Jerome Domingo and Scott Gettinger
Team uci178- SGJD
CS 178
Fall 2012

CS 178 Final Project - 2009 KDD Cup

Challenge

The challenge we set out to address in this KDD Cup project was to classify all three target values (appetency, churn, and upselling) for the small dataset. We decided that we didn't have the resources to properly address the large dataset. We also wanted to see the different effects of the same classifiers on upselling, which seems to be relatively easy to predict, and the other two target values, which appear to be more difficult.

Modifying the Data

We began by considering the dataset itself. First, we eliminated features which were obviously useless (either missing all values or all zeros). Next, we chose to impute values for the missing data points provided. Each missing value was replaced with the median of non-missing values of that feature.

To account for the imbalance between positive and negative classifications in the provided data (the data is overwhelmingly negative values), we replicated points with positive values at random, creating a much better balance between positive and negative target values. Scott wrote the Matlab code we use for data modification.

Classifier Choices

Even the small dataset for this challenge is quite large, so we knew that it would take quite a lot of time for certain classifiers to make predictions. We determined that one of the better classifiers to work with high-dimensional data is a decision tree. Scott was the team member working on our decision tree classification.

We also decided to use a k-nearest neighbor classifier for comparison, since we know that it's a simple classifier that often works well. We ran into some problems with KNN taking a long time to run. This was due to having to compute the euclidean distance formula for every point in the dataset. Jerome worked on our KNN classifiers.

Decision Trees

Scott Gettinger worked with the decision tree classifiers.

Demo Code

We began our study of decision trees by trying out the Matlab demo code provided by Qiang on Piazza. Without the code to correct for imbalance, the classifier was scored just a little over 0.5 for both churn and appetency, with a score of about 0.7 for upselling. This, along with some of the existing scores on the leaderboard, contributed to our belief that upselling is easier to predict than the other target values.

The other demo, which included replication for imbalance, was much better (0.1 better) on churn and appetency than its predecessor, and slightly better (0.02) on upselling. This convinced us of the importance of accounting for the imbalance in target values, which we did by repeating +1 data. Based on the performance of these single decision trees, we decided to use bootstrap aggregation to improve performance.

Bagging Decision Trees

For the decision trees with bagging, we used Matlab's built-in binary decision tree classifier (ClassificationTree), surrounded by a Matlab bagging script based on code provided in the Ensembles: Bagging lecture for CS 178. The built-in Matlab decision tree gave an easy way to change variables for the decision trees, such as the minimum number of parent points to split, and the split criterion (Gini or entropy). It also gave an easy way to output confidence scores for each point rather than +1/-1 predictions.

Varying MinParents

Now that a classifier had been decided, we needed to choose parameters for it. We started off with default values for the classification tree. For the bagging parameters, we used 50 classifiers, each with 5000 randomly chosen points from the dataset. The evaluation with these parameters took much too long and ran into memory issues. Changing MinParents, the minimum number of parent points to split, helped to alleviate these problems. Rather than the default of 10 minimum parents, we chose 50. This gave us our first predictions. After that, we chose MinParents values of 20, and then 30. We obtained confidence level predictions by taking the mean of the scores provided by the predictor for each point. These were the results of the submissions[1]:

	Appetency		Churn		Upselling		
MinParents	Train	Test	Train	Test	Train	Test	Total Score
50	0.7121	0.6616	0.8998	0.7824	0.8915	0.8381	0.7607
20	0.7331	0.6574	0.9083	0.7750	0.9106	0.8368	0.7564
30	0.7327	0.6745	0.8970	0.7800	0.8954	0.8351	0.7632

In the 50 row, the results are quite good, with a more than 0.1 increase relative to the single decision tree. They fit the training data slightly better than the test data, which is to be expected. With MinParents = 20, the classifier is beginning to overfit the data. The training accuracy is much higher than test accuracy, and the total score has gone down because of that. MinParents = 30 increased the overall score slightly compared to our original with MinParents = 50. Compared to the first ensemble, this result has much improved predictions on appetency, but slightly reduced accuracy for churn and upselling. We decided to use MinParents = 30 for further tests.

Changing the Split Criterion

Next, we tried using entropy as the splitting method, rather than the default Gini index. This increased the resulting score, but not much. The increase was small enough that it could have come from random sampling variation. Because it seems to increase accuracy, if anything, we decided to keep using entropy as the splitting criterion.

Increasing the Number of Learners

After the splitting criterion, we wanted to increase the number of learners. In order to run faster and create a better overall classifier, we decreased the number of features we used

[1] Full team results are available at http://www.kddcup-orange.com/results.php?ds=small&entrant=2559602

per classifier from all 212 (note that we eliminated 18 all-zero or all-NaN features) to 80. We doubled the number of learners from 50 to 100, and then to 200. This ran into some memory issues, but eventually we got it working. These gave great test results.

	Appetency		Churn		Upselling		
	Train	Test	Train	Test	Train	Test	Total Score
80 Features, 100 Classifiers	0.8651	0.7053	0.9788	0.7857	0.9322	0.8531	0.7814
80 Features, 200 Classifiers	0.8682	0.7111	0.9727	0.7922	0.9382	0.8552	0.7861

The first set's total score is much better than our previous score, mostly in appetency and upselling. The classifier seems to be overfitting to the test data some, based on the large difference between training and test accuracy. That could also explain why churn had the smallest increase in test accuracy: Churn appears to have the most overfitting, evidenced by its high training accuracy and comparatively low test accuracy. The 200 classifier set appears to overfit just as much, but gives a slight improvement across the board.

Decision Tree Conclusions

This is where we ended with the decision trees. If we had more time to change variables, we would try varying the number of features for each classifier to work with, as well as the number of points to give each classifier (for this project, we always used 5000 points for the trees). It would also be helpful to implement some self-scoring method, using hold-out data to evaluate results rather than uploading each time. The self scoring method would be useful for finding the optimal value for the many parameters we could change. This would take a lot of time (each run of the decision tree code can take many minutes), but it would be an effective way to produce the best classifier we could create.

K Nearest Neighbor

Jerome Domingo worked with the K nearest neighbor classifiers.]

Standalone K Nearest Neighbor

The first variant of the K nearest neighbors algorithm was the code provided to the class for the first homework assignment. However, using this code directly proved to be a significant challenge. Due to the algorithm's use of the Euclidean Distance computation on every point in the dataset, computing results was time consuming. Computation of results for churn training data alone took just over two hours. It was concluded that it would not be feasible to continue using k nearest neighbor without modifications for the rest of the dataset labels. Consequently, we have no score for standalone K nearest neighbor.

K Nearest Neighbor with Bagging

For the k nearest neighbor, we continued to use the Matlab code provided to the class for the first homework assignment. However, to improve upon it, it was used with the Matlab bagging code provided in the Ensembles: Bagging lecture slides for CS 178. This method would provide value predictions of either 1 or -1.

For bootstrap aggregation parameters, we used 10 classifiers, each with 1000 randomly chosen points from the dataset. K was set to 5. Due to a smaller set for k nearest neighbor to calculate the Euclidean distances, time for results calculation vastly improved. Time for computation had been reduced from 135 minutes for the entire training dataset with K = 50, to 13 minutes with the previously stated parameters; a reduction of time by an order of magnitude (a factor of ten). These parameters gave us the first k nearest neighbor results to score.

Appetency		Churn		Upselling		
Train	Test	Train	Test	Train	Test	Score
0.5	0.5	0.5	0.5			n/a

Due to an error during the upload of the first set, Upselling train and test were not scored. We recalculated the results once more, but this time, using K = 25.

Appetency		Churn		Upselling		
Train	Test	Train	Test	Train	Test	Score
0.5	0.5	0.5	0.5	0.5	0.5	0.5

The classifier is still not performing well. A score of 0.5 out of 1 is about as good as random. In the next iteration, we increased K to 200 and uploaded the results to have it scored.

Appetency		Churn		Upselling		
Train	Test	Train	Test	Train	Test	Score
0.5	0.5	0.5	0.5	0.5	0.5	0.5

Unfortunately, increasing the K yielded the same results as before. It appears that changing the K value would not significantly affect the score for a given size of random points and bagging value. This next iteration decreased the K value, as well as decreasing the number of points. K was set to 100, and the number of random points was set to 500. After the results were calculated, it was scored.

Appetency		Churn		Upselling		
Train	Test	Train	Test	Train	Test	Score
0.5	0.5	0.5	0.5	0.5	0.5	0.5

Decreasing the number of points did not help, so the K nearest neighbor algorithm must currently be underfitting due to the small number of points. Unfortunately, increasing the number of random points increased computation time. This negated the time efficiency that bagging had previously improved, and has made it unfeasible to compute given the current time constraint. To solve the underfitting due to random points, it would require adding more sample points from the training data. K nearest neighbor is thus an inefficient algorithm for calculation.

The most likely reason that score remained at 0.5 is that the code for K nearest neighbor or bootstrap aggregation was incorrect implemented, resulting in a state where all classified points were either all 1s or all -1s. All scores being a single result (only 1's or -1s) would explain why everything is scoring at 0.5, since it would mean half are right and half are wrong.