Contents

1 Introduction

This repo contains my exploration of basic natural language processing for text sentiment analysis. The framework and dataset come from the Kaggle Movie Sentiment Analysis competition, based on user reviews for movies on rotten tomatoes. The dataset is assumed to be complete and labeled, though some exploration of unlabeled data is conducted for fun. The basic approach is to create a bag of words from the provided phrases and train these on the associated sentiment scores. The main components of the code are to first "clean" the phrases to be analyzed by removing punctuation, removing "stopwords" (from nltk stopwords set), and creating stems using a stemmer, then to apply a vectorizer to the phrases, and finally to train a model on the word vectors vs sentiments and then predict using that model. A number of different models are compared.

2 Summary of Workflow

2.1 The Dataset

The dataset consists of 156060 phrases and sentiment numbers as well as some other less important labels such as "SentenceId" that can be ignored. The phrases are pieces of a sentence, generated from the original with replacement, which range from one word to most of the sentence. Importantly, for a

given sentence the sentiment score can be very different even for phrases from the same sentence. The sentiments are scores taking values in $\{0, 1, 2, 3, 4\}$. The distribution of the sentiments is concentrated at 2, corresponding to a "3 out of 5" or "neutral" rating, with over 50% of the sentiments being viewed as neutral (see Figure 1).

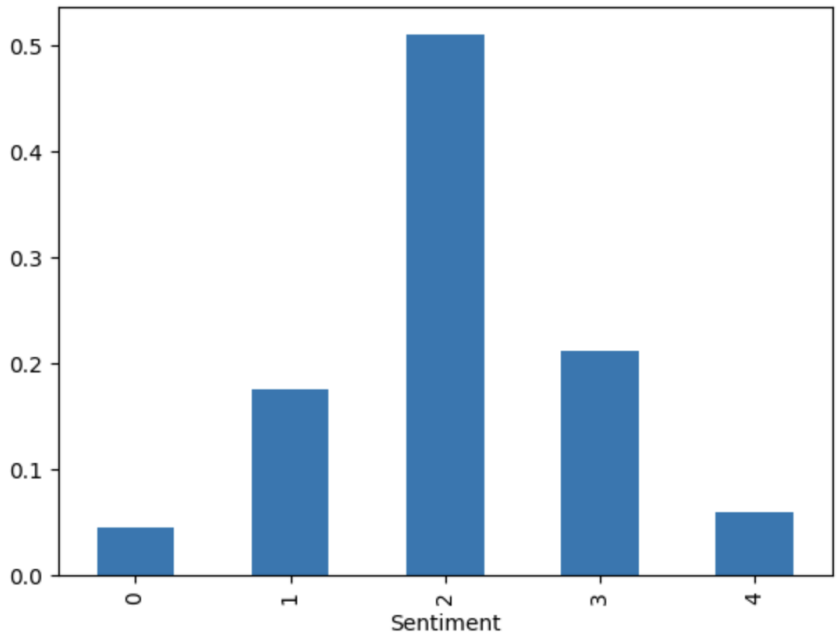

Figure 1: Distribution of Sentiments (percent of total data).

Many of the phrases consist solely of so called "stop words", such as but, it, this, etc., which are commonly deleted from training sets used for NLP tasks before embedding. However, in this dataset, this would completely delete many of the phrases and result in a large loss in data. We can demonstrate quantified losses from deleting stopwords in terms of accuracy / precision etc., as well as model stability (see later sections).

2.2 Vectorizing

Two different methods for converting the words in the text to vectors are utilized. The first is the count vectorizer and the second is the "term frequency inverse document frequency" (TFIDF) vectorizer. For the first case, the phrases are fitted and transformed according just to the word count. For the second, the TFIDF weights are applied to the words. Roughly, TFIDF is a statistical measure of how important

a word is to a whole corpus of data. The formula is

$$"tf \cdot idf" = t/Terms \cdot \log_e(Docs/t_{docs}$$

where t is the number of times a term appeared in a document, $Terms$ is the total number of terms in the document, $Docs$ is the total number of documents and t_{docs} is number of documents with term t in them. What we will see is that the TFIDF vectorizer creates a much larger number of features than the count vectorizer does and the subsequently trained model may be overfitting to the training data as a result. However, we will see that the TFIDF vectorizer does improve many metrics for the multinomial naive bayes classifier, but these improvements may be superficial.

2.3 Models Studied

In this implementation, I mainly compare 4 models [1], Gradient Boosting, Multinomial Naive Bayes, Logistic Regression, and Random Forest Classifiers.

1. **Gradient Boosting** Gradient boosting was slow to train and had relatively poor precision output compared to the other models. Mechanically the classifier fits 5 (because there are 5 sentiment classes) regression trees on the negative gradient of the multiclass loss function. There was some opportunity for parameter tuning here that was not taken up as the results were about 10% worse across all metrics which is unlikely to be overcome through clever choice of parameters (at least without much more computational power, but all the models should be better if we can do more brute force training save possibly the naive bayes classifier).

2. **Multinomial Naive Bayesian classifiers** are based on Bayes' theorem and the thinking in Bayesian Statistics. The basic algorithm for Bayesian inference goes like

   (a) Start with a prior distribution, say probability of landing in each sentiment "bin" in the above problem.

   (b) Compute likelihood of a phrase being a given sentiment conditioned on the observation of the sentiment. In a sense, this is the new 'strength of belief' in the probability of a phrase being associated to each sentiment once we see more sentiment observations.

[1] You'll realize I really studied only 3.5 if you keep reading and notice that we really only trained random forest classifiers long enough to determine that they 1) took forever to train on this dataset (which is large ish) and did not perform nearly as well as the NB or log regression models

(c) Multiply the quantities from 1. and 2. together and update the prior distribution with this new distribution.

As we can see from the algorithm, roughly speaking Bayesian inference seeks to find a *probability distribution* for the data based on prior beliefs about the data.

The classifier is referred to as "naive" because it assumes that features are independent of each other. In a natural language processing task, this is an ok assumption as if we ignore trivial words always present in english sentences (the so called "stop words" often removed in the cleaning step), we would see that sometimes the presence of a given word is affected by another and sometimes it is not. With sentiment analysis, the classifier is trying to link key words to the sentiment associated with the phrase it is contained in. These key words should be roughly independent if they truly capture the basic feelings being conveyed by the phrase.

The multinomial designation comes from the fact that the discrete features are assumed to have multinomial distributions. This means we use the multinomial distribution to estimate the likelihood of seeing a set of word counts in a given phrase when we compute the likelihood in step 2. above.

The multinomial NB classifier is common in and well suited for sentiment analysis because it 1) is easy to implement, 2) robust to irrelevant features which with a large corpus of documents are bound to occur, and 3) is highly computationally efficient which is an advantage over random forests for example in that it can be fit to the large datasets involved in NLP quickly.

3. **Random Forest Classifiers.** While in principle one may think this model would perform well at a task of classifying a 1-5 sentiment from a large corpus of phrases, these models are very slow to train on the high dimensional datasets being used and in the instances I was able to implement did not actually perform any better than the other models.

4. **Logistic Regression** A basic regression model that fits to categorical data. In this case, this would be a multinomial logistic regression model. This model took longer to train than the Naive Bayes classifier, but performed slightly better. In particular, when we compare the metrics for the different models, we will see that the logit model had better precision and recall than the naive bayes model. One reason for this better performance may be that the logistic regression model does not make the assumption that the features are statistically independent of each other, potentially allowing context to better drive the classification. The better performance was not by much, and while logit does not

make the assumption of independence it does assume a low collinearity so the context it is able to utilize seems to be slight. Nevertheless, perhaps this is driving better classification.

3 Output Analysis

We will focus on the results from the logistic regression and multinomial naive bayesian classifiers, as these were the most easily examined and accurate models.

Summary of Cross validation:

- (NB no tfidf) Cross-validation mean accuracy 59.91%, std 0.11.
- (NB with tfidf) Cross-validation mean accuracy 58.82%, std 0.11.
- (logit no tfidf) Cross-validation mean accuracy 62.74%, std 0.20.
- (logit with tfidf) Cross-validation mean accuracy 62.11 %, std 0.20.

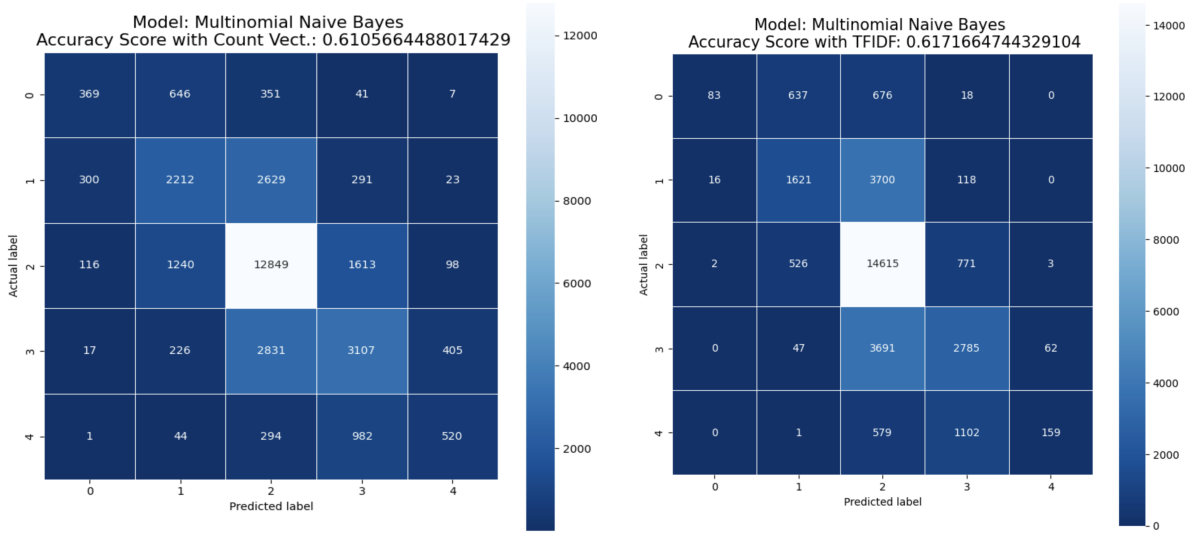

Figure 2:

For the Multinomial NB model, tfidf does actually improve precision substantially (by about 10%). However, we see that with the count vectorizer, the confusion matrix demonstrates a roughly normally distributed set of predictions in that its decay off the main diagonal (on the main diagonal, predictions are the same as the true values and these predictions do not contribute to the error) looks like a 2 dimensional Gaussian surface. With the tfidf vectorizer, it appears that the classifier wants to predict a label of '2' more often for the phrases, meaning it is treating more phrases as 'neutral' than the same model with

the count vectorizer. This may actually be helping this model make up for it's poor accuracy on other sentiment scores, as it has a couple thousand more true positives for sentiment 2 than the model using the count vectorizer. However, this improved accuracy would just be a quirk of the model biasing itself towards predicting the most frequent sentiment value in more cases, superficially inflating it's precision score.

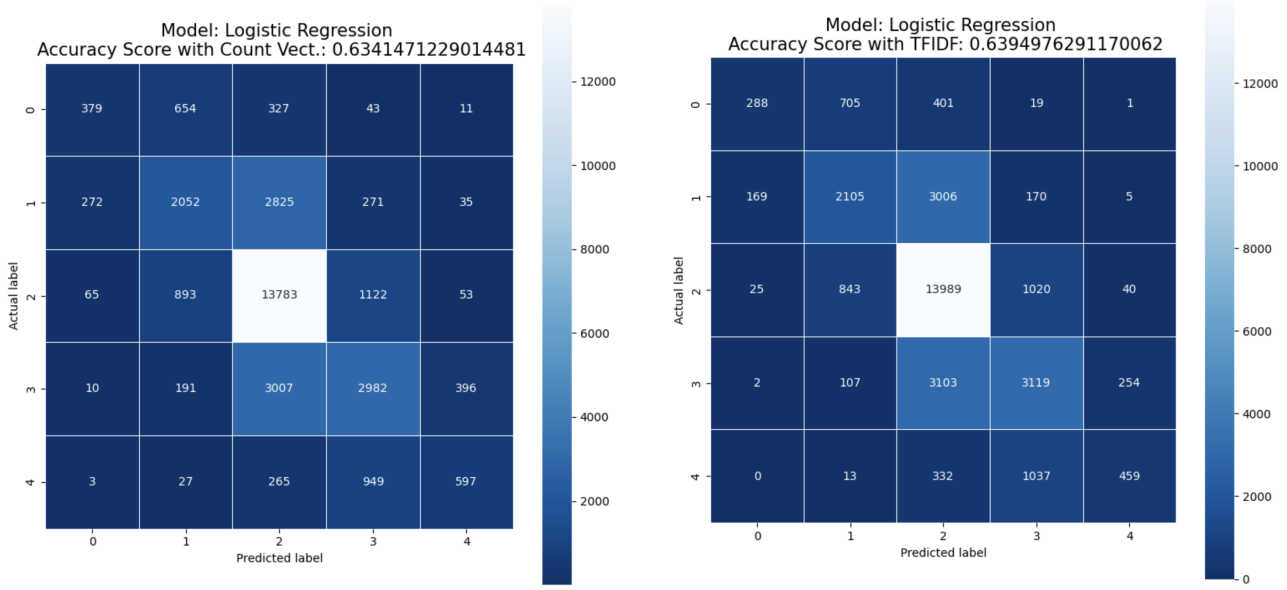

Figure 3:

For the log regression model, the results are essentially identical with or without tfidf. The confusion matrices also look very similar. It may be that log regression also has a slight preference for predicting sentiment 2 for both the vectorizers, which is interesting, but the results are essentially identical between the two vectorizers. It is also interesting that log regression doesn't seem to differ between vectorizers but this is a discussion that will be left to percolate for later.

The tfidf vectorizer essentially extends the count vectorizer by multiplying the counts by the inverse document frequency weights defined above. This prevents all words from being treated equally, and instead penalizes words that occur very often. As mentioned in the above there are a lot of phrases consisting solely of stopwords, which likely occur often. The IDF weight would therefore likely punish these words. However, since there are about 18-20 phrases / sentence, the frequency of every word is likely very high, so it's plausible that tfidf is not making as big a difference as it could be. It does appear from the cross validation scores above that the count vectorizer is performing slightly better, so it's likely that the tfidf vectorizer is playing a similar role as deleting the stop words would have played.

We can make a small but interesting observation by looking at the mean cross validation accuracy and standard deviation. The cross-val accuracies are consistent with the above discussion regarding the training accuracy of the models with tfidf vs cv vectorizers. What's interesting however, is that if one deletes the nltk stopwords set from the phrases, the resultant standard deviations for the cross validations one would get would be much larger, indicating less stable models. This is interesting because it's saying that the stop words in the phrases are a stabilizing input for the model, providing important info for both accuracy and consistency with predictions. This makes sense given that in the EDA we saw that a large number of the phrases are only 1-2 words, and if those words are stopwords, then deleting them deletes are large amount of information about how to assign sentiments.

The test accuracies on the kaggle test set were $\sim 56\%$.

3.1 Possible Improvements

There is (as always?) potential for hyperparameter tuning. Particularly how many features we use with the count vectorizer. But it's unlikely this will make a massive improvement in the accuracies. To make meaningful improvements, deep learning is likely necessary.

I plan to apply these methods as well as a later implementation of a deep learning approach to the fake news dataset. The questions of which model is best and which vectorizer is best (as typically tfidf is considered better as I understand it) may give different answers for this dataset.

4 Repo Structure

The repo consists of a python file containing the main pipeline used above, with options to specify count vs. TFIDF vectorization as well as classification model clearly indicated. There is this readme file, as well as a jupyter notebook showing the implementation described above in a step by step fashion. The notebook includes some of the same discussions the models and vectorizers as mentioned above, and should serve as a visual aid as well as an example (it is being included in lieu of a presentation style exploration).

A Sources

For more on NLP and word bagging. "Bag of Words Meets Bag of Popcorn" Kaggle tutorial

For more on stemming ** IBM 'What is Stemming?'

For the original Kaggle competition:

Addison Howard, Phil Culliton, Will Cukierski. (2018). Movie Review Sentiment Analysis (Kernels Only). Kaggle Link