Sentiment Analysis

Group G3: Tom Werner, Changghui Xu, John Tollefson

Project Description and Overview

We were tasked with doing sentiment analysis, and decided that the Large Movie Review Dataset, consisting of 50,000 ImDB movie reviews, would be a good dataset to work with. The datset is split into 25,000 training reviews and 25,000 testing reviews, and the positive/negative split is equal in both partitions. The reviews are coded with (1) meaning a positive review and (0) meaning a negative review.

In our project proposal we decided to target two models in particular - the Naive Bayes, model, well known in sentiment analysis tasks, and a Long Short Term Memory (LSTM) neural network model. If we had time we would explore additional models.

Our report will be divided into two main parts - one part for the more traditional approach and one part for the LSTM approach. We will detail our process for feature selection, model selection, and model structure selection, as well as data cleaning, preprocessing, and finally our testing accuracy.

Traditional Predictive Modeling

Basic Data Preparation

In order to work with the data, we needed to convert it from raw text into something more usable. The main steps we took were phrase splitting, word cleaning and negation propagation, and vocabulary assembly.

- **Phrase splitting**: Split each review on characters like ?.,!:; to identify "phrases"
- **Word cleaning**: Remove all the junk from words - every single character here was used at least once. "$%^&*()#@\'+-/0123456789<>=\[]_~{}|`
- **Negation propagation**: We used an idea from [2] to handle negations. If we saw "not", "n't", or "no" in a phrase, we negated it. So an example sentence "The movie, while not good, was decent", would be extracted as ["the", "movie", "not_while", "not_not", "not_good", "was", "decent"]. You can see how the middle phrase was negated. This allows us to handle things like that, where "not good" are used.
- **Vocabulary assembly**: We needed a global list of words in our training dataset to build our input matrix, where input[i][j] is the count of the j'th word in the i'th review.

Baseline model

With our data extracted, we first build a baseline model: a naive Bayes model that took binary inputs (word there/not there). It achieved an accuracy of 81% on 10 fold cross validation of the entire training dataset. We also tried using the counts instead of the simple binary inputs, but it made no difference for this model.

Data Selection

We knew we could do better than 81%, and we also knew that one way to improve was to use the inputs more effectively. As it stood, using the counts, the word "the" would be significantly more important than "awful", "terrific", or "amazing", simply because "the" has such a high frequency of appearance. To deal with this we looked at three different data sources.

Our basic count matrix, the TF-IDF transformed matrix, or the scikit-learn provided TF-IDF text extraction tool.

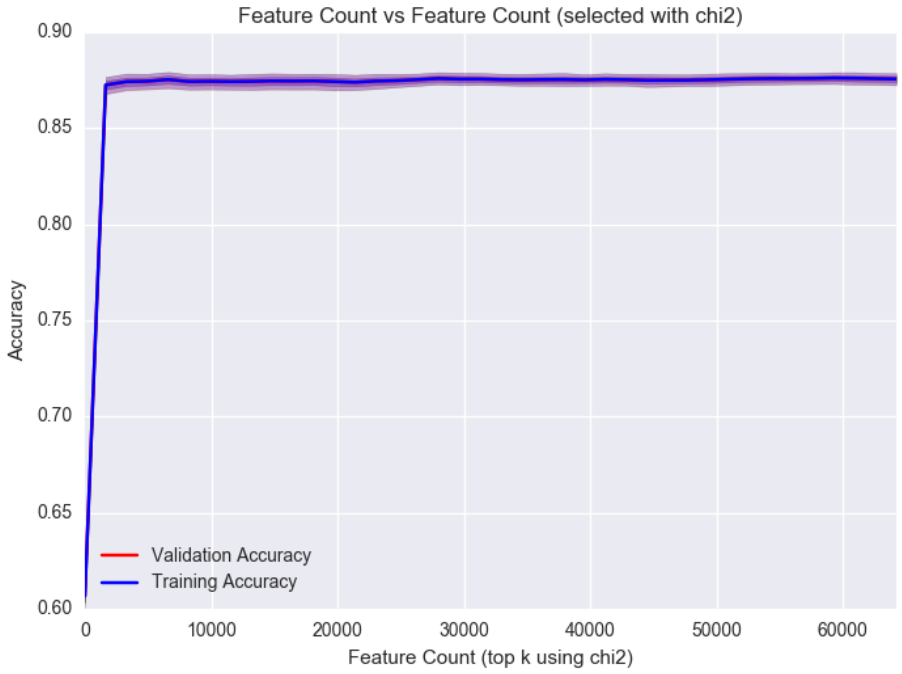
The TF-IDF transformation is the text frequency-inverse document frequency transformation, which take into account both how many times a term appears in a single document (review), but also makes it less important if it appears in many documents. This means words like "the" will be less important.
In order to empirically decide the best data source, we used 10-fold cross validation on our training dataset, using three models - we wanted to ensure that a single model performing poorly on a data source didn't overly influence our choice.

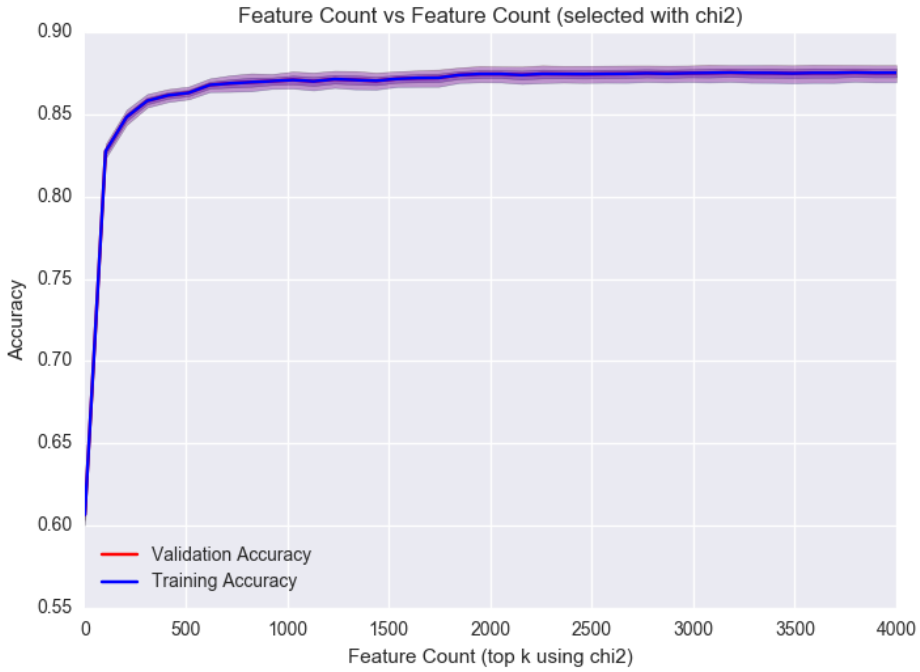
Model	Input Type	10-fold Accuracy	Stardard Dev
Bernoulli Naive Bayes	Normal Inputs	0.81	0.01
Multinomial Naive Bayes	Normal Inputs	0.79	0.02
LogReg	Normal Inputs	0.85	0.01
Bernoulli Naive Bayes	**TF-IDF Inputs**	**0.81**	0.01
Multinomial Naive Bayes	**TF-IDF Inputs**	**0.81**	0.01
LogReg	**TF-IDF Inputs**	**0.87***	0.01
Bernoulli Naive Bayes	Sci-kit learn TF-IDF	0.69	0.01
Multinomial Naive Bayes	Sci-kit learn TF-IDF	0.71	0.01
LogReg	Sci-kit learn TF-IDF	0.71	0.01

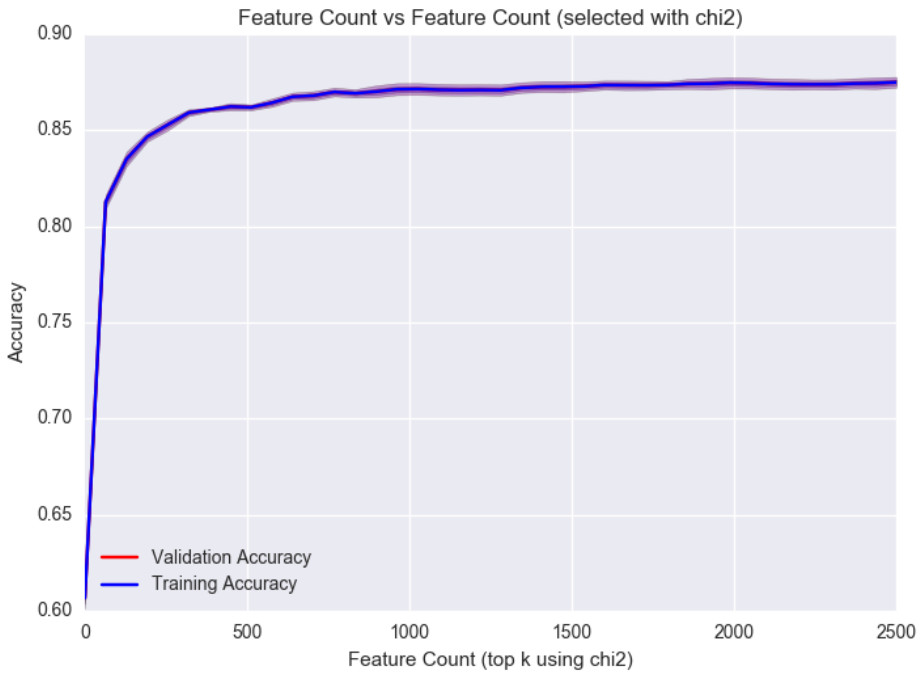
As we can see, the TF-IDF transformed input data outperformed the other data sources significantly. We also see that we have already beaten our baseline of 81% with a slightly misleading 87% accuracy. We'll go into why this is misleading in more detail later.

Feature Selection

Our vocabulary size from the review was over 60,000 words - or 60,000 variables to handle. The data is extremely sparse, which is why we could run this on personal computers, but it was still a large problem. To handle this, we used feature selection techniques that work on sparse data. To handle this, we used feature selection techniques that work on sparse data. Scikit-learn's K-Best feature selection tool allowed us to pick the top "K" variables, as determined by a chi-squared test. We then trained a model using the top K features, with K ranging from 0 to max, then 0 to 4000, and finally 0 to 2500. The two lines plotted are the training and validation accuracy. Because we do have balanced classes, accuracy was not unreasonable to use as a metric here. The model we used was a Logistic Regression model with L1 regularization, which we choose because it was a nice balance of training and performance.


Feature Count vs Feature Count (selected with chi2)
0.90
0.85
0.80
0.75
0.70
0.65
0.60
Accuracy
Validation Accuracy
Training Accuracy
0
10000
20000
30000
40000
50000
60000
Feature Count (top k using chi2)


Feature Count vs Feature Count (selected with chi2)
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
Accuracy
Validation Accuracy
Training Accuracy
0
500
1000
1500
2000
2500
3000
3500
4000
Feature Count (top k using chi2)



From these plots we can see that somewhere around 2000 features seems ideal. We then built a series of models a pipeline where we choose the top 2000 features, then used an L1 regularization feature selector, and finally trained the model in question to determine the best model.

Model	10-fold Accuracy	Standard Dev	10-fold train/pred time (seconds)
BernoulliNB	0.84	0.01	01.472
MultinomialNB	0.85	0.01	00.956
LinearSVC	0.87	0.01	02.895
L1_LinearSVC	0.87	0.01	03.802
LogisticRegression	0.86	0.01	04.200
L1_LogisticRegression	0.87	0.01	05.601
RandomForest	0.77	0.01	47.391
RBF_SVC	0.70	0.01	56 minutes

Even our simple naive Bayes models now perform better, and we have 3 models that are achieving our current best of 87% accuracy. There are a couple things to note here:

- The RBF kernel SVM took an extremely long time to run without much payoff. We excluded it from our later analysis.
- The 87% accuracy we achieve here is much more meaningful than our previous 87% that we called slightly misleading. Even though all of these models are being run using 10-fold cross validation, our misleading earlier results are still overfitting to a degree. Because that model used all 60,000+ features, the entire vocabulary of the training set, it still overfits because the test set will not have the same vocabulary. The top 10,000 words might be the same, even the top 20,000, but certainly not all of them. Because of this feature selection is essential for accurate model evaluation, and so our new models that achieve 87% accuracy on a very limited subset of the features are much more likely to generalize well. If we had run cross validation where we did vocabulary extraction from that particular training set, and used that on the testing set, we could eliminate this issue, but for runtime considerations we did not and focused instead on feature

selection to handle this issue.

N-Grams

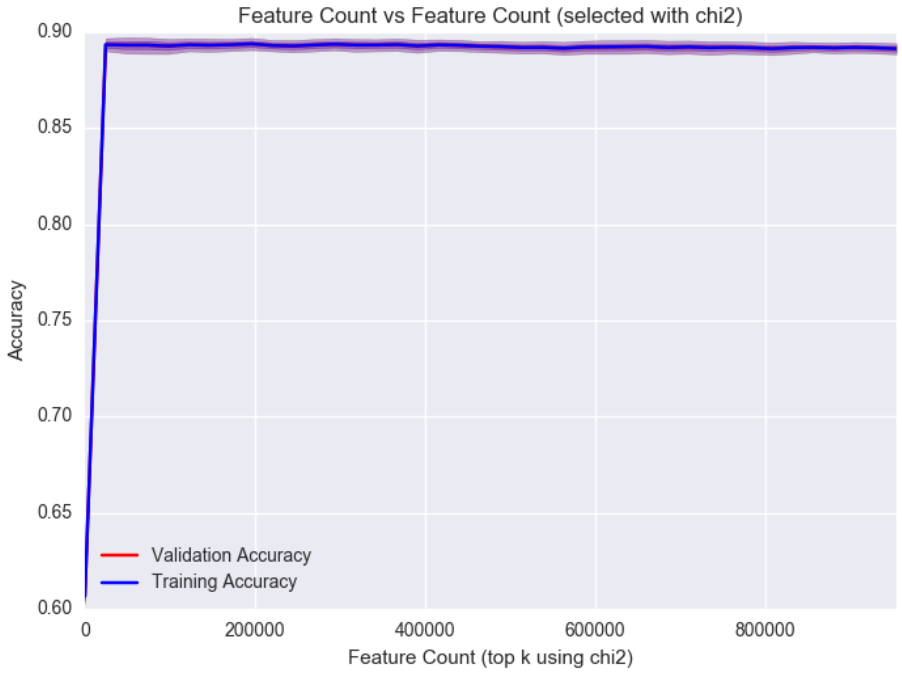
Another technique to improve the accuracy of sentiment analysis is the inclusion of N-Grams, which capture more than just individual words. This is useful in a sentence like "Thor was very good". The two grams are "Thor was", "was very", "very good". "Very good" gives us more information than either word on their own. Using 2 and 3 grams does dramatically increase the number of features, so we re-evaluated our data input choice (excluding the poorly performing default scikit-learn implementation.

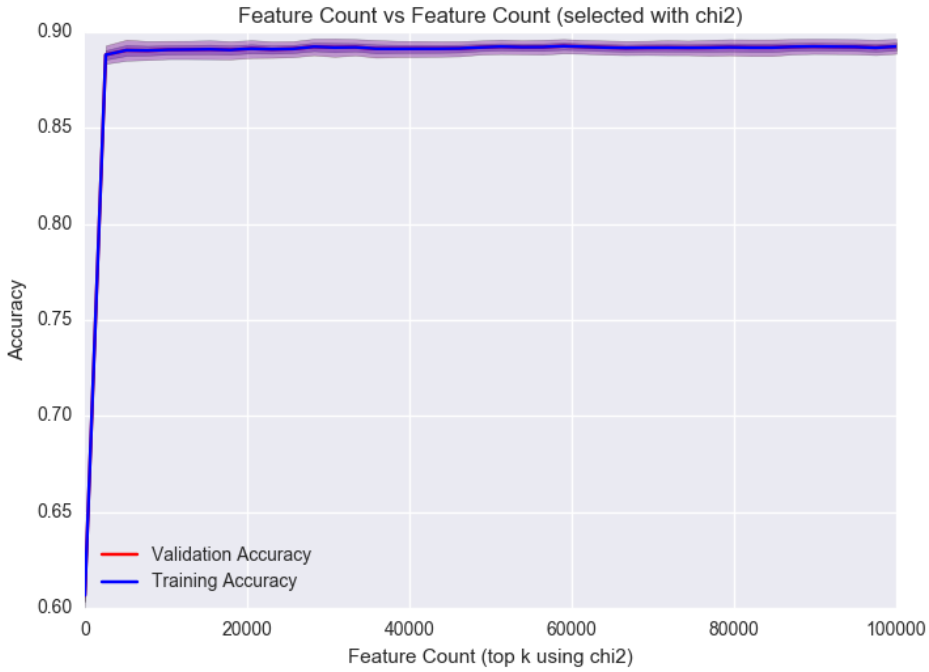
Model	Input Type	10-fold Accuracy	Stardard Dev
Bernoulli	Normal Inputs	0.86	0.01
Multinomial	Normal Inputs	0.85	0.01
LogReg	Normal Inputs	0.87	0.01
L1_LinearSVC	Normal Inputs	0.86	0.01
LinearSVC	Normal Inputs	0.88	0.01
Bernoulli	TF-IDF Inputs	0.86	0.01
Multinomial	TF-IDF Inputs	0.86	0.01
LogReg	TF-IDF Inputs	0.86	0.01
L1_LinearSVC	TF-IDF Inputs	0.88	0.01
LinearSVC	TF-IDF Inputs	**0.89***	0.01

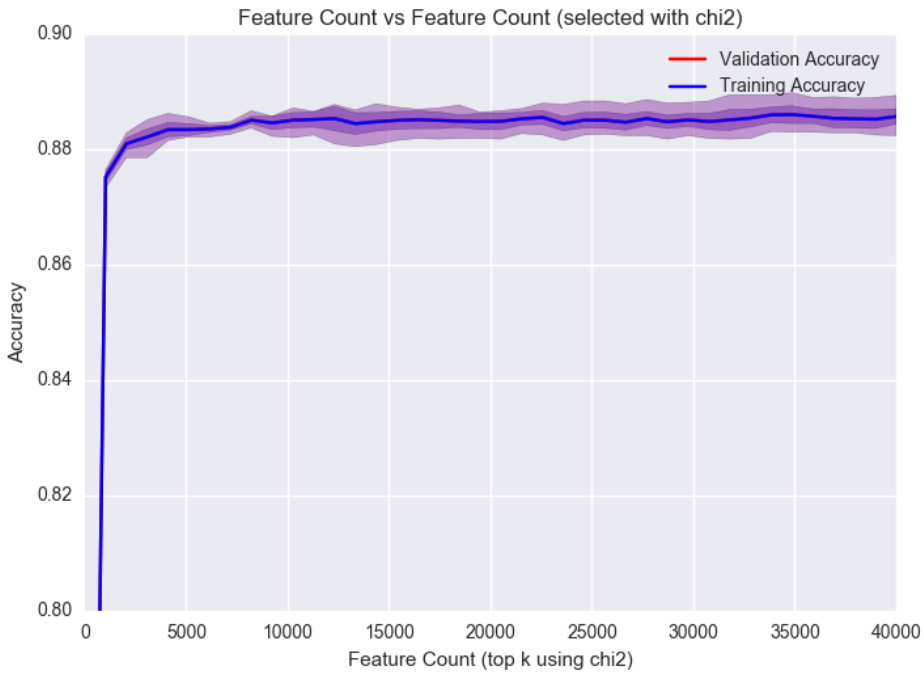
The TF-IDF data still performs the best, and we again see the misleadingly high 89% accuracy.

Feature Selection Part 2

Including 2 and 3 grams adds an incredible number of features to the model, so we repeated our feature selection pipeline. We were able to go from 1,000,000 features down to 13,000. The process followed here is the same as without the N-grams, it just took longer to run. Additionally, our evaluation model has now changed to an L1 regularized LinearSVC, since it out performs the logistic regression model.


Feature Count vs Feature Count (selected with chi2)
0.90
0.85
0.80
0.75
0.70
0.65
0.60
Accuracy
Validation Accuracy
Training Accuracy
0
200000
400000
600000
800000
Feature Count (top k using chi2)


Feature Count vs Feature Count (selected with chi2)
0.90
0.85
0.80
0.75
0.70
0.65
0.60
Accuracy
Validation Accuracy
Training Accuracy
0
20000
40000
60000
80000
100000
Feature Count (top k using chi2)



To test that evaluation model, we ran a series of tests using those top 13,000 features to choose our L1 selection stage.

Model Selection

L1 Feature Selection Step	Model	10-fold Accuracy	Standard Dev
LogReg L1	BernoulliNB	0.84	0.01
LogReg L1	MultinomialNB	0.86	0.01
LogReg L1	LinearSVC	0.87	0.01
LogReg L1	L1_LinearSVC	0.87	0.01
LogReg L1	LogisticRegression	0.84	0.01
LogReg L1	L1_LogisticRegression	0.86	0.01
Linear SVC L1, C=.1	BernoulliNB	0.82	0.01
Linear SVC L1, C=.1	MultinomialNB	0.84	0.01
Linear SVC L1, C=.1	LinearSVC	0.84	0.01
Linear SVC L1, C=.1	L1_LinearSVC	0.85	0.01
Linear SVC L1, C=.1	LogisticRegression	0.82	0.01
Linear SVC L1, C=.1	L1_LogisticRegression	0.85	0.01
Linear SVC L1, C=1	BernoulliNB	0.86	0.01
Linear SVC L1, C=1	MultinomialNB	0.86	0.01
Linear SVC L1, C=1	**LinearSVC**	**0.88**	**0.01**
Linear SVC L1, C=1	**L1_LinearSVC**	**0.88**	**0.01**

L1 Feature Selection Step	Model	10-fold Accuracy	Standard Dev
Linear SVC L1, C=1	LogisticRegression	0.86	0.01
Linear SVC L1, C=1	L1_LogisticRegression	0.86	0.01
Linear SVC L1, C=10	BernoulliNB	0.86	0.01
Linear SVC L1, C=10	MultinomialNB	0.86	0.01
Linear SVC L1, C=10	**LinearSVC**	**0.88**	**0.01**
Linear SVC L1, C=10	**L1_LinearSVC**	**0.88**	**0.01**
Linear SVC L1, C=10	LogisticRegression	0.86	0.01
Linear SVC L1, C=10	L1_LogisticRegression	0.86	0.01

We can see that the Linear SVC with L1 regularization yields the best general performance, at C=1 or 10, and then the Linear SVC with L1 or L2 regularization on top of that performs the best. (See bolded rows) Our pipeline now looks like

- Top 13,000 features using K-Best with chi-squared test
- Linear SVC with L1 regularization, take features with non zero coefficients
- Linear SVC with L1 or L2 regularization

Model Structure Selection

Now that we've established our pipeline, its time to see if changing the complexity of our final model has a significant impact on our accuracy. For SVM's, that parameter is C.

L1 Feature Selection Step	Model	10-fold Accuracy	Standard Dev
Linear SVC L1, C=1	LinearSVC C=.001	0.76	0.01
Linear SVC L1, C=1	**LinearSVC C=1**	**0.88**	**0.01**
Linear SVC L1, C=1	LinearSVC C=.1	0.86	0.01
Linear SVC L1, C=1	**LinearSVC C=10**	**0.88**	**0.01**
Linear SVC L1, C=1	LinearSVC C=100	0.87	0.01
Linear SVC L1, C=1	L1_LinearSVC C=.001	0.50	0.00
Linear SVC L1, C=1	**L1_LinearSVC C=1**	**0.88**	**0.01**
Linear SVC L1, C=1	L1_LinearSVC C=.1	0.83	0.01
Linear SVC L1, C=1	L1_LinearSVC C=10	0.87	0.01
Linear SVC L1, C=1	L1_LinearSVC C=100	0.86	0.01

From this we concluded that our optimal pipeline is

- Top 13,000 features using K-Best with chi-squared test
- Linear SVC with L1 regularization, C=1, take features with non zero coefficients
- Linear SVC with L2 regularization, C=1
- Using these settings, we achieved 88% accuracy on 10 fold cross validation with an average of 1697 features used. (std=18)

Test Set Evaluation

To prepare the test set input matrix, we used the same preprocessing as we did on our training data, but this time we had to use the same vocabulary as the training data. So for words/n-grams that appear in training but not in test, the entire column would be zero, and for words/n-grams that appear only in test, their information is lost.

Having done this, we trained our pipeline on the entire training set using the parameters selected through 10-fold cross validation, and achieved a final accuracy of **88.052%** accuracy on 25,000 reviews!

What features were used?

Here's a small subset of some of the features selected by our pipeline as useful. 2 and 3 grams have been bolded.

- bad
- great
- awful
- worst
- **the_worst**
- **not_even**
- **of_the_best**
- and
- just
- nothing
- boring
- minutes
- no
- waste
- **not_this**
- excellent
- love
- poor

K-Nearest Neighbors

We use `sklearn` package `KNeighborsClassifier` to classify movie reviews based on KNN method.

All words are vectorized using a TF-IDF vectorizer, which ignores words that appear in less than 5 documents and ignores words that appear in more than 80% of documents.

The number of neighbors in KNN classifier determines the accuracy of the prediction. We don't want the number of neighbors too few, which could not do a good job in classification. We don't want the number of neighbors too many either, which would consume too much computation power and could potentially over fit the model.

Below is a list of training and testing using a series of numbers of neighbors. In the end, we use 19 neighbors, which give an accuracy of about 77%.

number of neighbors

n_neighbors	accuracy
3	0.6801
4	0.6748
5	0.703
6	0.6989

n_neighbors	accuracy
7	0.7155
8	0.7154
9	0.7349
10	0.732
11	0.7448
12	0.7359
13	0.7526
14	0.747
15	0.7572
16	0.7524
17	0.7638
18	0.7592
19	0.7692

KNN classifier is relatively slower than Naive Bayes and SVM for text classification. Probably it is because of the complicated distance metrics.

LSTM Model

Due to computationly intensive of LSTM method, we only use two LSTM layes in our classifcation model. These two LSTM layes are bidirectional, which include a forwads LSTM and a backwards LSTM.

Feature extraction was done by reading all training reviews and tokenizing all english words, as well as removing stop words using `nltk` package.

Training in LSTM RNN contains two steps. First, run the neural network going forward. This sets the cell states. Then, you go backwards computing derivatives. This uses the cell states (what the network knows at a given point in time) to figure out how to change the network's weights. When LSTM updates cell states, we choose to use the default `Adam` optimizer (http://arxiv.org/abs/1412.6980v8), which is a method for Stochastic Optimization. The optimizer minimizes the loss function, which here is the mean square error between expected output and acutal output.

input matrix shape is (number of samples x maxlen)

`number_of_samples` here is 25000 reviews. All reviews are transform into sequences of word vector.

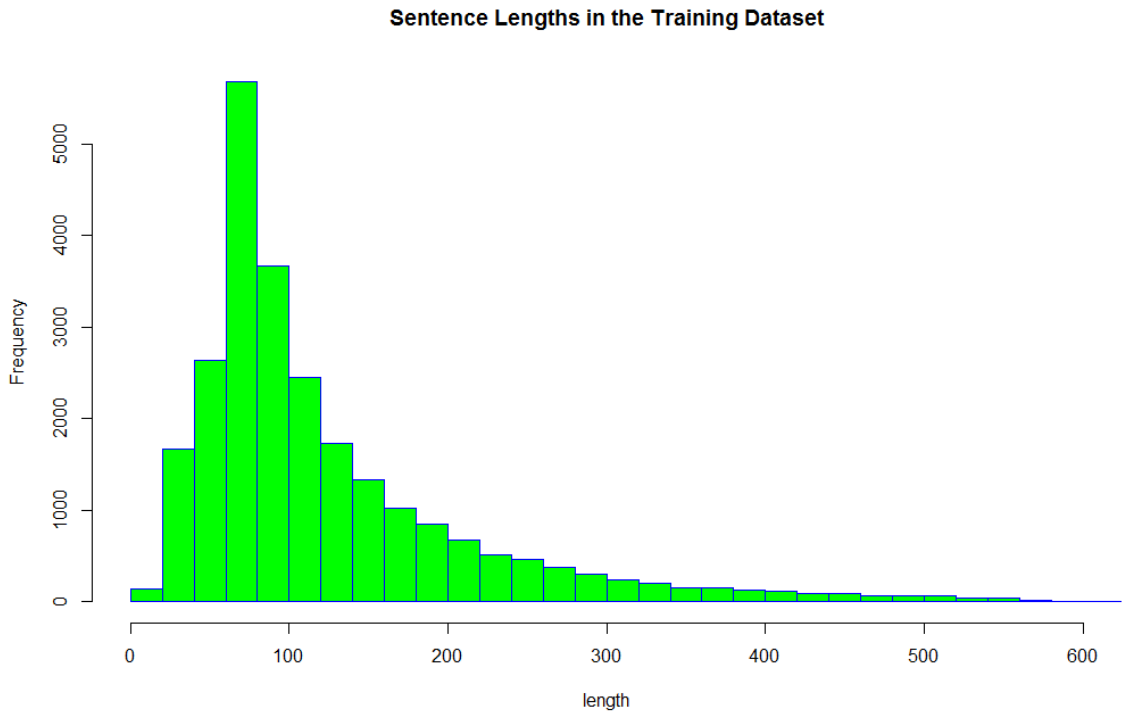
`maxlen` is the max length of each sequence. i.e., if a review has more than `maxlen` words, then this review will be truncated. However, if a review has less than `maxlen` words, then the sequence will pad 0's to make it a regular shape.

`max_features` is the dictionary size. The dictionary was created before data feed into LSTM RNN. Dictionary keys are purified words, dictionary values are the indicies, which is from 2 to 90000. Such that, the most frequent word has lowest index value. For those rarely occurred words, their indicies is large. We can use `max_features` to filter out uncommon words.

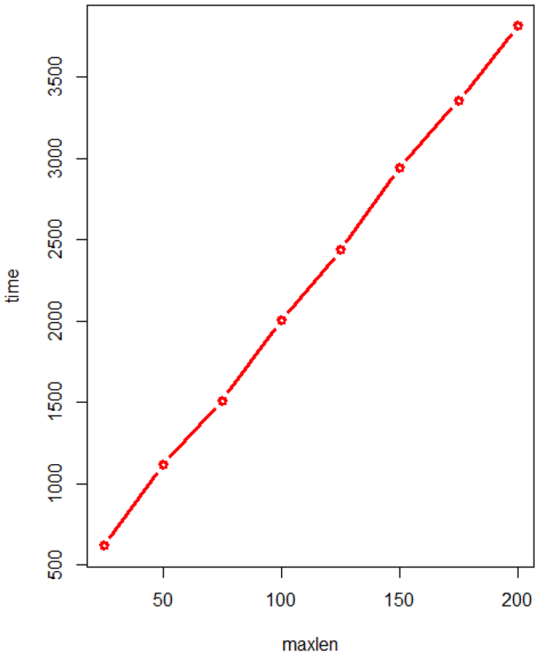
First, keeping the `max_features = 20000` , we tested the effect of `maxlen` , which varied from 25 to 200.

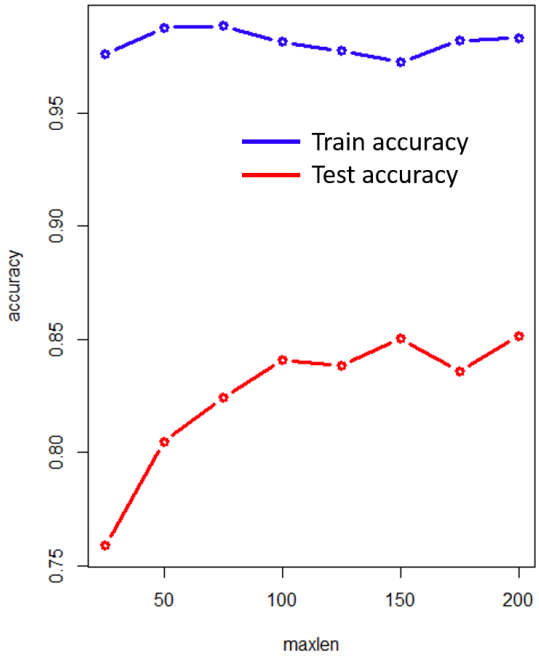
maxlen	time (s)	train accuracy	test accuracy
25	618	0.9757	0.7589

maxlen	time (s)	train accuracy	test accuracy
50	1113	0.9876	0.8047
75	1507	0.9882	0.8243
100	2004	0.9813	0.8410
125	2435	0.9774	0.8384
150	2939	0.9725	0.8503
175	3352	0.9819	0.8359
200	3811	0.9831	0.8514



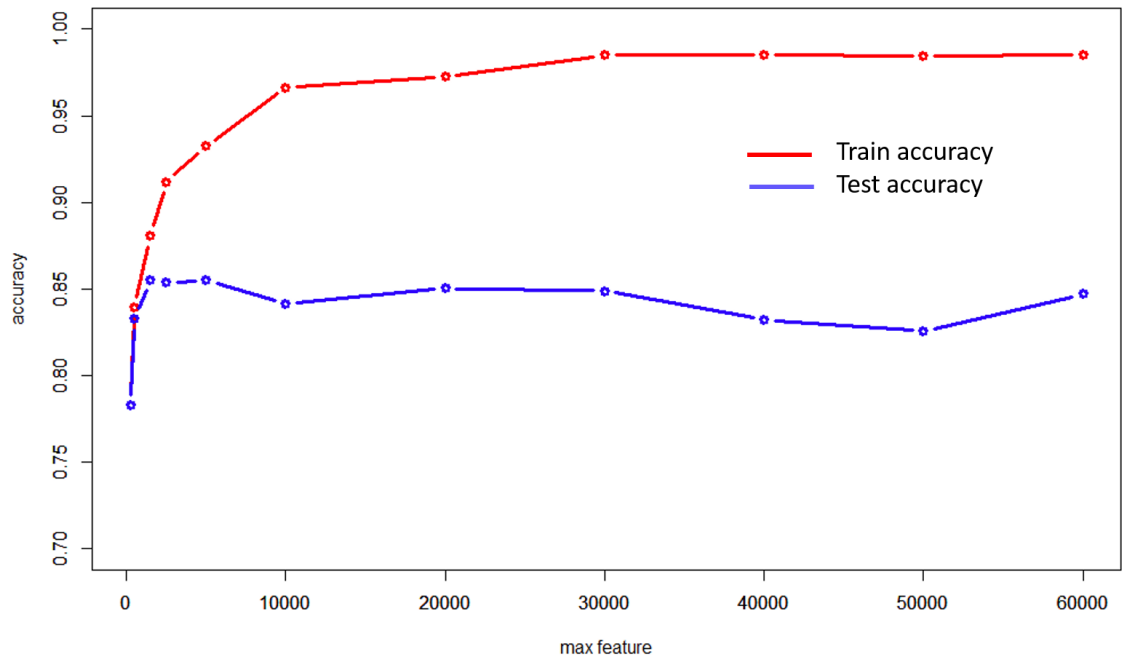
The length of sentences are right skewed (Q1:67, Median 92, Q3:152). With squence length of 150, about 75% of reviews are covered.





Second, keeping the `maxlen = 150`, we tested the effect of `max_features`, which varied from 250 to 50000.

max_features	train accuracy	test accuracy
250	0.7828	0.7722
500	0.8392	0.8328
1500	0.8806	0.8554
2500	0.9119	0.8536
5000	0.9324	0.8553
10000	0.9664	0.8412
20000	0.9725	0.8503
30000	0.9850	0.8489
40000	0.9854	0.8321
50000	0.9843	0.8257
60000	0.9854	0.8470



It is interesting to notice that the most frequently appeared 2500 english words could largely determine the sentiment of movie reviews very well. Britain's Guardian newspaper, in 1986, estimated the size of the average person's vocabulary as developing from roughly 300 words at two years old, through 5,000 words at five years old, to some 12,000 words at the age of 12.

Future impovements

(1) Something that could help cut down on extraneous words is pyenchant https://pythonhosted.org/pyenchant/api/enchant.html. Basic idea is to make your input text a list of words, and fix spelling errors (or recorrect words that shouldn't belong).
(2) Stem word and word2vec
(3) Adding GPU support for the LSTM

Sources

[1] Maas, Andrew L., et al. "Learning word vectors for sentiment analysis." Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1. Association for Computational Linguistics, 2011.

[2] Narayanan, Vivek, Ishan Arora, and Arjun Bhatia. "Fast and accurate sentiment classification using an enhanced Naive Bayes model." Intelligent Data Engineering and Automated Learning–IDEAL 2013. Springer Berlin Heidelberg, 2013. 194-201.

[3] http://deeplearning.net/tutorial/lstm.html

[4] James Hong, Michael Fang, Sentiment Analysis with Deeply Learned Distributed Representations of Variable Length Texts, 2015.https://cs224d.stanford.edu/reports/HongJames.pdf