CS434 Final Project Report (4-page limit)

Joshua Diedrich, Grayland Lunn, Race Stewart

1 Introduction

When beginning the project, we knew that we wanted to start with Naive Bayes. We also knew that our approach to the neutral tweets would be to simply return the entire tweet, as suggested. The three methods we tried were Naive Bayes, logistic regression, and LinearSVM (support vector machine). For all three, our general approach was to train a classifier on data in order to predict the sentiment. Following this step, for each subset of text in a tweet, we would get the confidence score of the sentiment and keep the highest. These general steps were followed in our implementations of LinearSVM, Naive Bayes, and logistic regression. We then took some steps to try and improve our scores by applying weights and metadata to the training. Using this implementation, LinearSVM yielded the best results (with an accuracy of 0.661 on Kaggle) and NB logistic regression both yielded worse results.

1.1 Team responsibilities

Joshua was in charge of implementing the Naive Bayes initially, then Grayland and Race went from there to test LinearSVM and logistic regression respectively. We all collaborated on the "other" methods and the report.

2 Method 1

The first method implemented was Multinomial Naive Bayes. We wanted to get a baseline idea of accuracy, and this was a classifier we are all familiar with. We started by training our classifier (after splitting into testing and validation sets) to predict positive, negative, and neutral sentiments. Then, for each subset of text in the tweet, we calculated the NB prediction probably for that tweets sentiment. We returned the subset with the highest of these values for positive and negative sentiment, and returned the entire text for neutral sentiment. The idea here was that the subset of text which contributes the most to a prediction of the sentiment would reflect the sentiment best. This approach is similar to the provided notebook titled "A simple solution using only word counts" [1], but instead of using word counts when deciding our best subset, we use the Naive Bayes prediction confidence.

To implement Naive Bayes properly we used a number of pre-processing methods. The first was to use CountVectorizer to transform our training and testing

set into feature vectors. We also applied a text cleaning alongside the vectorizer. For this we used the method from the notebook mentioned above [1]. It entailed removing casing, square brackets, links, punctuation, and words containing numbers. To further normalize the feature vectors, we used sklearn's tfidf transformer. To actually fit the training data, we used sklearn's MultinomialNB. After training and validating we tuned alpha along with the max-df, min-df and max-features values for the vectorizer to increase the Jaccard score.

3 Method 2

The second method implemented was logistic regression. The idea here was to use logistic regression rather than Naive Bayes to predict sentiment. Logistic regression was chosen because the group was experienced with it, and because some research showed it might perform better text classification than Naive Bayes. The pre-processing of the data was the same method used in the Naive Bayes classifier. We used the CountVectorizer and TFIDF Transformer to make feature vectors. The training of the features was then implemented using sklearn's LogisticRegression. We then tuned the classifier by adjusting the regularization, tolerance, min-df, max-df, and max-features to increase the Jaccard score.

4 Method 3

Our third method implemented was LinearSVM. This method followed the same idea as Logistic Regression. After reading online we thought that LinearSVM might actually perform better than the other two for text classification. We thought that SVM would perform well due to the high dimensional space of our data. One drawback we were worried about was over-fitting, as the training data is much larger than the number of features. The pre-processing for this method was the same as the others. We one again used CountVectorizer and TFIDF Transformer to make feature vectors. The training was done using sklearn's LinearSVM. We then tuned the classifier by adjusting the regularization, tolerance, min-df, max-df, and max-features to increase the Jaccard score. LinearSVM was ultimately the best predictor out of the three methods, and we decided to continue with it when trying to optimize our score.

5 Other methods

After training and testing our three base classifiers, we went back to look at how they could be improved by applying specific ideas about the problem statement and training data. We will use this section to show some "creative" explorations we tried to improve our Jaccard score with.

5.1 Removing Neutral Features

The first idea that we had was to actually lose the neutral features in the training set. Since we throw these values out as the entire text when predicting, we figured that training on only positive and negative sentiments might give our

classifier a better idea of which words were important. This approach greatly increased our sentiment predictions, but also greatly decreased our Jaccard score. We think that this is because the classifiers got better at predicting positive and negative sentiment, but wound up choosing too many words from each text. Having a good idea of what words are neutral helps keep unnecessary words out of your prediction, because they will have a close to 0 score.

5.2 Jaccard Weighting

Our next approach was to try and apply a weight to our features based on the Jaccard score of the selected text. The idea here was that features with a low Jaccard score would be more meaningful, as they used fewer words to represent the sentiment of a large sentence. We implemented this by calculating the Jaccard score for each feature, and applying the inverse as sample weights to the classifiers. This approach slightly worsened our accuracy for both sentiment prediction and final Jaccard scores on the testing data. When looking at the weights, we saw that the Jaccard scores were were very widely dispersed. This left some features with very large weights and others very small.

5.3 Selected Text Weighting and Class Weight

The next approach we took was weighing the features by the size of the selected text compared to the original text. This is the same general idea that Jaccard score weighing aimed to do, but it is less complicated, and would give less dispersed weights. We implemented it by calculating the ratio of text/selected-text for each feature and applying them as sample weights to the classifiers. Like the Jaccard score, this approach initially lowered our training accuracy, but by a slightly smaller amount.

We realized that this method was actually dampening the effect of our neutral features. Since neutral features almost always just contain the original text, we were giving them a weight of 1, while the positive and negative features were getting higher weights. We wanted to still weigh our samples based on the selected text size, but also still train on neutral features. Our solution was applying a higher class weight to neutral sentiments. This allowed the neutral sentiments to still have a strong effect on training, while the features with a small selected text to original text ratio could also be weighted higher. This method is what ultimately improved our classifier the most.

6 Results and discussions

Internal Jaccard Validation Matrix

Classifier	Standard	Weighted Sample	Weighted Class and Sample
Naive Bayes	0.614	.6006	Not applicable
Logistic Regression	0.630	0.631	0.635
LinearSVC	0.652	0.647	0.671

The table above shows the scores of each of our classifiers when trained on the training data, and when trained on the weighted training data. The class

weights above were given as {Neutral: 0.6, Positive: 0.4, Negative: 0.4}. The sample weights were the ratio of text/selected-text from the training data. As you can see, LinearSVM outperformed Logistic Regression and Naive Bayes. The standard classifier for LinearSVM had an acceptable accuracy when run on the standard training data. When just the sample weights were applied, we actually saw a decrease in accuracy. As discussed above, we believe this was because neutral samples maintained a weight of 1, so they had a dampened training effect. Applying a class weight to neutral samples seemed to correct this, and resulted in an over increase in accuracy of about 0.2. We believe this is because we were able to add emphasis on samples with a small selected text compared to the original, while still allowing the neutral samples to be considered.

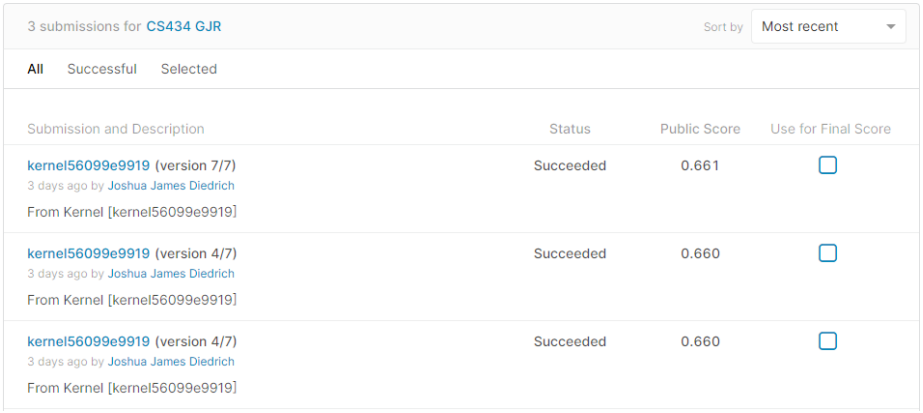

Figure 1: Kaggle Screenshot

Our final version submitted on Kaggle resulted in a Public Score of 0.661, while internal validation testing resulted in a score of roughly 0.67. This decrease from internal validation was probably due to more variation in the much larger testing set. It certainly seems that LinearSVM was the best way (that we tested) to implement a classifier for this data.

References

[1] Nick Koprowicz. "A simple solution using only word counts". In: (2020). DOI: https://www.kaggle.com/nkoprowicz/a-simple-solution-using-only-word-counts.