Modeling Sentiment with Ridge Regression
Luke Segars || 2/20/2012

The goal of this project was to generate a linear sentiment model for classifying Amazon book reviews according to their star rank. More generally, classification accuracy is also measured based on the model's ability to predict whether a review is “positive” (4 or 5 stars) or “negative” (1 or 2 stars). A dataset of 495,797 unique reviews were used for all experiments.

Data Preparation

After downloading the tokenized data file, a number of steps were necessary to prepare the data for regression. The output generated a 495,797 x 10,000 matrix that was used for the modeling and experiments detailed in future sections.

De-duplication

Matlab's unique function was used for de-duplicating reviews – each row that appeared more than once was reduced to a single occurrence. This technique can produce some false positives in the case that the same word frequencies occurred in legitimately distinct reviews (since the order of the tokens is not considered), but the chances of this occurring was considered improbable enough to avoid more complicated approaches.

The original dataset had 975,194 reviews and the de-duplicated dataset contained 495,797 reviews.

Feature selection

Due to resource limitations, a subset of the available features had to be selected for inclusion in the model. I chose to use the top 10,000 tokens (features) when ranked in order of frequency of appearance over the full data set; a larger subset of the tokens would have been used if more system memory was available. This could be a path to explore in future work.

Stopwords

A second data matrix was generated that reported zero counts for all of the words in the dataset-provided stopwords.txt file. The columns were not removed in order to keep the column ID => keyword mapping intact so that the results could later be reinterpreted.

Normalization

Each review (row) was normalized so that all values sum to 1 in order to control for different review lengths. Each feature therefore became an indicator of the percentage of a review that an individual word accounted for.

General Model

The model I chose to implement is a ridge regression model with a gradient-optimized lambda value. The ridge regression model helps avoid some of the troubles that the least squares approach encounters (namely the presence of singular matrices) and is able to achieve decent predictive performance. The
The equation that ridge regression uses to compute its beta model is:

\[ \hat{\beta} = (X^T X + \lambda I)^{-1} X^T y \]

The lambda value is somewhat arbitrary but can be optimized for a particular problem. I used Matlab's `fminunc` function to optimize the lambda value using a gradient descent algorithm with the goal of minimizing the RMSE rate. Graphs of the optimization process are included in the next section but the RMSE rate seemed to remain relatively stable with a range of lambdas.

**Performance Results**

The classifier was tested against a test set of approximately 50,000 reviews that were randomly selected from the full corpus (but were not included in the training set). The selection process was repeated for each iteration meaning that no formal partitions were drawn and maintained throughout the cross-validation process. All results reported below are the averaged values over 10-fold cross-validation.

The same classifier was tested with a number of different input sizes to examine the hypothesis that more data would improve match quality. The performance consistently improved as the size of the dataset was increased, as expected. Each score was confirmed by three independent executions of the 10-fold cross-validation process described above.

*Figure 1: Error rates decreased while AUC and lift scores improved as the size of the input dataset increased. Note the lift score scaling.*
Generating predictors with ridge regression

**RMSE**

The optimal RMSE rate achieved using this predictor was .9232 using a stopword filter and .9380 without the filter.

**Lambda optimization**

Ridge regression required the selection of an offset (“penalty”) weight \(\lambda\). I attempted to optimize the value of \(\lambda\) to minimize RMSE using a gradient descent method but the chosen value seemed to have very little impact on the classification accuracy with the except of negative \(\lambda\)s which had a significant negative impact on the classification scores.

Several \(\lambda\) values were lower than the optimal value that the gradient descent method converged on. I suspect that this was because the optimization process was based on the results of a cross-validation experiment that used random selection to seed its experiment & test groups and the lower scores were simply outliers that couldn't be repeated by later attempts.

![Lambda Optimization](image)

*Figure 2: Attempts to optimize the \(\lambda\) value had very little impact on the error rate.*

**ROC / AUC**

The classifier was able to perform quite well over the course of the full experiment. The average 1% lift scores for the full review sample were 81.65 (no stopword filter) and 82.54 (stopword filter), and the average AUC's were .931 (no stopword filter) and .934 (stopword filter).
The stopword filter improved performance in all experiments, although the effect was relatively small in each case. This is an expected behavior because it indicates that the stopwords don’t have any significance for partitioning the datasets; each review vector was normalized, eliminating the bias that can arise from correlations between longer reviews and the absolute number of stopwords present in the review.

<table>
<thead>
<tr>
<th></th>
<th>1% lift</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>No stopword filter</td>
<td>81.65</td>
<td>.931</td>
</tr>
<tr>
<td>Stopword filter</td>
<td>82.54</td>
<td>.934</td>
</tr>
</tbody>
</table>

*Figure 3: The classifier was able to achieve high accuracy among the reviews that received the most confident rating.*

**Highest / lowest weighted words**

The following words received the highest (positive) and lowest (negative) beta scores.

**Positive**

<table>
<thead>
<tr>
<th>Mens</th>
<th>Taking</th>
<th>Quiet</th>
<th>Related</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments</td>
<td>Agree</td>
<td>Tree</td>
<td>Processes</td>
</tr>
<tr>
<td>Understand</td>
<td>Myths</td>
<td>Italian</td>
<td>Murder</td>
</tr>
<tr>
<td>Adults</td>
<td>Tom</td>
<td>Themes</td>
<td>Reality</td>
</tr>
</tbody>
</table>
This is a somewhat ambiguous set of terms that seem to be relatively uncorrelated with standard use of the language. Some positive words ("agree", "understand") and some negative words ("criticism", "sad") appear in the respective sets, but the overarching themes seem to be less clear. Each category contains some words that could be topical, and interestingly the word “mens” appears in the positive set while “womens” appears in the negative set. This could be based on a male gender bias in the distribution of the reviewers who could use the term “womens” in a negative manner.

**Computation Speed**

Due to a lack of knowledge about Matlab's particular implementation of matrix operations it was only possible to do a very rough estimate of the number of floating point operations that my machine could handle. I ran all experiments on a Core 2 Quad 2.4 Ghz and the entire sparse matrix could fit in memory at once. Matlab utilized all cores for the matrix multiplication operations.

The approximate computation rate of the operations was around 1.74 gigaflops based on my rough estimates. The time required to compute the core matrix multiplication was 57.05 seconds, and I estimated the number of operations based on the average number of words per record (86.1) and the assumption that computing each beta value took $2n$ operations (one per element for multiplication and for adding to other products), where $n$ is the length of an average row in the matrix.

I suspect that this value is an underestimation since there is probably additional functional overhead required for the matrix multiplication.