SENTIMENT ANALYSIS ON MOVIE REVIEWS

Rishita Roy Samir Shingane Justin Yu

1 Abstract

A movie review predictor will indicate certain words and phrases to be either good or bad. With a different level of degree of how bad or good a words and phrase can be, we know that this can be linearly classified. Based on this, we decided to try SVM to see what our training accuracy was.
We have also tried a lot of different classifiers as well to test training accuracy including logistic regression and Naive Bayes. Out of all the classifiers, we stick with SVM because the outcome came with the best accuracy
We also tried to implement Ada-boost on these classifiers, but the accuracy always dropped. We believe this happened because our classifier comes as a strong learner, and ada-boost is only very effective on weak learners.
We ended up using SVM, which for the training set gave us a 73.8% accuracy, a precision score for the 5 labels to be [0.72, 0.68, 0.77, 0.68, 0.72], F-Measure to be [0.6, 0.63, 0.82, 0.64, 0.61], and Recall to be [0.51, 0.59, 0.88, 0.62, 0.53].

2 Tools Used

We used **sklearn** as our main library to use different classifiers. Sklearn gave us the tools to use to extract data from the data set, classifiers to use to test for accuracy, and metrics to account the results that we were looking for. We also used **numpy** which allowed us to get our data in a certain format and fashion that was easier for us to analyze and understand. We also used **nltk** to import stopwords to allow us to filter the data we want. We had a list of words we thought should not be accounted for in our training set, and this package gave us a better list than if we put in a list ourselves. We also used **pickle** to de-serialize the python object taken in as a file and putting it as a string. This allows us to store the vectorized words from the training to be used during test time. We used **csv** because the input file for all our data is in a csv file, which we need to parse correctly in order to get the data we want. We used **math** so that we can truncate our predicted labels to be what they were supposed to be. This was used when we were finding out what the predicted labels were.

3 Data Pre-Processing & Feature Extraction

We removed a few things from the training set in order to keep what we believe to be a better result. We first removed low frequency words. We think these words are not impactful if they only show up a few times, and do not necessarily impact the result that severely. We then also removed stop words that do not give us useful information. These include conjunction words such as "the, a, in" because these do not help us. We also made all the words lowercase, because we do not want capitalization to affect or training data. Words that are capital and not to us are the same thing. These things are all handled by sklearn.
To get these features, we tried a few methods to achieve this. We tried bag of word which stored all the vocabulary together and we assigned weights to each one. We then used count vectorization to see the frequency of the vocabulary we were given. Finally we used tf idf vectorization so that we give each vocabuilary an index and evaluate them by index.

4 Approach

Once we had tackled preliminary data pre-processing and feature extraction, we looked to implementing different learning models. As recommended by our professor and TA, we chose to start with using some pre-existing modules from the `sci kit learn` library. There was a bit of a learning curve with ensuring that our data was in the correct format and understanding the breadth of the library's offerings, but once we were comfortable, we began our training and testing.

We knew we wanted to testing out multiple different models, so we began by looking at ones we had covered in class. Decision trees and K-nearest neighbors did not seem as if they would work well on this data set, as we knew we would be extracting many features and this was a multi-label classification problem. We chose to begin by implementing logistic regression as our first classifier. We used several different versions of it, including vanilla regression, regression with a bias term, with regularization, and with both, but noticed there was not a signifiant difference between the train and test CV scores between them. They all had accuracies around `0.683192` for training scores and `0.625135` for test scores.

We then chose to implement multinomial Naive Bayes, Perceptron with a random state, and linear SVM. For Naive Bayes and linear SVM, we chose these particular modifications because after some research, these would perform the best on large data sets and with multiple labels. The following are the 10-fold CV results for these other classifiers

Average across 10-fold CV	**Naive Bayes**	**Perceptron**	**SVM**
Train Score	`0.632251`	`0.653539`	`0.727624`
Test Score	`0.583905`	`0.563775`	`0.626643`

Out of these other three classifiers, we noticed that SVM did quite a bit better than the other classifiers, and in both cases, outperforming in terms of CV accuracy.

Once we decided to focus on SVM, we tested out several parameters, such as the C value and the number of total iterations. We were initially excited, as increasing the C value continuously gave us higher training accuracies before looking at the CV results. Once we realized that this was because the higher values were simply overfitting to the training set, we decided to reduce C to minimize these effects on the test set values.

5 Experimental Set-Up

For this project, we wrote everything in Python 3 and tried to follow a format where any different kind of approach we would try would be contained within its own method, and everything would be called from a "main" method. This allowed us to easily test out different approaches easily by removing or adding method calls to our main method.

When we were implementing our different classifiers, we wanted to be able to see the performance statistics to compare and gauge which one specifically we would potentially want to use for our final result. This lead to the creation of a function called `printPerformance`, which would take in a list of expected labels and a list of actual labels and print out performance metrics. This can be seen in Figure 1, where the output included the 5x5 confusion matrix of labels, overall accuracy, and the precision, recall, and f1 measure by each label.

```
============================================================
Naive Bayes + tfid
============================================================
CONFUSION MATRIX:
[[  340  1883  2601    90     0]
 [   63  6030 12438   516     3]
 [   18  1447 52038  2361    21]
 [    1   238 12583 10027   111]
 [    0    19  2257  3610   547]]
ACCURACY: 0.6314604273081782
RECALL: [0.06919007 0.31653543 0.93116221 0.43671603 0.08503031]
PRECISION: [0.8056872  0.62701466 0.63525276 0.60389063 0.80205279]
F1 SCORE: [0.12743628 0.42069278 0.75525754 0.50687494 0.15375966]
```

Figure 1

Once we had this implemented, we began preliminary comparisons of our classifiers by using the entire given set to train our model, and testing the model on the same set. While this gave us an idea of which classifiers to consider over others, this couldn't be a fully accurate measure of performance as we testing on the data we trained with. After realizing this, we tweaked our existing methods for our classifiers to take in an additional testing set, and we also incorporated the sklearn cross validation model to more easily test across different folds of the data.

After feature extraction and training our models, we had to once again tweak our set up to incorporate the given test set. Obviously we wouldn't be able to run our performance metrics as they were dependent on having the actual set of labels for the instances, so we temporarily removed calls to print performance, and created a new function that would write the predicted labels out to a file. This file was written using the `csv` library to match the expected output format.

6 Results

We ended up using the `SVM` classifier, specifically the `Linear SVM` classifier. We chose this classifier because it resulted in the highest accuracy and cross validation score of our implemented models. To avoid overfitting and to keep runtime reasonable, we set C, which controls the amount of overfitting, to 5. This led to a higher accuracy and cross validation score, without too much overfitting of the data. Below are the metrics and cross validation results for the `Linear SVM` classifier.

```
============================================================
SVM
============================================================
CONFUSION MATRIX:
[[ 2538  1734   535   101     6]
 [  733 11287  6287   702    41]
 [  205  3027 49221  3277   155]
 [   24   531  7161 14149  1095]
 [    3    47   494  2473  3416]]
ACCURACY: 0.7379121583273832
PRECISION: [0.72452184 0.67887646 0.77272442 0.68346054 0.72480373]
F-MEASURE:  [0.60306523 0.63275031 0.82321066 0.64811507 0.61295532]
RECALL:  [0.51648352 0.59249344 0.88075512 0.61624564 0.53101197]

10 folds, test score:  [0.62557203 0.62868387 0.62978217 0.63093822 0.62120103 0.62418749
 0.62400439 0.6175959  0.63050444 0.62556074]
train score:  [0.7457484  0.74415151 0.74421254 0.74575099 0.74556032 0.74624437
 0.74606129 0.74585787 0.74407795 0.7452883 ]
```

Figure 2

When compared to the Naive Bayes classifier above, we see that the accuracy alone is over 10 percent higher, along with drastically higher values for precision, F-measure, and recall across the board. Below, we can also see the results of 10 fold cross validation on the Naive Bayes classifier. We can clearly see that using Linear SVM has increased the cross validation scores across the board, which means that using Linear SVM will result in higher test scores. These higher test scoreas are partially in part to Linear SVM's ability to look at and analyze the interactions between features, which will lead to higher cross validation scores and general metrics.

```
10 folds, test score:  [0.58347062 0.58475197 0.58630789 0.5793135  0.57826803 0.57969422
 0.58280692 0.58216607 0.58436327 0.58363087]
train score:  [0.6353798  0.63466781 0.63534928 0.63491563 0.63427857 0.63551297
 0.63528921 0.63515699 0.63535024 0.63574691]
```

Figure 3

7 Ideas for Future Work

An idea that we had for future work was to implement a `Random Forest` model. Unfortunately, the idea to implement this model slipped our minds until the final day, merely a few hours prior to the deadline for the predictions. During our hurried research, we saw that the classifier depends on the depths of the trees that make up the forest. From the sklearn model, we saw that the default tree depth was 100. When implementing this, however, the runtime was impractical for the purposes of the assignment, exceeding over thirty minutes before the process was ended. Reducing the tree depth to ten or twenty did improve runtime

to within ten minutes, but we were unsure as to how this would affect the results of the model in terms of over or underfitting. The results of this model did result in accuracy greater than any of our other models, but we unfortunately ran out of time to properly research and implement this model. Another idea we had for future work was to process the data more and add more features, such as parse the data for phrases, rather than extracting the data a single word at a time. This would allow us to predict the sentiment with greater accuracy, as certain phrases, such as "not bad", mean the opposite of their individual words.