Miniproject 1: NA CS LUL

Joey Hong, Rohan Choudhury, Chris Haack

February 13, 2017

1 Introduction

- Team Name: We selected the team name NA CS LUL as a pun on Computer Science, as well as the common Twitch meme related to creep score in League of Legends and the poor performance of NA.
- Group Members: Joey Hong, Rohan Chodhury, Chris Haack
- Division of Labor:

  - Joey Hong: Wrote modules for importing data and applied classifiers for these methods and general parameter search for these classifiers. He also worked on the ensemble classifier.

  - Rohan Choudhury: Helped with general parameter search and improving classifiers using rigorous parameter tuning for model selection.

  - Chris Haack: Built and created deep neural networks for classification, and assisted with development of methods to reduce overfitting.
- Project Code: `https://github.com/jxihong/cs155-kaggle`

2 Overview

We summarize our approach in this section and go into more detail in following sections.

2.1 Models/Tools used

It was clear from the data that a majority of the features were categorical. Intuitively, this means that tree-based approaches will probably perform better than numerical approaches such as linear classifiers and neural networks.

We ultimately decided to consider all the major classifiers offered in scikit-learn, and tune the parameters for the best-performing one. We also tried a blended ensemble method where we selected models from a library based on performance on a validation set.

Our results are summarized below:

- In the general selection process, we varied the parameters for each classifier to determine the one with the best cross-validation error.
- We settled gradient-boosted trees, or XGBClassifiers with XGBoost, which, after being tuned heavily for optimal parameters, resulted in a best accuracy of 0.78625 on Part 1 and 0.75477 on Part 2.
- Our ensemble approach did not work as well as expected, perhaps due to the sheet number of mediocre models that were selected into the ensemble, and we ended up reducing the to ensemble to solely XGBClassifiers. This got a best accuracy of 0.78238 on Part 1 and 0.75471 on Part 2.

2.2 Work Timeline

Most of our work was done on the last week, due to our whole team being involved with other homework.

- Week 1: Set up codebase on GitHub. Used scikit learn Gridsearch to test out all models in `sklearn.ensemble` and determine the best one. Begin testing neural networks.
- Week 2: Constructed ensemble classifier. Tune the parameters for XGBoost carefully. Further work on optimizing fully-connected neural networks.

3 Approach

3.1 Data Processing

After loading the data, we got a total of 381 features, which erred on the larger side for more computationally intensive models.

Our first intuition for reducing dimensionality without compromising accuracy was to remove all features with 0 standard-deviation. Such features would provide no new information. We found a total of 15 features with 0 SD, and removing them gave 366 features.

Since we were considering numerical methods, we thought it would also be beneficial to standardize the data. We used `sklearn.preprocessing.StandardScaler` to normalize the data to 0 mean and 1 SD. Standardizing the data would not help with any of our decision tree models, but it shouldn't negatively impact performance, so we standardized the data before training every classifier. Furthermore, we wrote standard data reading and writing routines to speed up work.

3.2 Model Selection

We considered linear classifiers such as linear support vector machines and logistic regression, nonlinear classifiers like SVMs with non-linear kernels and neural networks, as well as ensemble methods such as boosted and bagged decision trees and decision stumps.

To find the optimal model, we checked the performance of several model classes via scikit-learn. Specifically, we investigated seven different methods:

Method	Description
Linear SVM	Linear support vector machine, separates the data with a hyperplane
Radial Basis SVM	Support vector machine, with radial basis kernel
Logistic Regression	Linear classifier that outputs probabilities for each class
Random Forest	A large set of decision trees trained on bootstrapped datasets
Extremely Randomized DT	Similar to random forest, but uses randomly generated splits in the trees
Adaboost	Boosting for decision stumps, reweights the data based on misclassification
Gradient Boosted DT	Boosting for decision trees, but learns to predict residuals of other trees in the ensemble
Neural Network	Uses a deep network to learn nonlinear features for classification

For each model (except the neural network model, where this was implemented by hand due to compatability issues between keras and Sklearn), we used the `sklearn.ensemble.GridSearchCV` method, varying the input parameters over relatively wide ranges for each model. We also computed the 3-fold cross validation error to select the best parameters, and to find the optimal validation performance. We summarize the result below:

- **SVM**: Support-vector machines. We varied the margin parameter C, as well as the kernel between linear and radial-basis. For the radial-basis kernel, we selected different radii of influence gamma. The best cross-validation accuracy was 0.771.
- **RF**: Random forest. We varied the number of estimators as well as max depth and features used for constructing a tree. The best had cross-validation accuracy of 0.781.

- **ERF**: Random forest with extra randomization. We varied the number of estimators, the split decision criterion (either Gini or Entropy), and max depth and features for the tree. The best validation accuracy was 0.771.
- **AB**: Adaboost decision stumps. We varied the number of estimators, as well as the learning rate, which was sampled from a exponential distribution between 10^{-4} and 10.
- **LOG**: Logistic regression classifier. We varied the regularization parameters exponentially. The best validation accuracy was 0.770.
- **XGB**: Gradient-boosted trees using XGBoost. We varied number of estimators, with linearly spaced parameters in maximum depth of tree, and percent of training data and total features used to build each tree. The best had validation accuracy of 0.785.

We found training the gradient-boosted trees with scikit-learn to be too slow, and found that xgboost provided a similar implementation that trained 10x more quickly. We used `xgboost.sklearn.XGBClassier`, which wrapped the xgboost classifier with scikit-learn parameters, so the model still worked with gridsearch.

In the case of neural networks, which we implemented outside of scikit-learn, we attempted to vary the layers and structure to optimize the performance. due to compatability issues with the Keras package we had to implement a lot of the neural networking code by hand.

To do this we randomly sampled the data and created a training and a validation set and implemented a version of gridsearch to select optimal parameters. exploring a space of 100 different combinations of parameters we created a model that produced an error of 77.24% on the validation set. This was not as good as our other models and thus was not selected. This error method was not as good as a 3-fold cross validation, but seeing as the upperbound was 77.24% on the dataset, we did not implement more accurate methods of measuring performance. This network ended up having 2 hidden layers with a sigmoid activation function on the last layer, and 100 neurons in both the first and second layer with a dropout rate of .5 applied to the first layer.

The best validation performance came from the XGBoost,which we submitted and got a test error of 0.78625. So, we decided to further tune the parameters to increase the accuracy of the model.

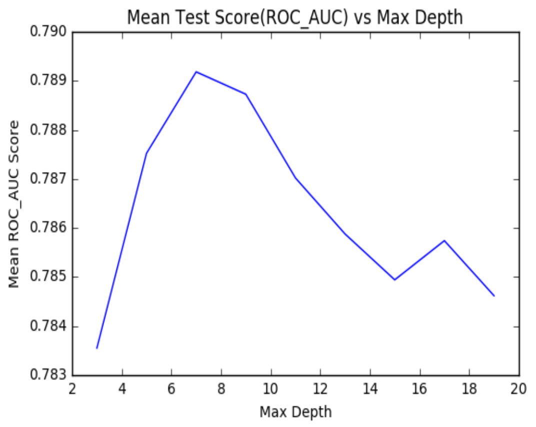
3.3 Parameter Tuning

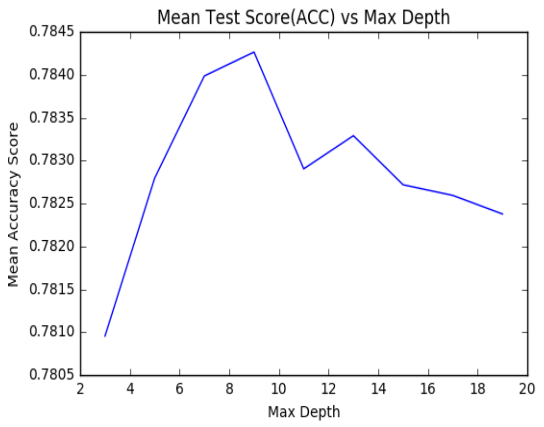
We describe the parameters of XGBoost in further detail.

- `n_estimators`: This is the number of decision trees in the forest. We varied this between the range of 10 and 1000.
- `max_depth`: This is the maximum depth of trees in the forest. We varied this between 3 and 20 with step size 2.
- `sub_sample` The fraction of the data to be sampled for each tree in the forest. We varied this between 60% and 100%.
- `col_sample` The fraction of features to be sampled for each tree in the forest. We varied this between 60% and 100%.
- `reg_alpha` Regularization parameter (L1 regression).

To do the parameter search in reasonable time, we first held the number of estimators at a constant 100, and searched for `max_depth`, `sub_sample`, `col_sample`, and `reg_alpha`, in that order.

We first varied the maximum depth of each tree, and used 3-fold cross-validation to score models. We explored both accuracy and ROC AUC scores as our metric, which produced similar results.



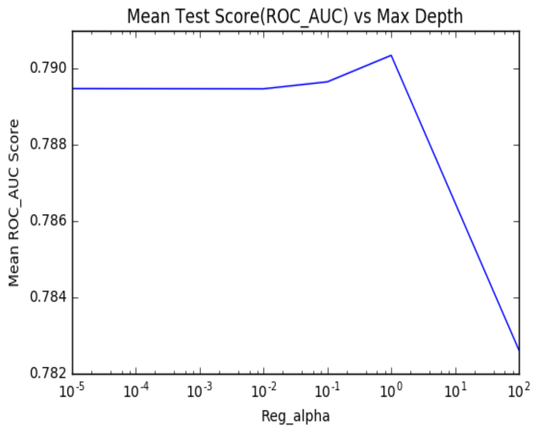


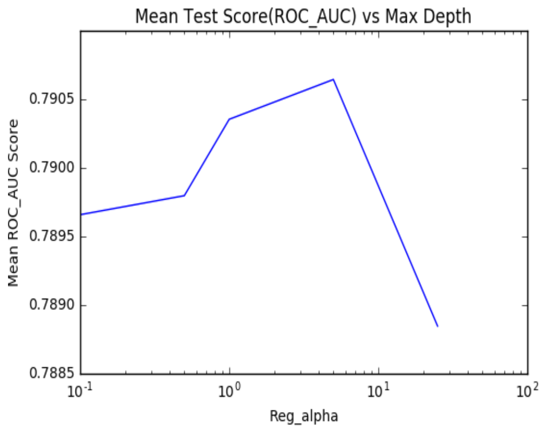
Though the accuracy plot reached a maximum at 9, we decided to follow the maximum of the ROC AUC plot and use 7. This is because ROC AUC generally behaves better as a metric for more imbalanced datasets.

Using 7 as our depth, we then varied both subsample and column sample. Since they were not independent parameters, we took all combinations of such parameters, leading to 16 different pairs. We used ROC AUC as our score metric. The highest score produced was with subsample=0.9 and colsample=0.6, where a mean cross-validation score of 0.7894 ± 0.0036 was achieved. We reproduce the top 5 pairs below:

Rank	Subsample	Colsample	Score	Standard Deviation
1	0.9	0.6	0.7894	0.0036
2	0.8	0.8	0.7893	0.0041
3	0.8	0.9	0.7893	0.0048
4	0.7	0.9	0.7892	0.0052
5	0.8	0.7	0.7891	0.0031

Finally, using our values for depth and subsample/colsample, we varied the regularization term α. We first tried exponentially distributed values from 10^{-5} to 10^2, then did a tighter search around the most promising α value. We stuck with ROC AUC as our score. Consider the following plots:





Using the plots for each parameter as well as our knowledge about the models, we decided on a maximum depth of 7 for the trees, a subsample value of 90%, a column sample of 60% and an α value of 5. This gave a test accuracy of 0.786.

3.4 Ensemble

From looking at individual models, XGBoost's gradient-boosted trees was a clearly the best classifier. We thought that perhaps attempting an ensemble classifier, using a model library consisting of all the methods we considered, would perform better than any single model.

Our first step was the compile a model library. We chose a library of approximately 500 base learners, ranging from 7 different classifier types with varied parameters. They included SVM, RF, LOG, XGB, and the following models:

- **DT**: Decision trees. We varied the maximum depth, and the maximum number of features used.
- **KNN**: K-nearest neighbors. We found 25 linearly spaced values from 1 to 300 for number of neighbors K to look at.
- **SGD**: Linear classifiers trained with SGD. Due to the number of features we trained with Elasticnet penalty. We varied the regularization parameters using an exponential distribution.

We trained all our models on 90% of the training data, saving the remaining 10% as the validation set. We then selected a single model, that improved validation accuracy the most, and included it in our ensemble. We used a simple majority voting mechanic when predicting labels.

```
1. Start with empty ensemble.
2. For each model in library:
 -    Test accuracy of ensemble with model on the validation/hillclimb set.
 -    Record the best model and accuracy.
3. Add the best-performing model to the ensemble.
4. Repeat Step 2 until accuracy stopped increasing for 3 iterations.
```

We realized that the algorithm was selecting the same RF model numerous times (out of the 47 in the ensemble, 14 were RF with the same parameters). We wanted a different method that rewarded models that predicted the right value, even if overall accuracy didn't increase.

We can approximate the probability that our ensemble model will predict the right value as the percent of models in the ensemble that predicted correctly. Instead of maximizing accuracy, we maximized the sum of log percentages of correct predictions. That way, if a model is predicting the correct value, but the accuracy is unchanged, the score will still improve. Running the algorithm produced 68 models, with more variety among models chosen.

We trained the selected models on the entire training set, and tested it on the test set. We realized, however, that the ensemble did not predict as well as a single XGB model with optimal parameters, arriving at approximately 0.764 accuracy.

In the end, we decided to stick with XGB models, training a majority-vote classifier with 10 selected XGBClassifiers on a hillclimb set with different parameters. This gave a test accuracy of 0.782.

4 Results

We chose the two models with the highest Part 1 test accuracy.

Once we found the optimal parameters for the XGBoost model, we submitted it to Part 1 and Part 2 with the tuned parameters. We got a best accuracy of 0.78625 on Part 1 and 0.75477 on Part 2.

The classifier performed very poorly in Part 2, which could possibly be attributed to overfitting with the parameters we chose. We should have reduced the number of estimators to around 200 rather than 1000.

We also submitted our voting classifier using 10 differently parametrized XGBClassifiers. This got a best accuracy of 0.78238 on Part 1 and 0.75471 on Part 2.

Overall, this model never performed as well as single tuned XGBClassifier, and still overfitted to the Part 2 test set.

Our best model was in fact an XGBoost model with few estimators than the one we submitted, which achieved Part 1 accuracy of 0.001 higher. This further shows that we suffered from overfitting in model selection.

5 Conclusion

5.1 Discoveries

- First one of the major lessons we learned is how little we are able to learn about some data sets. For example the data is skewed such that roughly 75% percent of the objects are of one class. Simply guessing that class would give an accuracy of around 75% percent. This meant that the best classifiers only led to a 4 percent improvement and that is pretty poor given the number of training examples.
- While hyper-tuning parameters did improve the classifier, the increase in accuracy was an insignificant 0.002 on the test set. It became clear that if we really wanted to do much better, we would need to employ a different ensemble method altogether. Also, though the ensemble method trained on the hillclimb/validation set looked good on paper, it clearly did not perform as well as anticipated, likely due to the fact that mediocre classifiers were mixed in with good ones, as well as that our validation set was too small to represent the distribution of the actual test set.
- Another important lesson that we learned was to not learn the test data, for example we kept selecting our models based on the fact that they were doing better than the online example provided in Test 1. By using this selection criteria, we chose a model that may not have learned the features of the actual data, but rather of the test set itself.

5.2 Challenges

- Like every other team we struggled to learn from the data. This in itself was particularly difficult in the neural network examples where our models kept making random guesses and consistently just dying. Through a lot of work in normalization, regularization and making sure the training data was randomly sampled (i.e. we didn't learn any bias from the ordering of the training data) we were able to overcome this and create a neural network that learned the data set. (abeilt not as well as some of our other models)
- Another issue we continually debated was determining how to evaluate our models properly and to prevent against learning the test set as opposed to learning the underlying voter premise. Unfortunately we ended up overfitting, and not selecting our best model.
- In addition, we realized that our models were consistently predicting too many 1-labels, and not nearly enough 2's. Our best classifier predicted around 85% 1's whereas the true distribution was closer to 74%. We thought that oversampling the data, by multiplying the number of points labeled 2, would help compensate by increasing the error for prediction a 2 incorrectly as 1. However, doing so only seemed to worsen the test error of the model. Perhaps more sophisticated sampling techniques would need to be used. We considered a voting classifier where each classifier was trained on samples of the data, some with mostly 1's and others with mostly 2s, but did not have time complete the task

5.3 Concluding Remarks

- We learned a lot about the process of evaluating models and determining how to chose the right model for selection. Even though we overfitted and selected the wrong model during the competition we were still able to learn a lot.