CS 155: Kaggle Competition Report

Advith Chelikani, Loko Kung, Joonhee Lee

February 11th, 2016

1 Overview

To start of the project, we listed out all the models we could think of and split them among group members. We started with the basic models we learned from class proceeded to include more advanced models as we researched into the specific field of text sentiment classification. We also made a GitHub Repository so we can reuse preprocessing and file writing code. For preliminary models, Advith made models for stochastic gradient descent, SVM, and neural networks. Loko made models for random forest, neural networks, and genetic evolution. Joon made models for random forests, ensembles of many different models, naive bayes, and AdaBoost.

We started off each working on different classes of models in hope to eliminate weaker models and eventually focus on only those that yielded superior results. As we tried and eliminated models, we ended up homing in on using random forests and AdaBoosted models. When we juiced as much as we could out of those classes, we took their results and began using a voting system to decrease the error by as much as we could. Ultimately, these would result in the best models we made and submitted.

For the most part, we encountered difficulties whenever we tried to implement or use a new class of models. Learning the new frameworks and understanding what parameters to tune in order to get the models to produce the best results was often time consumming and tedious. While some frameworks provided cross validation tools, the one's that did not required manual cross validation setups and rigs. Aside from tuning the parameters, we also had a hard time trimming features and normalizing the data. (More on this under Data Manipulation.) Finally, we were constantly worried about overfitting to the training data so we carefully selected our final submissions in order to use more random models that are less prone to overfitting.

2 Data Manipulation

2.1 Manual Cross-Validation

Whenever a model class framework did not provide an out-of-box cross validation tool, we often started with the simpler approach of splitting our training data with 80/20 split and training on the first 80 percent and testing on the remaining 20 percent. If the results were promising, we would continue to try using 5 or 10 fold cross-validation methods to further test the model. For the most part, we used the 80/20 split method to rule out model classes if they were incapable of performing better than a score of 0.7 on the test data. (We made a couple of exceptions since we also wanted to have a a set of decorrelated submissions which is beneficial for the voting classifiers.)

2.2 Feature Selection

Although our final submissions did not really use any feature selection at all, we tried using the feature selection tools that the SKLearn Toolkit provided. We tried a couple of different approaches, namely a percentile cut on the features, a constant number, and a more conservative method of only removing features

that yield basically no information.

For percentile selection, we attempted to remove the bottom quartile of features (based on how evenly distributed they are between the two classes). The results generated using this form of feature selection was at best identical to those produced with all the features and was often weaker. When we tried removing more, around the bottom 30 percent, the results began to diminish greatly. Therefore, we decided that this form of selection was too indiscriminant and not conservative enough. Similarly, removing a constant number of features was also too indiscriminant since the two are essentially equivalent.

Finally, when testing with the most conservative method we realized that less than 10 features were being removed and decided that the difference was so small that there was no reason to both with removing any features at all.

2.3 Feature Normalization

After looking online for better methods to normalize the data, we found the TFIDF, and Delta TFIDF normalization methods that is geared towards test sentiment classification problems. Luckily we found that SKLearn also provided a tool to run this sort of normalization. Unfortunatley, the results were again at best identical to our best results so very quickly we decided to drop the normalization. The normalization also caused some of our models to work incorrectly so the decision was also based on convenience.

3 Learning Algorithm

3.1 Linear Models

3.1.1 Stochasitc Gradient Descent

Although the problem was a classification problem rather than a regression problem, we felt that the classic machine learning paradigm, the stochastic gradient descent models were worth trying. Using the SKLearn Toolkit's sklearn.linear_model and parameter searching tools, we ran the SGD with all combinations of losses (hinge, log, modified_huber) and penalties (l1, l2, elasticnet). After running each (loss, penalty) combination for 3000 iterations, we found the score on its predictions on our testing portion of the data set. We took the combination with the highest score, which was (modified_huber, l1) with a score of .652 and used it to predict on the public leaderboard data. This resulted in a score of .522.

Since the model produced relatively low scores, we decided that even though we invested time in tuning the parameters, the linear model was too weak to accurately predict the sentiments from text. We did little investigation into the model afterwards.

3.1.2 SVM

As we were researching online, we found that the SVM often did well with bags-of-word representations so even though the stochastic gradient descent linear model was too week, we figured we would nonetheless try the SVM model along with all the different possible SVMs that SKLearn provided.

Using SVM from sklearn, we found that LinearSVM reslted in the best score for the data. We then proceded to tune parameters for our SVM. We used an error tolerance of 10^{-6}. We set the dual flag to False, which sklearn documentation indicated was beneficial when the number of features is less than the number of training samples. This got us a score of .594 on the public leaderboard data.

Unsurprisingly, the model was not too great at predicting since the data was probably non-linear. Again, since the preliminary results were weak, we dropped any further research of the model.

3.2 Neural Networks

Towards the beginning of the class, neural networks were introduced by not particular elaborated on. That being the case, we decided to research into the model and try implementing it with different frameworks. This proved to be one of the harder models to implement simply due to the large number of parameters that could be tuned and the lack of real understanding of the inner mechanisms of the model. We nonetheless used two different libraries, Keras and PyBrain to write neural network models.

Using a Dense neural network with a binary classifier in Keras, we ran through all possible combinations of losses, optimizers, activations, and initalizors in an overnight trial and decided on (binary_crossentropy, adagrad, sigmoid, he_uniform) the respective parameters. While giving a score upwards of .72 on our own testing data, this unforunately only gave a score of .590 on the public leaderboard.

Using the PyBrain framework, we only touched the very basics. We built a couple of example networks and ran some tests that resulted in test validations of under 0.55. Since by the time we started trying the framework we were already running out of time, we never fine tuned any parameters and didn't end up making any submissions with the framework.

3.3 Naive Bayes

Even though naive bayes isn't known to produce very accurate predictions, we thought we would give it a try after learning about it in lecture. We also found in literature that Naive Bayes can sometimes do well in sentiment predictions. We used existing software known as Weka which is produced in New Zealand. The software provided a built in support for cross validation so testing the ability of the Naive Bayes model was very easy. Since there aren't too many parameters that needed to be varied to test the model, we exhastively tried everything and found that the model had capped at a score of roughly 0.6. This included all sorts of data manipulation such as feature selection and even the TFIDF normalizations. Since this was significantly weaker than any of the other models, we never made any submissions with these models and didn't continue researching about them.

3.4 Random Forest

Of all of the other models, the random forest model quickly proved to be the best, yielding test validations of scores 0.7 and up. Since these were clearly the superior models, we invested a huge amount of time in tuning the parameters and testing all the combinations to maximize their yield. Although the SKLearn Toolkit framework provides a Grid Search tool which allows us to exhaustively run cross-validations for huge ranges of paramters, we also ran some manual tests in order to visualize the changes as we varied certain parameters. Overnight, however, we would run huge searches across large ranges of parameters in hope of developing the best possible models. The following graphs were the results of the manual tests for visualization purposes:

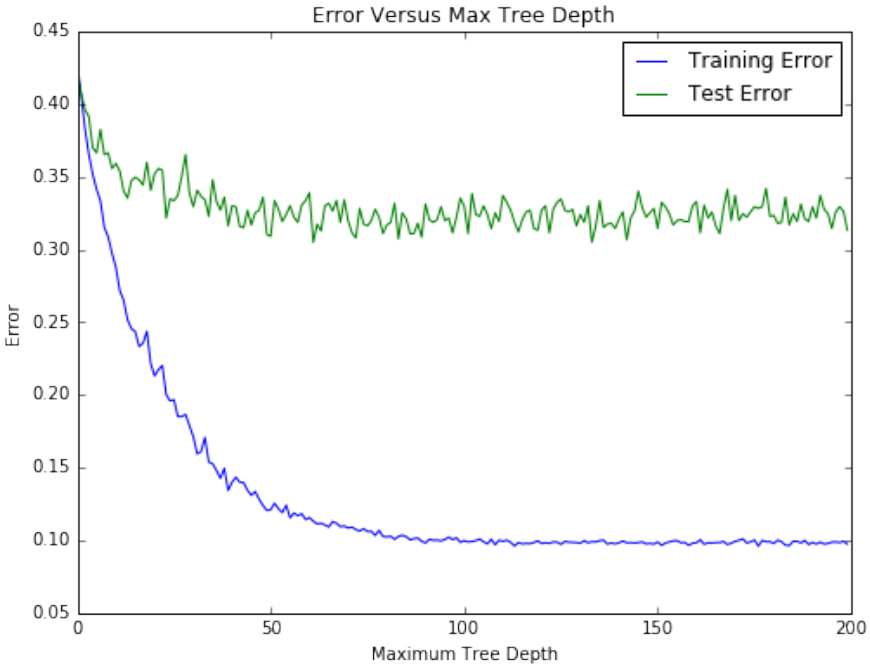

Figure 1: Notice that the error flattens out as we approach the larger and larger values for the maximum depth. This indicates that if we go any further we could easily be overfitting.

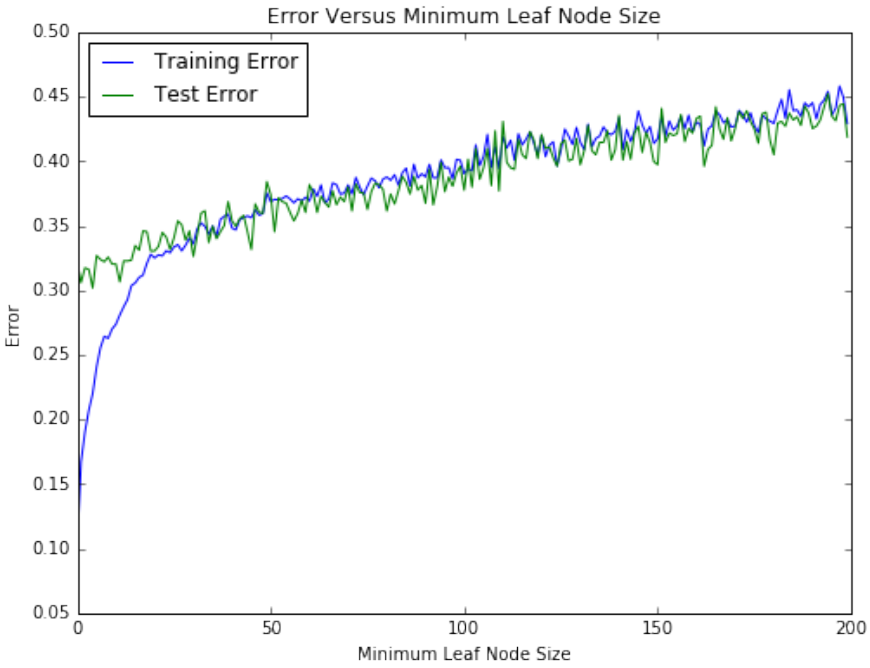

Figure 2: Notice that although the smaller the size of each node, the lower the error, if we use such small node sizes, we would be overfitting greatly to our training data.

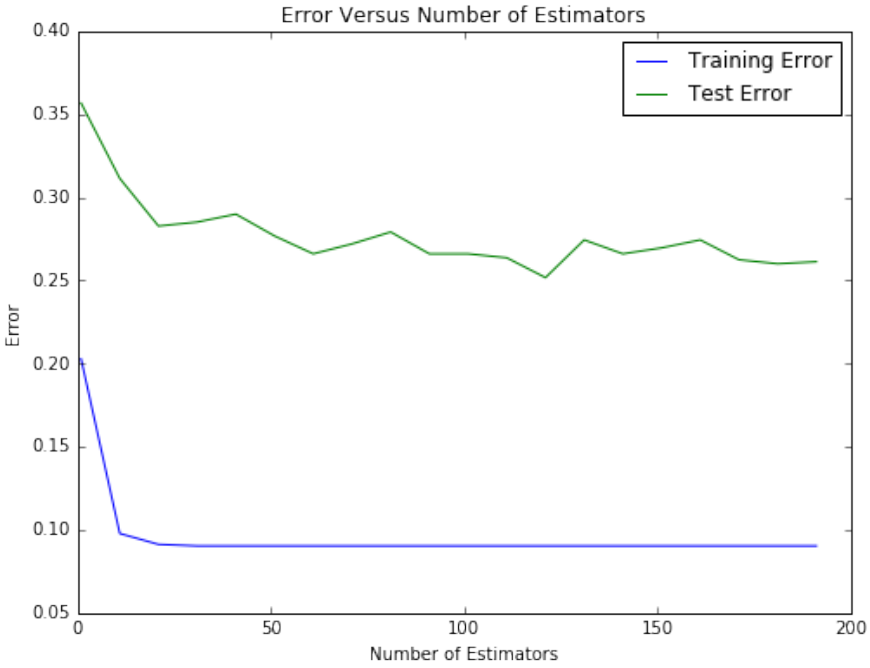

Figure 3: Although it seems that at some point, the number of estimator no longer lowers the error, in our grid search, we found that on a logarithmic scale, error still decreases as we increase the number of estimators. This graph's scale simply doesn't include a large enough range to show these changes.

Individually, we ultimately found that the random forest models were by far the most powerful models in classifying. In fact, the model that produced the best private score for our team was a simple random forest with 1000 estimators. (We unfortunately did not select this model for the scoring.) Since we also realized that random forests don't really overfit to data even if we increase the number of estimators, we ended up producing models with tens of thousands of estimators in hope of getting the highest score possible. The other parameters, however, were all limited by the optimal values we found using the grid searches.

3.5 AdaBoost

Once we felt like we had pushed the random forest models to their limits, we began to AdaBoost them to further increase their scores. Ultimately, an AdaBoosted random forest would yield the private score that we selected to submit. In order to boost our models, we used SKLearn's tools but found that this often took huge amounts of time. We almost exclusively ran these overnight and did not bother manually cross-validating since it would take too long. We simply used the built in features from SKLearn.

Towards the end of the competition when we had no other great ideas to submit, we ended up running multiple AdaBoosted random forests in hopes that by chance we could increase our scores.

3.6 Ensemble

Since we had tried basically all sorts of model, our final resort was to make ensembles out of all the previous models. We made voting ensembles out of collections of different models since we wanted to minimize the correlations between each set of votes. Our "super" ensemble was a collection of Logistic Regression models, AdaBoosted random forest models, SVM models, and Naive Bayes models. These "super" models performed

similarly to the AdaBoosted random forest models with slightly more deviations.

We also tried weighing different models within the ensembles with different weights to try to maximize the possible results. We often weighed the AdaBoosted random forest models heavier than the others since it was individually the best. This unfortunately caused the ensembles to yield very similar results to the AdaBoosted random forest models on their own. Nonetheless, there was still improvement.

Again, like the AdaBoosted random forest, it was hard to make graphs from the ensembles and cross validation was essentially meaningless for these models since by assumption each of the individual models within the ensemble were optimized.

3.7 Genetic Evolution

In terms of genetic evolution, we were simply trying to increase our public score. With each submission, we were told a percentage of how many correctly classified instances there were. Using this information from each submission, we tried to use a sort of mastermind like approach to find which values were most likely correct and which ones were not.

We used the DEAP framework to do the genetic evolution and used a scoring function that tried to generate a set of predictions that satisfied all the percentages from the previous submissions. By doing this, we were hoping that with each submission, we would narrow down the instances that we were predicting correctly and which ones we were predicting incorrectly. This method was geared only towards increasing the public score and would essentially be useless for the private score because we were completely overfitting to the public score set. Nonetheless, we had intended to use a random forest or ensemble model for the private score anyways.

Unfortunately, we did not make enough submissions using this method to actually produce anything significant. This method was more of a toy experiment than a real model for the data since we had no way to actually compare each generation in terms of the whole data set.

4 Model Selection

Selecting the optimal model towards the end of the competition was actually pretty simple. Using random forest models allowed us to eliminate (for the most part) much of the possibilities of overfitting since our parameters were tuned well using cross-validation. We unanimously decided to include the highest public score AdaBoosted random forest model.

The other model we included was a voting ensemble of fine-tuned models. Again, since we cross-validated for each individual model within the voting ensemble, we were confident that we wouldn't have too much problems with overfitting. To make it simple, we again simply selected the highest scoring model for the public score to be our submission.

5 Conclusion

We ended up with a score of .69072 (11th place) on the private leaderboard and a score of .66716 (10th place) on the public leaderboard. Our minimal variance in position and score from the public to private leaderboard indicates that we did not overfit to the public leaderboard data.

We could have potentially improve our model by experimenting with different splits for our training and testing data. We used a 80/20 split, but we could have given thought to some other splits as well. We could also have better tuned our parameters for various models, most notably neural networks. Some of the more complex models have a lot of fine tuning options which we did not take advantage of due to both lack of knowledge and lack of time. If we were able to determine relevant parameters in models and tune them, we would likely have seen an increase in our score.

There are several things we learned from this project that we would implement in future Kaggle competitions. We would not spend time implementing models like SGD, which are clearly less powerful than models like random forests. We also realized the huge difference that even a small variation in parameters can have on a model. For future competitions, we would spend tuning parameters for powerful and viable models like random forests.