**CS 178: A Comparative Study of Classification Methods regarding Sentiment Analysis**

**Rebecca Park**

**Nicholas Hosun Song**

**Summary:**

In our study we tried to work on a comprehensive comparison of various classification methods to determine their efficacy in sentiment analysis targeting movie reviews. We selected a diverse array of models – Naïve Bayes, Linear Support Vector Machines, Decision Trees, Feedforward Neural Networks, and Convolutional Neural Networks – each providing us with different approaches to classification. Our goal was to identify which model most effectively and accurately classifies reviews as positive or negative, considering the complexities that occur in natural language processing. Lastly, we used the stacking method to combine all of these models to see compare the results as well.

**Dataset:**

We utilized the Large Movie Review Dataset, a collection of 50,000 movie reviews evenly split for training and testing. Originally rated on a scale from 1(negative) to 10(positive), we decided to use the pre-tokenized Bag of Words (BoW) features for our analysis. The BoW, is in a LIBSVM sparse-vector format, gave us a foundation for our study which allowed us to focus on the comparative analysis of classification methods rather than on text preprocessing because the presence and frequency of words provide us with a strong indication of which target variable we need to align with our words.

**Models:**

***Naïve Bayes***: These models are probabilistic models which apply the Bayes’ theorem with the assumption of independence between features. They are well suited for text classification tasks due to their simplicity as well as effectiveness in dealing with high-dimensional data. We implemented this using Scikit-learn.

***Linear Support Vector Machines***: Linear SVMs are a model that finds a hyperplane in a high-dimensional space to separate different classes. The linear SVM was a model we chose because of its ability to handle large feature spaces like in our dataset. The main hyperparameter we decided to tune was the regularization parameter C and we also used Scikit-learn.

***Feedforward Neural Network***: A linear neural network where the inputs are fed to the outputs through different layers that hold a series of weights and hidden nodes. The data is fed through one direction in this type of Neural Network, and we chose to go with this model because it can capture complex patterns in text data. The hyperparameters we chose to go through, and test were the number of neurons in each hidden layer, the dropout rate, and the learning rate for the optimizer. We used the Keras library to create this model.

***Decision Trees***: Decision Trees are tree structures where an internal node represents a feature (or attribute), the branch represents a decision rule, and each leaf node represents an outcome. In the context of text data, they help in making decisions based on whether the word is present. We first tried using different hyper parameters like the minimum sample splits, minimum sample leaf, and maximum depth. We implemented this using Scikit-learn.

***Stacking****:* Stacking is an ensemble method. Multiple well performing “base models” are trained on the same set of data, then their results are used as training data for the “meta model”. We wanted to see how the individual models performed in comparison to a combination of all of them. We used the models that were implemented with Scikit-learn (Naïve Bayes, Linear Support Vector Machines, Decision Trees) as our base model, and implemented stacking through Scikit-learn as well.

**Experiment:**

For our experiments we used a variety of splits for our data depending on the model. Generally, for all the models, we used our two BoW files split into 25,000 reviews each. For training, we set aside 5000 reviews for validation, and the rest of the 20,000 reviews were used for training. After training we used the rest of the 25,000 reviews from the test dataset to test our model.

Naïve Bayes - Nicholas:

For a baseline we chose to use a Naïve Bayes model because it works great for dealing with high dimensional data as well as with text classification. We assume independence; therefore, this model does have an issue distinguishing patterns and figuring out contextual relationships between words.

The training error rate that we observed with this model was 0.1000 and for our test error rate we got 0.1864. This test error rate is what we will use to compare our final chosen models to see how well more complex models that can consider more factors perform against this simpler model.

Linear Support Vector Machine - Nicholas:

For this model we chose to use a linear kernel SVM which tends to work well when doing binary classification tasks. We chose to use L2 regularization as it doesn’t force feature selection as aggressively as L1. We tuned the regularization strength parameter to prevent overfitting and to improve the generalization of the model. This parameter encourages the model to have smaller weights/coefficients for each of the features, which will make the model less sensitive to the noise in the training data. Doing this helped us control the margin between the decision boundary and the training points to find the decision boundary that maximizes the margin between the labels while also correctly classifying the training examples.

|  |  |  |
| --- | --- | --- |
| Regularization Strength ~ C | Validation Error Rate | Test Error Rate |
| 0.00001 | 0.2356 | 0.2308 |
| 0.0001 | 0.1446 | 0.1557 |
| 0.001 | 0.1074 | 0.1203 |
| 0.01 | 0.1052 | 0.1234 |
| 0.1 | 0.1196 | 0.1419 |
| 1 | 0.1278 | 0.1526 |

A graph with a blue line and a red line

Description automatically generated

From the test done above, we can see that the regularization strength starts by underfitting, and we find the ideal regularization strength at .001. We see that as we decrease the strength by increasing the value of C, we start to overfit our data as we are capturing too much of the noise causing our error to go up.

Feedforward Neural Network - Nicholas:

We chose to use the Feedforward Neural Network because it allows us to create a neural network that is not too complex, therefore it won’t be as prone to overfitting. We also implemented early stopping which is a part of the Keras library to help prevent overfitting by reading the losses at each and previous epoch and making sure the loss doesn’t go up. The main hyperparameters we chose to adjust were the number of layers, number of neurons in each layer, the dropout rate, and the learning rate. The number of hidden layers determines the depth of the neural network. Each of these layers can learn different features or representations of the data while allowing us to see how complex our network needed to be to learn our dataset. The number of neurons determines the capacity of the layer to learn different aspects of the input data, we also changed this parameter for similar reasons to the number of hidden layers. The dropout rate is a regularization technique that is used to prevent overfitting by randomly setting to zero a proportion of the neurons in a layer during training. We wanted to see the effects of regularization and how impactful it was to our model. The final hyperparameter we changed in this testing was the learning rate for our optimizer, which controls how much the weights in the network are adjusted with respect to the loss gradient. The following shows the 7 tests with the highest accuracy out of 100 tests.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number of Layers | Neurons per Layer | Dropout Rate | Learning rate | Test Accuracy |
| 2 | 64 | .5 | .0001 | .8793 |
| 2 | 64 | .3 | .0001 | .8782 |
| 2 | 128 | .5 | .00001 | .8804 |
| 1 | 32 | .5 | .0001 | .8772 |
| 2 | 128 | .1 | .001 | .8763 |
| 1 | 64 | .3 | .0001 | .8750 |
| 2 | 128 | .3 | .00001 | .8828 |

Decision Trees - Rebecca:

For this model, we decided to adjust multiple hyperparameters to see which one contributed the most to improving the error rates. The parameters we chose to adjust were the minimum number of samples required to split, minimum number of samples required to be a leaf node, and maximum depth.

A graph of a line graph

Description automatically generated with medium confidence A graph of a graph

Description automatically generated with medium confidence A graph of a line

Description automatically generated with medium confidence

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hyperparameter | Best Parameter Setting | Training Error | Validation Error | Testing Error |
| Minimum sample split | 400 | 0.219 | 0.260 | 0.260 |
| Minimum sample leaf | 50 | 0.229 | 0.268 | 0.273 |
| Maximum Depth | 20 | 0.123 | 0.278 | 0.281 |

We found that setting the minimum sample split at 400 gave us the best testing error rates, although not by a large difference. The maximum depth seemed to be slightly overfitting to the training data, as it had the smallest training error, and the largest validation error and testing error as well. The decision tree models also performed worse in comparison to all the other models we used in this project to classify our data, almost doubling the optimal error rate returned by our linear support vector machine model. In addition, adjusting single parameters seemed to have little effect on the error rates. Using a decision tree model with no adjustments to its hyperparameters still gave us a testing error rate of 0.288, which wasn’t much larger than the optimal error rate we got with adjusting minimum sample split.

Stacking - Rebecca:

As our final test, we decided to combine our optimized models using the stacking method to see if we could improve our prediction accuracy. For our base models, we used our optimized Naïve Bayes, Linear SVM, and Decision Tree models. (We did not use our Neural Network model because it was implemented using Keras.) The rest of our parameters were the default ones set in the StackingClassifier provided by Scikit-learn because we wanted our results to be solely based on our optimized models from above.

|  |  |  |
| --- | --- | --- |
| Training Error | Validation Error | Testing Error |
| 0.0678 | 0.1084 | 0.1239 |

**Analysis and Conclusion:**

From our experiments we get the following rankings for our different models comparing the testing error:

|  |  |  |
| --- | --- | --- |
| Ranking | Model | Test Error |
| 1 | Linear SVM | 0.120 |
| 2 | Feed Forward NN | 0.117 – 0.129 |
| 3 | Stacking | 0.124 |
| 4 | Naive Bayes | 0.186 |
| 5 | Decision Tree | 0.260 |

Looking above we can see that we chose Linear SVM as our best model. The feedforward NN gave us instances of a better test error rate, but in terms of consistency and computational resources, the linear SVM gave us much faster and more consistent test results. The context in which we were to use these different models would definitely change which model is ‘best’ but if we take into account everything being equal in importance, we strongly believe the linear SVM provides us with a model with just the right amount of complexity to prevent underfitting and overfitting our data. While the top three models had similar results in terms of testing error, the Naïve Bayes and Decision Tree models had larger gaps in testing error in comparison to the other models. The Native Bayes model was likely too simple compared to more complex models like Neural Networks. As for the Decision Tree model, we suspect that it may be due to the large number of features, because it was also one of the models that took the longest time to run.