1 Introduction

In this project we implement a linear regression model in the context of sentiment analysis for online product reviews. Given a set of review articles, we extract the words from the review texts as features, and train a linear regression model to predict the numerical rating for each review.

We used the tokenized data from the JHU Amazon book review dataset [1] and computed the exact solution of the ridge regression model with $L_2$ error function. We adopted the bag-of-word model of text, and use the terms as features. Moreover, we experimented with different number of features, regularizer weights, stemming, and stop-word removal. The models are evaluated using the average AUC and 1%-lift scores of 10-fold cross-validation. We used Matlab for our project, and all experiments are conducted on Amazon EC2 instance with 8GB memory running Ubuntu Linux 10.0.4LTS.

In the remaining sections of the report, we will discuss the following aspects of our project.

- Data Preprocessing
- Regression Model
- Feature Selection
- Experiment results

1Source code available at: https://github.com/rxin/mining-sentiment-regression
2 Data Preprocessing

In the data preprocessing stage, we generate the X matrix and y vector for reviews, which is used later in the regression and analysis pipeline. The X matrix is the feature matrix for all reviews, while the y vector consists of the corresponding numeric rating, on a scale of 1 to 5, for each review.

We started with the tokenized.mat file, and applied several techniques to reduce the memory usage, and to improve efficiency.

In order to speed up the processing, we wrote the entire data and model pipeline in Matlab. tokenized.mat data file is read in. The file is a 3 by \( n \) matrix (\( n \) being the number of tokens). The first two rows give information about the line numbers and positions of the tokens, and the third row contains the actual indices of the tokens. Since we view text as bag of words (and only consider unigrams as features), we immediately removed the first two rows as the position information is irrelevant to our model. This way, we reduced the amount of memory required to generate \( X \) and \( y \) by 67%. We are able to separate the reviews and extract the ratings with the help of the corresponding tags.

We then converted the reviews to get the matrix \( X \), and the ratings into the \( y \) vector. The total memory used to generate \( X \) and \( y \) in total is less than 5G. Total number of features (i.e. size of the dictionary) is 1796313.

2.1 Review Deduplication

We noticed that the dataset have a large number of duplicated reviews. as a matter of fact, out of the one million reviews, we found only 488012 distinct reviews. We recognize duplicate reviews by using the first ten words of each review as the hash key, and use a hashmap (containers.Map in matlab) to do the dedup. This considerably drove down the memory usage, and speeded up the model training.

2.2 Stemming

We also tried using the Porter stemming algorithm \(^2\). Stemming is the process of reducing inflected and derived words to their stem \(^2\). Word \( w_{di} \) in our document representation is then the result of applying the stemming function to the \( i \)-th word of the document as it appeared in the input data set. Stemming incidentally reduces the number of features used in the classifier. For example, using the Porter Stemmer \(^2\), both “apply” and “applying” become the same stem “appli”.

\(^2\)Matlab source code modified from http://tartarus.org/martin/PorterStemmer/
2.3 Stop words

We collected a list of English stopwords from

http://www.tomdiethe.com/teaching/remove_stopwords.m

and tried removing stopwords in building our model.

The list has 571 stop words, and 521 of them are in our dictionary. We find out that removing stop words reduced the size of $X$ and $y$ by 50%. This makes sense since $X$ is stored as a sparse matrix, and most documents will have stopwords (non-zero count in stopword features). Removing stopwords essentially removes a large portion of non-zero features.

We presented experiment results of stemming and stop word removal in Section 5. We can see from the ROC plots (using top-11000 tokens) in Figure 1 that default, stemming and stop word removal perform on par with each other. If we look at AUC scores (Figure 2 and 4), we see small advantages of default and stop word removal over stemming, while stemming seems to do a little bit better in terms of 1% lift scores. Overall we don’t see strong effects of stop word removal and stemming on the performance of the regression model, but given its improvement of running time result from sparsity, removing stop words is preferable.

3 Regression Model

We used the ridge regularization with $L_2$ error function. To compute the exact solution of the regression, we used Matlab’s built-in linear solver for sparse matrices. We experimented with different regularization weights ($\lambda$), and the results are presented in Section 5. We can see from Figure 2 and 3 that changing $\lambda$ didn’t affect the performance by a lot. We ranged $\lambda$ from 5 to 1400, and the AUC scores stay almost unchanged, while the higher $\lambda$ values give a slightly better 1% lift score.

4 Feature Selection

We only used uni-gram as features in our model. However, we examined the effect of frequency-based feature selection. Given parameter $k$, we pick the $k$ most frequently occurred tokens as our features, and experimented with various $k$ values ranging from 1000 to 10000. We can see from Figure 4 and 5 that the number of features has effects on the performance. Larger $k$ values give better results in terms of AUC and 1% lift scores. The result improves fast when $k$ goes from 1000 to 5000, and stops changing much after $k$
reaches 6000. Smaller $k$ values mean less workload for solving the regression model, thus picking $k = 6000$ is a good tradeoff between performance and efficiency.

5 Experiment results

In this section, we plot the ROC curve, as well as the AUC and 1% lift scores against various parameters.
Figure 2: AUC vs. $\lambda$

Figure 3: 1% lift score vs. $\lambda$
Figure 4: AUC vs. $k$

Figure 5: 1% lift score vs. $k$
**Term Weights.** In our ridge regression model, we computed the $\beta$ vector, and made prediction by multiplying $\beta$ and the feature vector. The strongest positive terms will have large positive values for their corresponding entries in $\beta$, and the strongest negative terms will have small (large absolute value) negative values for their corresponding entries in $\beta$. We sorted the entries in $\beta$ respectively, and obtained the indices of the first and last 10 tokens, then looked them up in the dictionary.

Note that this table is rather different from the terms we observe in assignment 1, in which the highly-weighed terms are all rare terms (e.g. shrek, mulan). The reason is that in assignment, there was no pruning of features, i.e. all words were used. In this assignment, however, we only take the top-K most frequent features, eliminating the uncommon words.

<table>
<thead>
<tr>
<th>Top Positive Words</th>
<th>Weights</th>
<th>Top Negative Words</th>
<th>Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>excellent</td>
<td>0.20</td>
<td>waste</td>
<td>-0.71</td>
</tr>
<tr>
<td>awesome</td>
<td>0.18</td>
<td>disappointing</td>
<td>-0.57</td>
</tr>
<tr>
<td>invaluable</td>
<td>0.15</td>
<td>poorly</td>
<td>-0.56</td>
</tr>
<tr>
<td>highly</td>
<td>0.15</td>
<td>disappointment</td>
<td>-0.52</td>
</tr>
<tr>
<td>amazing</td>
<td>0.15</td>
<td>worst</td>
<td>-0.47</td>
</tr>
<tr>
<td>outstanding</td>
<td>0.15</td>
<td>boring</td>
<td>-0.46</td>
</tr>
<tr>
<td>wonderful</td>
<td>0.14</td>
<td>disappointed</td>
<td>-0.46</td>
</tr>
<tr>
<td>loved</td>
<td>0.14</td>
<td>useless</td>
<td>-0.42</td>
</tr>
<tr>
<td>pleased</td>
<td>0.14</td>
<td>hoping</td>
<td>-0.32</td>
</tr>
<tr>
<td>fantastic</td>
<td>0.14</td>
<td>lacks</td>
<td>-0.31</td>
</tr>
</tbody>
</table>

Table 1: Top words and their weights using a lambda of 1405

**Time Efficiency.** After applying the techniques discussed in Section 2, the time spent for loading data is fairly light, and the running time is dominated by the computation of the exact ridge regression solution. The heaviest step is solving a linear system of the form $Ax = y$. The $A$ matrix in our model is the review by token matrix (add $\lambda I$), which is sparse, so the running time is determined largely by the number of nonzeros in the sparse matrix. The largest experiment we conducted is by picking the 10000 most frequent tokens as features. The dimension of the matrix is then $(nu$ of reviews)$ \times 10000$. As we noted earlier, we applied two techniques to make our matrix even sparser. By deduplicate the reviews, we drove down the total number of reviews (i.e. total number of rows), and by removing stop words we made each row contains less non-zero entries. The total number of nonzeros in our matrices (with top-10000 tokens, and stop words removed) is $22.9 \times 10^6$. 
6 Conclusion

We designed and implemented a linear regression sentiment analysis classifier. Using a book review dataset, we employed a number of implementation techniques to reduce memory resource requirement and improve computation speed. We constructed linear regression models with a number of different parameters, including stemming, stop-word-removal, extracting the top-K most frequent words as features. We concluded that the best setup is to employ stop-word-removal and taking the top 6000 most frequent features for smallest memory footprint while not sacrificing quality.

References
