receive subsets of the training feature vectors, every mapper will need access to all test feature vectors. We accomplish this using job configurations to send the testing feature vectors to every mapper. This is far from an ideal solution as it loads all testing features in memory. However, as the number of testing articles should always be significantly less than the number of training articles, it should not be a large issue. In the future, using a flat HDFS file or database for the testing features would be largely preferred. The KNN mapper then performs a similarity calculation on each combination of test feature vector and training feature vector. This similarity calculation is basically a euclidian distance where each unique word is a separate dimension. It can also be seen as a cross product normalized to the number of dimensions. Then the testing article name becomes the key, with the similarity value to that mapper's training article as the value. Upon the reduce, each test case has access to the full list of similarities to every training case. The reducer (found in ReduceClassKNN.java) then iterates through these lists for every test case, adding each similarity to a Priority Queue (implemented as a Max Heap). Upon completing the iteration the top 5 elements of the priority queue are returned, essentially setting K=5 for the KNN and giving the 5 matched training article for every test article.

Finally, the list of the matched 5 articles needs to be converted to a single category with which to classify the test article. This is done by cross referencing the 5 returned training article names with a corresponding list of labels, taking the maximum occurrences of a single label in the returned set. Similarity values are used as a tiebreaker.

Although imperfect, the results of our implementation of text categorization were promising for a first attempt. Across two large corpuses of data, our precision was above 70%. This number could be improved by parsing the articles before featurizing, combining words with the same roots, and removing suffixes and prefixes. In our implementation, we simply removed any whitespace and punctuation from the articles. Our precision matrices for each corpus are shown:

Corpus 1:

313 CORRECT, 130 INCORRECT, RATIO = 0.706546275395034.

CONTINGENCY TABLE:

	Dis	Pol	Str	Oth	Cri	PREC
Dis	72	1	16	0	0	0.81
Pol	4	82	56	2	0	0.57

```
Str     5     5     124    0     1     0.92
Oth     1     2      13    7     2     0.28
Cri     3     3      15    1    28     0.56
RECALL  0.85   0.88   0.55   0.70   0.90

F_1(Dis) = 0.827586206896552
F_1(Pol) = 0.691983122362869
F_1(Str) = 0.690807799442897
F_1(Oth) = 0.4
F_1(Cri) = 0.691358024691358
```

Corpus 2:

```
277 CORRECT, 82 INCORRECT, RATIO = 0.771587743732591.

CONTINGENCY TABLE:
        O      I      PREC
O      222    27     0.89
I       55    55     0.50
RECALL  0.80   0.67

F_1(O) = 0.844106463878327
F_1(I) = 0.572916666666667
```

With only two categories, our precision improved to 77%. As is expected with KNN, precision is higher in categorizing the articles from the more populous category, since more training articles are available as comparison.