Final Project - Textual Analysis (ECE590.06)

An Exploration in Sentiment Analysis

Aaron Williams

Due: 12-16-2018

Introduction:

Sentiment analysis is a categorization method for passages of text which predicts the view of the writer by picking up on word usage. In the case of an RNN, the order of the word use is also important. Sentiment analysis is important for analyzing large packages of text, such as comments on message boards or tweets. Not only can it be used to determine if someone is overall positive or negative in their assertions, but it could potentially categorize passages into more intricate contextual and emotional subsets. In the day and age where people's words are often linked directly to themselves as pertains to a large percentage of the population, there's a lot you can glean about the person behind the words. This has limitless benefits in targeted advertisements and articles or even just analyzing the public effect of a certain choice.

For this project, I've chosen a relatively simple task to allow more in-depth coverage of the topic. I'll be picking up where we left off in Assignment 12 using the same NLTK movie reviews set. This consists of 2000 reviews, around evenly distributed as positive and negative. I'll be looking at a few different things. Does having a training set improve the ability to determine the sentiment of a review, and by how much? Does using word embeddings (or a vector representation of words taken from a large model) improve our ability to categorize a review? Does a more complex model of the given information help and to what extent? The overall goal is to qualitatively analyze several different methods and perhaps identify how to proceed with sentiment analysis in the future.

Methodology:

Tools and Environment

This entire project will be compiled in a Jupyter Notebook coded in Python 3. I ran this code on a Windows 10 Desktop with a GTX1060 GPU.

General data handling will be done with Numpy. NLTK provide the dataset and some corpi information for lexicon expansion and word embeddings. When using SVM, will be using the sklearn SVC method. For the Neural Network Models, we'll be using Tensorflow, either just the 'layers' methods or the contrib.rnn.LSTMCell for our RNN. For Word Embedding Models, we'll be using gensim. I'll go more in-depth in how these are applied later.

Cleaning the Data

Each review consists of an ordered list of strings with its sentiment attached as either 'pos' or 'neg'. In order to just look at the word usage, we're removing punctuation and anything that could be converted into a number.

While punctuation could potentially be used to signify tone, we're just not going to use it as I'm not sure how it would interact with Word Embeddings and whether it might cut up the one-hot vector space. I also considered using a stemmer for this section to combine like words, but as we're only looking at high frequency words for one-hot vectors and as I'm not sure how exactly the embeddings are stored, we're going to keep the words as they are. In both cases, like words, if utilized, end up occupying similar areas in multi-dimensional space.

Training and Test Set

The datasets will have their order randomized initially. A seed was included to replicate datasets in the future if necessary. We're going to use a 60-40 training-test split. This means that the training set will consist of 1,200 passages while the test set will have 800. Overall, the dataset is small, but 1200 full passages of text should be enough information, if not to provide an intricate model, to at least provide a somewhat standardized basis for what the data looks like. I'll talk about how these sets will be applied as they pertain to the specific models and methods.

Lexicon Expansion

The setup for lexicon expansion will be somewhat improved from what was submitted in Assignment 12. We'll start with 5 words, both good and bad. I picked words that I hoped would fully define the good and bad qualities of a movie, but obviously it differs by genre among other things. These words are as follows:

good_words = ["excellent", "interesting", "perfect", "impeccable", "exhilarating"]

bad_words = ["boring", "unimaginative", "terrible", "bad", "embarrassing"]

As in Assignment 12, we'll use wordnet. For each word in the set, we'll look at each synset of that seed and select the first word that isn't already included in our set. If none are selected, we move on to the next synset and then the next word. The goal of selecting these words is to reach some pre-determined good and bad set size. One analysis we'll be looking at is how that set size changes classification.

In order to find our final logit values, we simply iterate through each item in the Test Set (800 points mentioned in the previous section) and count how many good words were contained in the review and count down for any bad words. It is important to note that our final logits will not be centered as there is no training set to establish an average of good and bad word counts. It would be possible to center the outcomes roughly with regards to the distribution of classes. Analysis will generally take place using ROC curves.

Top 1000 words

When we want to gain more information from words, we'll use two methods. The first is to establish a list of the top 1000 most common words in the training set. 1000 is an arbitrary cutoff but given that there are a few hundred words per review, one thousand guarantees that the word will appear in a few different reviews so that some information might be gained from its presence. If we use too few words, we run the risk of only including non-informative common words like 'the' and 'and' and losing vital information. If we use too many words, our model potentially overfits around uncommon words that appear only in a few reviews.

Word Embeddings

Our other method is word embeddings. NLTK has a word embedding model already saved with the vocabulary pruned to 44,000 words. This model is derived from the Google News Dataset. Word Embeddings allow us to use each word in any given review while being confident that too much information is neither gained or lost.

We can also use word embeddings to expand a list of good and bad words like lexicon expansion, but instead we'll rely on the vocabulary in our embedded model. It's possible to query the model with the *.most_similar()* method by including our lists of words both in the positive and negative section. By specifying the number of words to include, we can easily bolster an existing list of seeds. We'll use our positive and negative seeds in the same way as was implemented in lexicon expansion to make predictions on our test set.

Applying our Information Methods

The reviews from our training and test generally include a list of words. To use our top-1000 words method, we're going to create one-hot vectors. Simply, any word that's within the list included in the review that is among these top-1000 words will be inserted as a 1000-length zero vector with a one in the word's index position. Anything not in the list will be inserted as a 1000-length zero vector. For the embedded words, the same methods apply. Any words in the external genism model will be included as the 300-length vector it defines. Any words not in the model's vocabulary will be inserted as 300-length zero vector.

Association Analysis

Before we start using models, we're going to test a method of sentiment analysis which uses our top 1,000 words but does not involve one-hot vectors. We're going to use a trick that's a little like how we applied topics in Assignment 9. Essentially, we count the number of times each of the top 1000 words is used in either positive or negative association. We add one to each of these counts to help with normalization. For each word, we take the log of the ratio between the positive and negative associations and divide it with respect to the total counts for that word. We also consider the number of total words with positive and negative connotations and apply that before taking the log.

To score our test set, we simply look through each word and if it's in the top 1000 words, we add its association score which eventually sums to a logit for the review.

Different Models and How They'll be applied

We're going to be using several different models, hopefully in a manner that goes from least to most complicated. These models will be trained and tested on both the one-hot vector sets and the word embedded sets. The models we are going to use are a Linear Least Square Regression, a Soft Margin SVM, a 2-Layer Neural Network, and a Recurrent Neural Network.

For every model besides the RNN, each review has its word vectors summed and divided by the word count. This results in a review datapoint of either 1000-length or 300-length for one-hot vectors and embeddings respectively.

Linear Least Squares Regression

We will be using the *np.linalg.lstsq()* method with defaults. The training set labels will be characterized as either '1' or '-1'.

Soft Margin SVM

We're going to use the *sklearn.svm.SVC()* method with a *kernel='rbf'*. We'll be adjusting our C value to maximize Test Accuracy as for all soft margin SVMs, there needs to be some sort of allowance term to allow the SVM to set an appropriate boundary. Other than that, we will be using the default settings of the method including an automated 'gamma'. The labels will be '1' or '-1'.

2-layer NN

For our 2-layer NN we'll be using tensorflow. In terms of standard values, we're going to be training off batches of 50 data points. Our hidden layer will have 50 neurons (regardless of whether we're training off one-hots or embedded vectors) and will have RELU activation function. 50 neurons between both will allow us to analyze results better and seems like a fairly good number for both a 1000 and 300 sized input vector. All layers our defined as *tf.layers.dense()* with otherwise default settings. The loss optimizer we'll be using is *tf.train.AdamOptimizer()* as I think it's a great generalized Gradient Descent model that solves most problems adequately.

RNN

I'm not too experienced in training RNNs, so you'll have to bear with me. Each dataset item will be maintained in its vector list form. We will train on 5000 random selections from the training set. We're going to use the *tf.contrib.rnn.LSTMCell()* method with 50 neurons and a 'tanh' activation function. Long short-term memory is an RNN method that's generally more computation intensive due to its number of gates. These gates include an 'input gate', 'output gate', and a 'forget gate'. However, LSTMs perform better when passages have a larger overall length which is applicable to the reviews in this dataset.

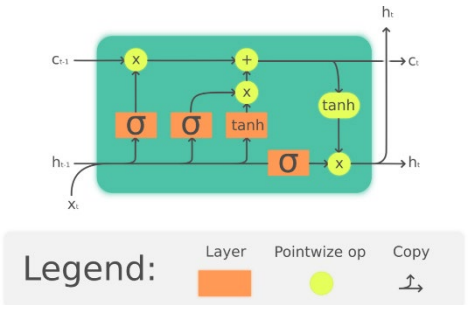

Figure 1 - LSTM Cell

It's important to include an RNN in this analysis, even a poor one, as an RNN is by far the best method of identifying sentiment. Even a simple phrase like 'not good' would be broken down as *'not'=neutral 'good'=positive* whereas an RNN would be able to identify the overall negative connotation by storing that 'not' as negator.

Looking at Results

For the most part, we'll be analyzing ROC curves. For our untrained models, we'll be looking at how expanding our seed count is associated with a *best accuracy* term. Our *best accuracy* is the highest accuracy we achieve when iterating the decision boundary. The reason we use this rather than the original accuracy is that the datasets are generally poorly balanced between positive or negative words, so we tend to predict a higher volume of positive reviews. Also, we will look at the C value in an SVM and see what changing that says about one-hot vectors and Word Embeddings. We will also analyze how our 2-layer NNs converge as the results for that were interesting and perhaps indicative.

Results + Analysis:

Lexicon Expansion vs. Word Embeddings

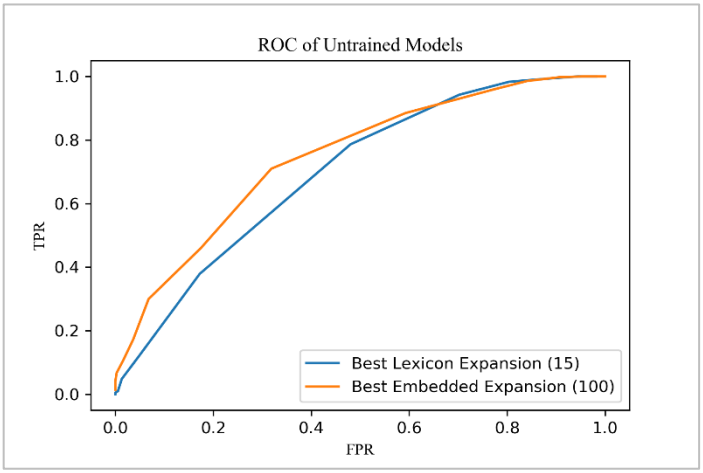

Figure 2 - ROC of best Expansion Models

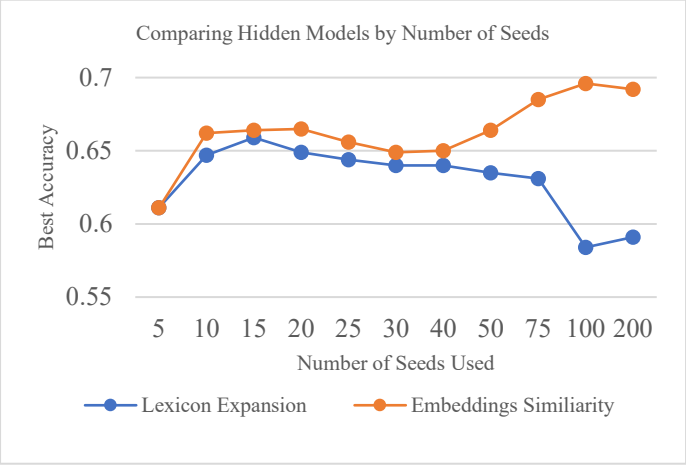

Figure 3 - Accuracy by number of Seeds

As you can see in the figures above, the Embedded Similarity metric performs better for most decision bound placements. After adding more seeds after 10, lexicon expansion deteriorates in best accuracy whereas the embeddings results continue to improve.

The main question is where we attribute the gains between these two untrained methods. Lexicon Expansion runs the risk of finding words that aren't related to the intent of the seeding and continuing to diverge from the original set. It's likely that this caused the best accuracy drop-off when the Embedding similarity list is restrained to the original list and will always find the nearest selection of words in high dimensional space. It's also possible that the pruned Google News list is simply more comprehensive then Lexicon Expansion Set and thus can find a better selection of words.

Association Analysis

The figure on the left shows the ROC comparison between our Expansion methods and our trained association method. It's quite clear that the Trained Association Method can glean more from the top-1000 words than either expansion method is able to guess the proper word associations on their own.

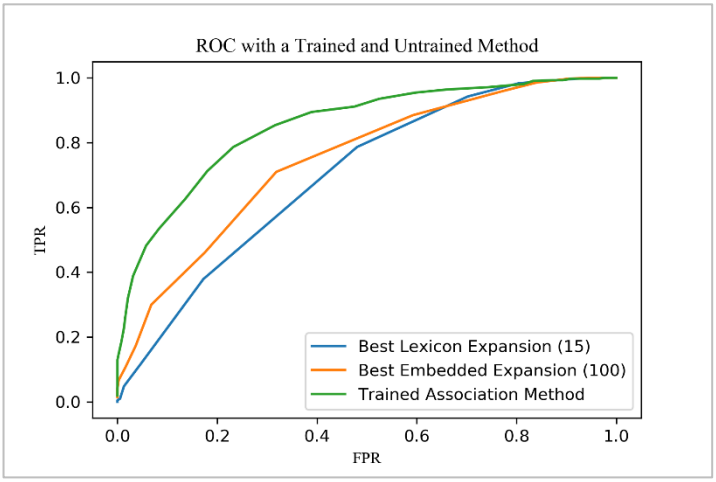

Figure 4 - Comparing Trained and Untrained Method

We must assume that the trained method has a better idea of the word usage included in the movie reviews, which means its better adjusted to determining the tone of a test passage. Also, it's important to note that the Association Method produced an accuracy outcome of *0.780* without iterating the decision boundary. This demonstrates that a training set isn't just convenient, but a necessity for achieving accurate sentiment predictions. Rather that guess at word usage, we can rely on proven word associations.

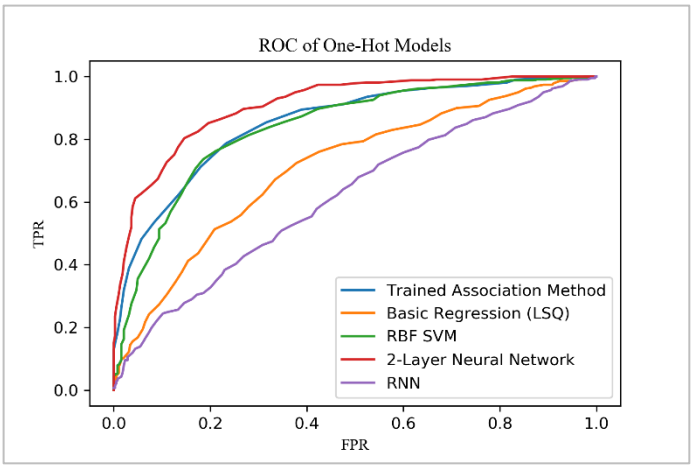

Figure 5 - One Hot Vector Model Results

Trained Model Comparison

Looking at our One-Hot Vector Models shows a few things. Only the 2-layer NN performs strictly better than our trained association method, and the SVM performs comparably. In turn, SVM outperforms Basic Regression. Our RNN performs very poorly and in tests only had an accuracy of *0.567* which is indicative of model issues.

The linear model doesn't perform exceptionally well, and I believe the main cause of that is that it's difficult to identify a separation in space with such a large vector space, particularly when you have so my features that have different importance levels. SVM and the associative method perform about the same, and that's likely due to their similarities in establishing word importance based on separation from the boundary. The 2-Layer NN performs best by far, likely as the model can establish complex interactions between different word appearances and perhaps build complex associations even just from one hot vectors.

As for our RNN, my thoughts on why its performance is bad center around two issues. One is that each review text is quite long and RNNs can experience diminishing values across lengthy sets if not properly tuned. Two, the RNNs took a long time to train and it was difficult to get a good perspective on the loss curve. I'm not too experienced at training RNNs, and this was a difficult problem for that.

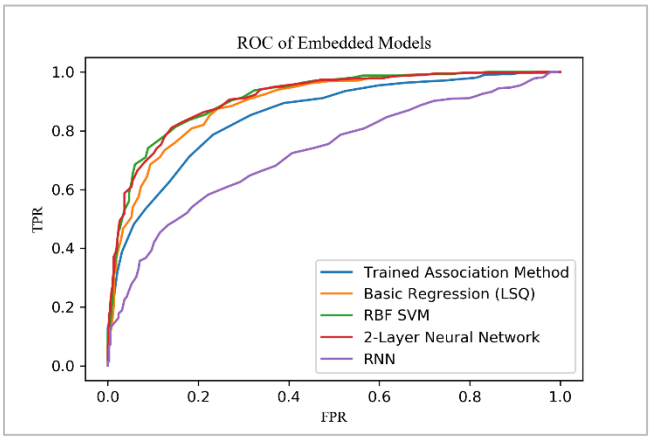

Figure 6 - Embedded Model Results

Embedded Models

Our embedded Model performance, at a glance, is much better than just using one-hot vectors. We see now that every model but the RNN performs better then the Trained Association Method. This is likely due to how information is included in embedded vectors that are acquired from a vast data model. Each word is maximally separated from all other and provide each model with as much information as possible on 300 dimensions. As is visible, this affects our linear model perhaps the most.

In total, RBF SVM and the 2-layer NN perform similarly and produce the two highest accuracy outcomes of our models overall. *2-layer NN accuracy ~ 0.835* and *RBF SVM accuracy ~ 0.829*. Our Linear Least Squares Method isn't far behind as shown in the above ROC.

Regression

Linear Least Squares Regression improved the most out of any model when adding in embedded vectors. This is likely due to the improved characterization and vector distance in space that resulted from using these trained terms.

SVM

Included to the right is a graph which compares SVM accuracy to the C constant. C is a soft margin parameter that weights how penalties are applied to individual points. As we can see, the C constant peak is lower for the embedded model while the OneHot Model prefers a higher C. This is likely due to the innate separability of the two models. The OneHot Model requires a higher allowance for boundary transgression while the Embedded Model maximizes distance parameters between each word, so the vectors are farther away in space. It's worth noting that I initially had a large amount of difficulty in setting up the SVM as I didn't realize how much I needed to scale the C constant due to the small space separation and overlap of our data set.

2-Layer NN

The training rate differs between the one-hot and embedded model despite their same parameters in batch size, gradient descent terms, and hidden layer neurons. The one-hot model appears to peak at around 2,500 iterations while the embedded still seems un-converged. This indicates that although the one-hot model has a larger vector size, less information is contained in each of those vectors so generalization occurs much quicker.

RNN

While the results of the RNNs are generally disappointing, it's obvious the RNN is improving classification above random guessing and that using embeddings improves our results. By using embeddings, you can take two similar but not exact sentences which use different words and find the same meaning in them. By having the associations for words in context, you can establish a more robust method of determining sentiment between ordered phrases.

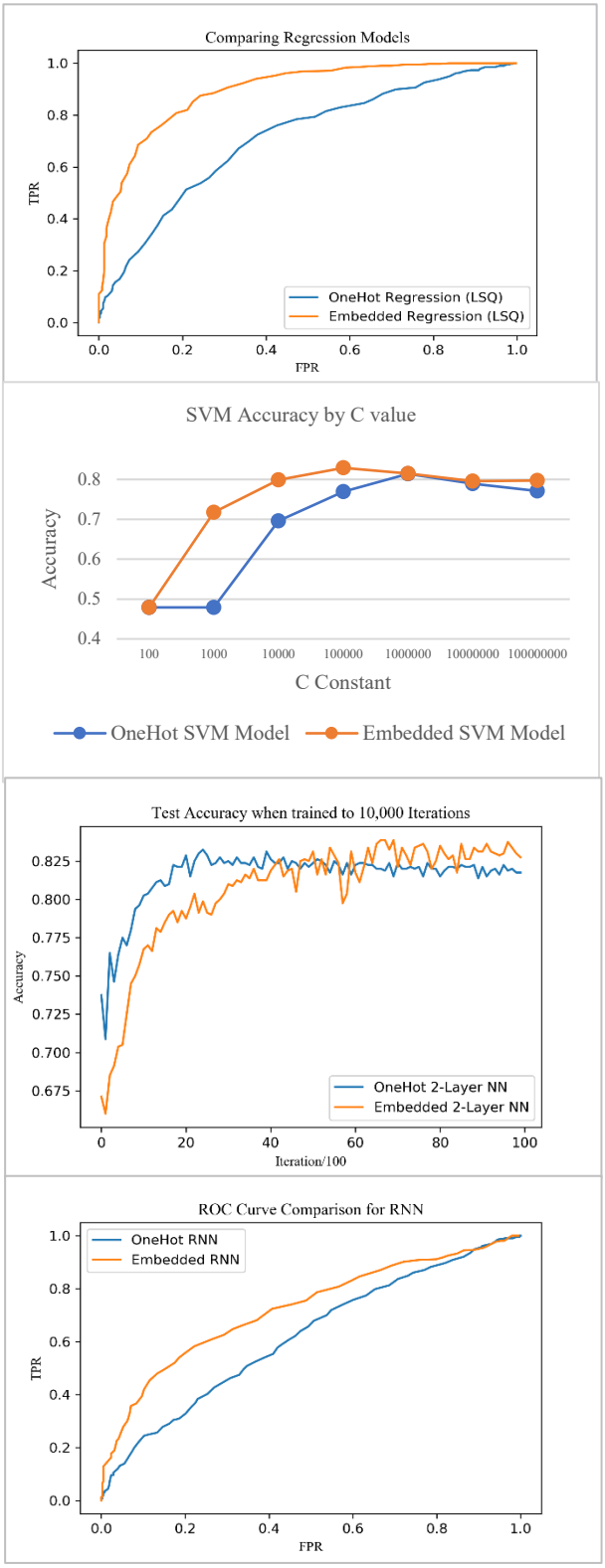

Figure 7:10 - Comparing One Hot to Embedded Models

Conclusion:

Overview

Embedded Vectors can provide information determined from high word count sets that can improve results across the board for various models. This additional information improves the results of trained and untrained methods for classification of text. Improvement occurs for all models but is most noticeable in linear models, where space separation is important, and RNNs, where order of different similar words might have different associated meanings. This concept is similar to transfer learning where a NN is initialized with preset weights from a model trained on a similar problem.

The existence of a training set is also largely impactful on categorizing sentiment. Besides it being impossible to train eminent models such as RNNs without a training set, you can also derive important contextual and intention information. However, if a training set isn't present, its best to use existing models to help determine associative sentiment.

Improving the RNN

I'm pretty sure I understand the primary issue in the RNN and how it could be improved. Essentially, a single idea is contained in a sentence. My dataset is pruned to improve classification on the other models but would be better suited for the RNN organized as a large collection of sentences. Each sentence is taken from the training set and classified by sentiment of its parent review. Naturally, this could cause issues where a positive sentiment sentence lies within a negative review. However, most sentences in a negative review have a negative sentiment. With much shorter passages we'll be able to do many more iterations and fine tune a model that doesn't have to operate for nearly as long. Finally, when want to classify test reviews, we get the sentiment of each sentence included in the review and sum the logits together.

Even using an LSTM means we can still lose information over large sets if we haven't done enough training to correctly train neurons. I think analyzing things on a sentence by sentence basis means we don't need as many iterations or need to worry about vanishing gradients. I'm a little frustrated I didn't think of this method before implementing everything as I'm sure it would improve results.

Other Modifications to Improve Results

The simplest method of improving results across the board would be to find a bigger training set. In any application problem in the real world, data collection is more than half the problem.

Another way to improve results would be to include additional information in the form of part of speech tags on each of the words. I'm not quite sure how embedded models are set and if it's possible to include multiple representations of a word based on its part of speech, but that information would be useful for all models in addition to the RNN as it would remove incorrect associations with each of the words. I think if you're trying to maximize precision, even using a part of speech tagger and using the information it provides could improve results.

In a situation where you're not using an RNN, perhaps you can include transition terms like a Markov Chain, but simple vector counts like the words were included as one hots. You could maybe find a few highly indicative yet common word transitions. This would be better with more training data and RNN is designed to do this automatically, but it would be an interesting work around.

References:

1. https://www.kaggle.com/alvations/word2vec-embedding-using-gensim-and-nltk - Used to figure out embedded models and how to get the pruned Google News Set.

2. https://scikit-learn.org/stable/tutorial/basic/tutorial.html - SVM

3. Guillaume Chevalier, https://commons.wikimedia.org/wiki/File:The_LSTM_cell.png – Included to illustrate the interior of an LSTM cell.