A Naive Bayes Classifier was implemented and run in several configurations on the Cornell Labeled Movie review data set. The data-set was preprocessed in several ways. Several different collections of stop words were tested including the google and MySQL collections. Additionally, the Porter stemming algorithm was used to normalize the word vocabulary. F1 scores were used to judge the efficacy of each implementation. True positive hits measured the number of positive reviews returned by the query. In effect, this makes the F1 score measure the results of a search for positive reviews simulating a commonly desired result. Differentiating words were determined by finding terms with the largest difference in occurrence rates in each class.

The baseline implementation uses a binomial model with no stemming or use of stop-words and a smoothing value of 1. This seems like an appropriate baseline as the binomial model loses the most information, not removing stop-words maintains the most irrelevant words, stemming treats identical concepts as different words, and a smoothing of 1 undermines the impact of infrequently occurring terms which might be strongly associated with one classification or another.

Ideally, in order to test for the effectiveness of a technique like using stop-words or Porter stemming multiple tests would be run on different arrangements of the dataset or more data, but only one cross-validation partitioning of the data set was used.

**Evaluation**

**Baseline**
In the baseline case, differentiating words, words which tend to appear more in one class than in another, oftentimes looked to be a result of over-fitting. At least, differentiating words don’t seem to be usual means to communicate valence. For example, the movie “The 13th Warrior” was almost universally panned and most articles commented on the lead character’s role as a viking so therefore *viking* became a term heavily associated with negative reviews even though that was only particular to this data set. Interestingly, enough the phrase *mpaa* was significant as it was usually raised in the context of a movie being controversial and thus interesting. However, other terms came out to be relevant with clear meaning such as *uncompromising* as positive and *preposterous* as negative indicators of valence. Filtering out unhelpful proper nouns based on their effective communication of valence from a separate set data might be a worthwhile strategy to avoid the impact of nouns like *Elmore*, an author of a book which inspired a positively reviewed film. On the other hand, a large data-set would be less likely to suffer from these problems unless movies with vikings tend to be terrible in general and movies based off of books by Elmore are good in general. The most important takeaway is that, for the most part, we aren’t learning about which words are most indicative of valence, but which words are correlated with it.
Differentiating Words Baseline 10-Fold Cross Verification

**INSERT TABLE**

**Improving The Baseline**
Since the most differentiating words were not stop-words I doubt that using them will benefit the reviews all that much. However, stemming words has a chance at being helpful as few of the differentiating words are common carries of valence. Perhaps, stemming will cause different forms of powerful adjectives to surface this may also be the case with using a smaller smoothing value.

**Stop-Words**
The baseline configuration was used with several different sets of stop-words including the set used by Google, MySQL, and two lists from the Ranks.NL Article Analyzer. Contrary to our initial hypothesis the stop-words improved the results. However, none of the sets showed a significant advantage over any other.

<table>
<thead>
<tr>
<th>List</th>
<th>Google</th>
<th>MySql</th>
<th>RankNL Small</th>
<th>RankNL Large</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg F1</td>
<td>.73</td>
<td>.74</td>
<td>.73</td>
<td>.74</td>
<td>.70</td>
</tr>
</tbody>
</table>

**Stemming**
Stemming holding other baseline parameters constant did not provide any noticeable improvement.

<table>
<thead>
<tr>
<th>Stemming</th>
<th>Used</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg F1</td>
<td>.70</td>
<td>.70</td>
</tr>
</tbody>
</table>

**Smoothing Value**
Smoothing holding other baseline parameters constant did provide a measured improvement, but it may not be significant.

<table>
<thead>
<tr>
<th>Value</th>
<th>.1</th>
<th>.25</th>
<th>.5</th>
<th>.9</th>
<th>Baseline (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg F1</td>
<td>.71</td>
<td>.71</td>
<td>.70</td>
<td>.70</td>
<td>.70</td>
</tr>
</tbody>
</table>

**Multinomial**
Using a multinomial model actually was worse than the baseline for the given permutation.

<table>
<thead>
<tr>
<th>Value</th>
<th>Multinomial</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg F1</td>
<td>.68</td>
<td>.70</td>
</tr>
</tbody>
</table>

**Combined Shot in the Dark**
Oddly enough the combined results of using all of the above techniques aggressively, stemming, using the large stop-words list, a very small smoothing factor of .001 and using a multinomial model produced an average F1-score of ~.77 for multiple cross-validated permutations of the input data. Perhaps with the removal of often repeated stop-words the repetition of less common words became relevant which would be been hidden by aggressive smoothing.

**Questions**
Why does the combined set of techniques robustly work better on the given data? Would this hold for other similar sets of reviews? There are several different tuning parameters available, how does one determine the amount of data needed to avoid over-fitting?

**Other Notes**

**Issues With Scala**
The algorithm itself is elegant, but the majority of time was spent wrestling with scala or implementing routines because scala’s esoteric mechanisms made it difficult to use the NLP and LA libraries. Here are some interesting and annoying issues encountered.

Interestingly enough the most difficult aspect of the project was getting used to Scala’s type hierarchy and the quirks of it’s type system. Due to the sparse documentation and lack of examples or indication of purpose, most of the functionality found in Scalala and ScalaNLP wasn’t really understandable until after much of the functionality provided by these libraries was reimplemented. One of the strangest errors encountered occurred in the following snippet of code:

```scala
//Build dictionary and index
var dict = new HashMap[String,Double]()
var binDict = new HashMap[String,Int]()
for (doc:LabeledDocument[Double,String] <- corpus.toIndexedSeq) {
  for ( i <- 0 until (doc.fields.get(featureBody).size)) {
    val ss = doc.fields.get(featureBody).get(i)
dict.put(ss,dict.getOrElse(ss,0.0)+1.0)
```
binDict.put(ss,1)
}
}

If the bold numbers are changed to integers the getOrElse method is inferred to be have String as its parameterized type.

Another lovely problem involved the Scala type hierarchy. Since immutable and mutable Maps have the same name compile time type, errors would result from using mutable.Map implementations in place of immutable.Map objects except that only the class name would be used and not the fully qualified class name. Therefore, you get an error indicating that a Map cannot be used in place of a Map.

Resources

Data Sets
Labeled Movie Review Data: http://www.cs.cornell.edu/people/pabo/movie-review-data/

Stop Words
- MySQL stop words: http://www.koders.com/c/
   fid6C772BFE1925D9797AD84213004ABB198FC41069.aspx?s=crc
- Porter Stemming Algorithm: http://tartarus.org martin/PorterStemmer/scala.txt
- RankNL StopWords: http://www.ranks.nl/resources/stopwords.html
- Google StopWords: http://www.ranks.nl/resources/stopwords.html