Boosted Decision Trees and Neural Networks on Rainfall Data

Lai, Eric *ellai@uci.edu*

Liu, Yiming *yiminl10@uci.edu*

Shen, Tong *tshen4@uci.edu*

December 8, 2016

1 Introduction

In this paper, we will use boosted decision trees and neural networks to predict whether there is rainfall at a certain location. The data we use to train our models are based on (processed) infrared satellite image information. The training data set contains 14 features and 200,000 observations in total. In order to properly assess our models, a validation split was performed, and we will use 160,000 observations for model training purpose while the last 40,000 observations will serve as our validation set. All performance measures of our models will be based on this validation set. Model performence will mainly be accessed using a receiver operating characteristic (ROC) and computing the area under the ROC curve (AUC). A priori, we do expect that the boosted decision tree models will perform better than the neural network models despite more computation time being required for these methods.

2 Methods

2.1 Adaboost

Adaboost is one possible method of training a boosted classifier. A boost classifier is a classifier in the form

$$F_T(x) = \sum_{t=1}^{T} f_t(x)$$

where each f_t is a weak learner which takes object x as input and returns a value indicating the class of the object. For example, in the two class problem, the sign of the weak learner output identifies the predicted object class and its absolute value gives the confidence in that classification. Similarly, the Tth classifier will be positive if the sample is believed to be in the positive class and negative otherwise.

Each weak learner produces an output, hypothesis $h(x_i)$, for each sample in the training set. At each iteration t, a weak learner is selected and assigned a coefficient α_t such that the sum training error E_t of the resulting t stage boost classifier is minimized.

$$E_t = \sum_i E[F_{t-1}(x_i) + \alpha_t h(x_i)]$$

Here $F_{t-1}(x)$ is the boosted classifier that has been built up to the previous stage of training, $E(F)$ is some loss function and $f_t(x) = \alpha_t h(x)$ is the weak learner that is being considered for addition to the final classifier.

At each iteration of the training process, a weight ω_t is assigned to each sample in the training set equal to the current error, $E(F_{t-1}(x))$, in that sample. These weights can be used to inform the training of the weak learner, for instance, decision trees can be grown by splitting sets of samples with high weights. For Real Adaboost, the output of decision trees is a class probability estimate $p(x) = P(y = 1|x)$, the probability that x is in the positive class. In [3], an analytical minimizer for $\exp(-y(F_{t-1}(x) + f_t(p(x))))$ for some fixed $p(x)$ was derived, typically chosen using the weighted least square error.

$$f_t(x) = 0.5 \log(x/(1-x))$$

Thus, rather than multiplying the output of the entire tree by some fixed value, each leaf node is changed to output half the logit transform of its previous value. We are going to compare the performance of real Adaboost method with that of common Adaboost to decide the number of iterations needed afterwards.

2.2 Gradient Boosting

Like Adaboost, Gradient boost is another very effective ensemble algorithm, which aims at constructing a strong learner through a combination of weak learner. However, unlike Adaboost which tries to generate weights for typically a set of existing weaker learners or randomly generated base learners, Gradient boost would generate a new learner (i.e. a decision tree in our case) at each round of the learning process to optimize the objective function.

For instance at the tth boosting round we would normally optimize following objective function:

$$\mathbf{Obj(t)} = \sum_i^n L(y_i, \hat{y}_i + f_t(x_i)) + \Omega(f_t) + \textit{constant}$$

Where, L represents a chosen loss function, and $\hat{y}_i$ is the predicted y_i from the $t-1$th boosting round, and $f_t(x_i)$ is the new learner which is to be added in the current round, and $\Omega(f_t)$ acts as the complexity control for f_t. Then, the general training task would be finding the right f_t to minimize the above objective function. This process is repeated for each subsequent boosting round.

There are several well-developed implementations for Gradient boosting currently. In this particular project, the implementation of XGBoost is used due to its flexibility and computational efficiency.

2.3 Neural Networks

For this dataset, utilizing a single linear perception does not give results better than simply guessing the correct classification. In order to improve our model, we also attempt to apply multilayer perceptron (i.e. feedforward neural network) models to our data. A multilayer perceptron consists of multiple layers of nodes or neurons where each node is a processing function with a (nonlinear) activation function (see Figure 1). Each multilayer perception consists of at least two layers. The first layer and last layer called the input layer and output layer, respectively. The number of nodes in the first layer is determined by the number of features in your dataset. Usually, the number of nodes in the last is one but can vary depending on the methodology used. The other added layers are called the hidden layers.

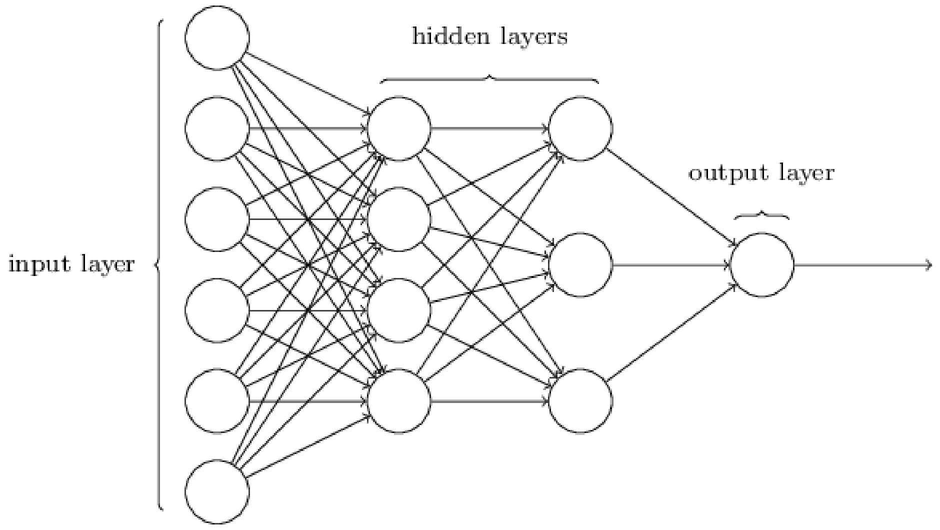


Figure 1: A multilayer perceptron model with four layers. The first, second, third, and fourth layers have six, four, three, and one nodes, respectively.

For the results in this paper, we looked into a backpropagation neural networks model due to the benefits in computational speed [5]. Backpropagation works by feeding the gradient of the activation function into an optimization method which in turn uses it to update the weights to minimize the loss function. For computation purposes, we will only use at most 3 layers and 60 nodes in each of those layers. Lastly, for our models, we will be using the logistic activation function, which is the most common activation function for binary response data. In particular, an identity activation function often results in convergence issues.

3 Results

3.1 Adaboost

For Adaboost, we first try to decide a reasonable number of iterations (i.e. estimators), so we generate a plot of training and testing error rates by the number of estimators using different methods.

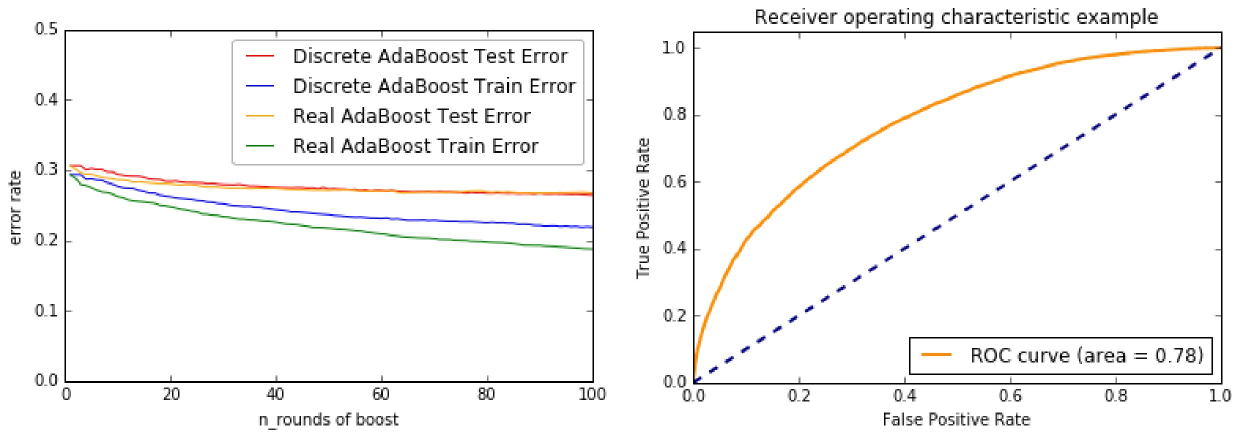


Figure 2: ROC curve and performances for different Adaboost models.

This figure shows the performance of some different methods on the data, and we can see that the Real Adaboost classifier performs better than Discrete Adaboost classifier. The error rates almost stay constant when the number of rounds of boost is larger than 20, so we decide to set the rounds of boost to 20.

From previous results, the decision tree classifier works well when max depth is 50 and min leaf is 64. Here we try several different combinations of tuning parameters.

1. Model 1: Max Depth = 50, Minimum Leaves = 64, AUC = 0.78

2. Model 2: Max Depth = 10, Minimum Leaves = 128, AUC = 0.76

3. Model 3: Max Depth = 5, Minimum Leaves = 256, AUC = 0.72

4. Model 4: Max Depth = 3, Minimum Leaves = 512, AUC = 0.71

Finally, we train an Adaboost classifier with a maximum depth of 50 and minimum leaves of 64 for each individual tree, and AUC value on this validation set is 0.78.

3.2 Gradient boosting

Before we formally train a meta-learner with Gradient boosting, we need to do a proper parameter tuning, often very lengthy and frustrating. To simplify this process, we decide to mainly focus our efforts on the three most important parameters, which have critical influence upon the formation of individual tree and proceedings of learning process.

1. η: The step size shrinkage used in updates to prevent over-fitting. η actually shrinks the feature weights to make the boosting process more conservative.

2. max-depth: The maximum depth of a tree. The deeper the tree; the higher the complexity.

3. min-child-weight: The minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min-child-weight, then the building process will give up further partitioning.

For the basic tuning process, we first use a grid search to narrow down a few parameter choices and then perform a 3-fold cross validation with training data to gauge each potential models' performance. Finally, we choose the optimal parameter setting based on the cross-validation performance of our candidates. Four particular parameter settings are cross validated at the end of our tuning process.

1. Model 1: $\eta = 0.3$, max-depth = 6, min-child-weight = 1

2. Model 2: $\eta = 0.2$, max-depth = 6, min-child-weight = 2

3. Model 3: $\eta = 1$, max-depth $= 25$, min-child-weight $= 0.7$

4. Model 4: $\eta = 0.5$, max-depth $= 15$, min-child-weight $= 1$

Each of these mode's cross validation performances are displayed below.

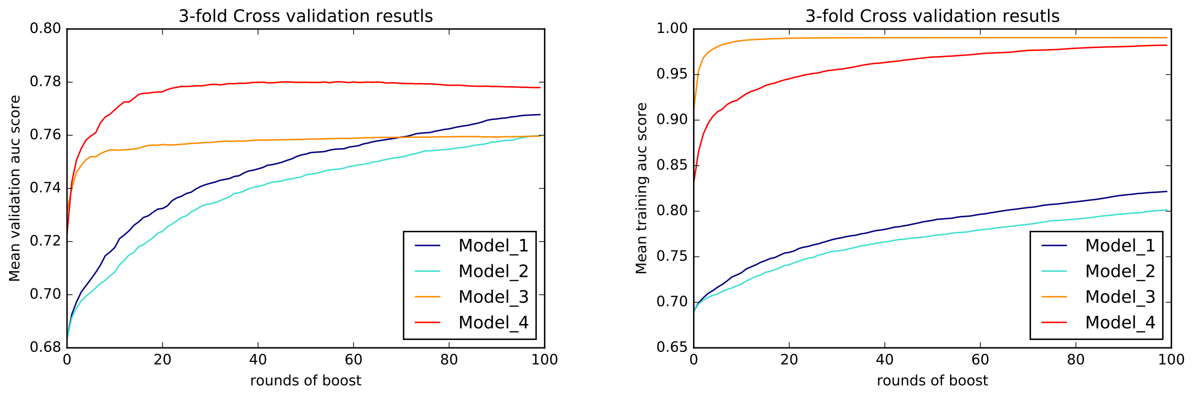


Figure 3: Cross validation results for different Gradient boosting models.

From the above plots, we can see that with this particular dataset (without feature reduction and modifications) high-complexity models like models 3 and 4 tend to reach peak performance with relatively fewer rounds of boosting than low complexity models, but they also show a dangerous tendency to over-fitting, resulting in validation AUC scores being capped at around 0.77. Low complexity models like models 1 and 2, on the other hand, show a healthy growth in terms of validation AUC score while also having a significantly lower training AUC score than models 3 and 4. In general, model 1's parameter setting seems to be the most optimal, since from above validation plots, we can see model 1 displays a great momentum to surpass all others in a couple of more rounds of boosting, in terms of validation AUC score, while having a relatively low training AUC score at the same time. The parameter setting of model 1 will be used in our final training model for Gradient boosting.

The optimal training performance on our validation set is obtained around 400 rounds of boosting. After 400 rounds, although the validation AUC score can still be further improved, its computational complexity begins to take a heavy toll on the entire training process. In fact

400 boosting rounds already took more than 30 minutes to run, even with multi-threading. the ROC curve of this optimal classifier is shown below.

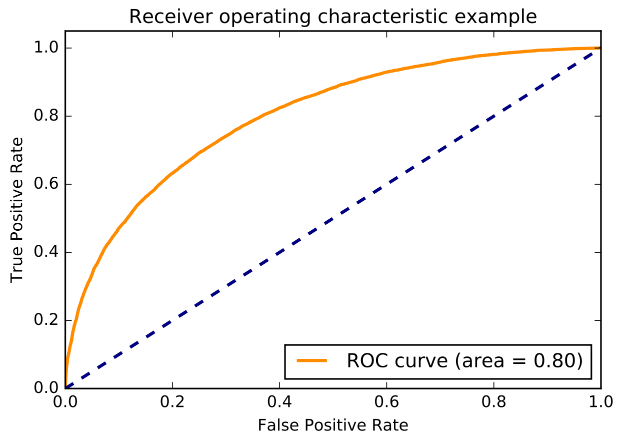
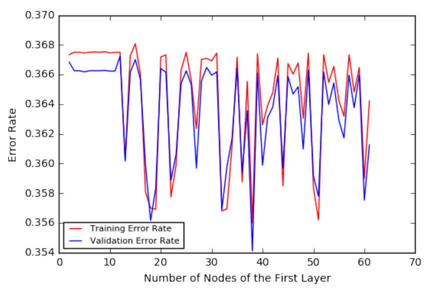


Figure 4: ROC curve for the optimal Gradient boosting model.

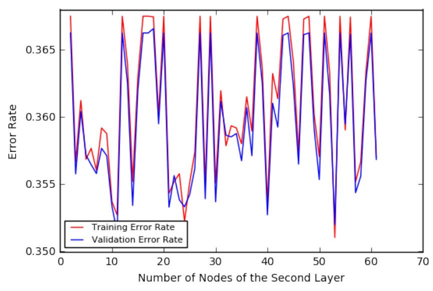
3.3 Neural Networks

In order to find the optimal neural networks model, we need to find the optimal number of layers, nodes in each of the layers, and gradient solver that results in the smallest validation error. We first focus on the former two issues and then the latter two. In Figure 5, we have a plot of the training error and testing error found by varying the number of nodes in each of the layers. We begin by finding the optimal number of nodes in the first layer and then use that value to find the optimal number of nodes in the second layer; the same process is repeated using the optimal number of nodes in the first two layers for the number of nodes in the third layer. The optimal nodes was found by finding the node that corresponded to the lowest testing error.

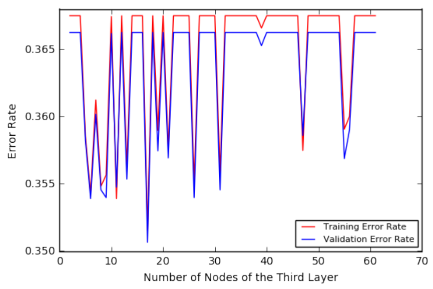
Utilizing the optimal number of neurons found for the three layers, Figure 6 then shows the area under the receiver operating characteristics (ROC) curve for our validation data in order to access the prediction capabilities in using up to three layers for a multilayer perception model. The multilayer perception model with three layers appears to perform the best but not by a lot. The plots in Figure 6 are unexpected, since initially we believed



(a) The minimum testing error was 0.354 and occurred when 38 nodes were used for the first layer.



(b) The minimum testing error was 0.351 and occurred when 11 nodes were used for the second layer.

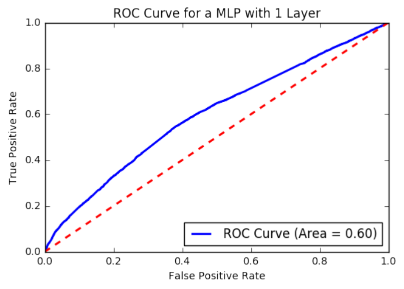


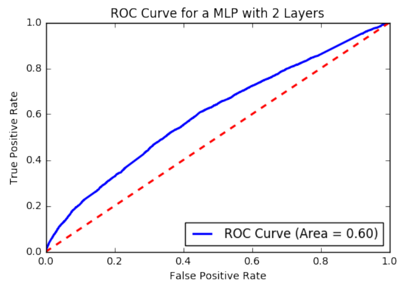
(c) The minimum testing error was 0.35 and occurred when 17 nodes were used for the third layer.

Figure 5: Testing and training errors of a backpropogation neural networks model when varying the number of nodes in each of the three layers.

that the training error and testing error should decrease monotonically for a while when we increased the number of neurons in each layer.

In Figure 7, we compare different optimization algorithms for our model. For neural networks, the model with the best prediction capabilities appears to be a three layer neural network model with a logistic activation function and using the ADAM method optimization solver; layers one, two, and three has 38, 11, and 17 nodes, respectively. This does surprise us as the ADAM method is computationally efficient and well suited for problems that are large in terms of data [4]. Though, the overall increase in prediction did not increase by a lot throughout this process; most likely due to the fact that a one layer multilayer perception with nodes between 2 and the total number of features in the dataset is usually sufficient [2].





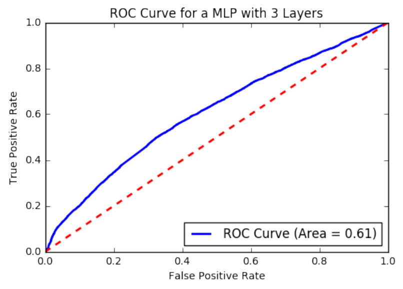
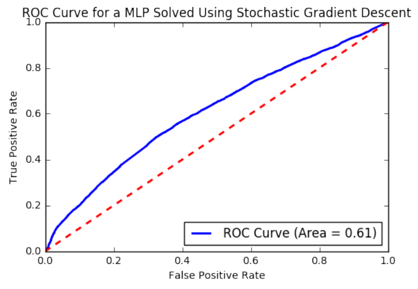
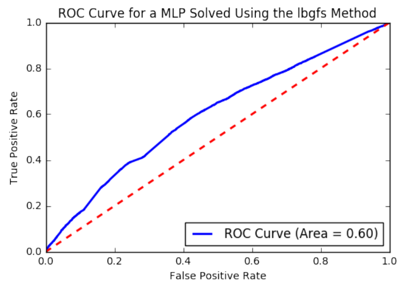


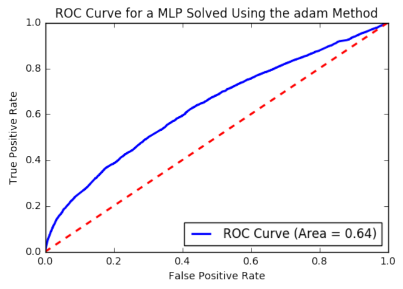
Figure 6: ROC curve and associated areas for a one, two, and three neural networks model with the optimal number of nodes found above.



(a) Stochastic gradient descent was used as the gradient solver.



(b) The Broyden, Fletcher, Goldfarb, and Shanno (BFGS) algorithm was used as the gradient solver.



(c) The ADAM method in [4] was used as the gradient solver.

Figure 7: ROC curve for a three layer neural networks model using three different gradient solvers.

4 Discussion

Overall, the optimal prediction models for Adaboost, Gradient boost, and neural networks in each of the three methods had varying levels of predictive capabilities on our validation set. The AUC for Adaboost, Gradient boost, and neural networks was 0.78, 0.80, and 0.64. On the other hand, the running time for Adaboost, Gradient boost, and neural networks was around 5 minutes, 30 minutes, and 1 minute, respectively. Thus, Gradient boost predicted the best on our validation set, but neural networks had a significantly faster run-time. In general, neural networks as a single complicated learner, although is highly computationally efficient in this particular case, it is heavily out-performed in terms of predictive accuracies by two boosting methods, which aims to construct strong meta-learners through the ensemble of weak learners. Further analysis can revolve around using boosting and general ensemble methods upon learners such as neural networks, instead of only decision trees, and compare its performance and computation complexity with conventional decision tree ensembles. Further progress can also be made through designing new features and optimizing the model with any special aspects of the dataset such as outliers and sparsity.

5 Acknowledgments

The authors would like to thank Professor Alexander Ihler for his guidance and support throughout the quarter. Without him, this project would not have been possible. We would also like to thank the University of California, Irvine's Center for Hydrometeorology and Remote Sensing, including Dr. Soroosh Sorooshian, Dr. Xiaogang Gao, Dr. Kuo-lin Hsu, Dan Braithwaite, Yumeng Tau, and Negar Karbalee for allowing us the use of this dataset.

6 References

[1] J. Friedman, T. Hastie, and R. Tibshirani, *Sparse inverse covariance estimation with the graphical lasso*, Biostatistics, 9 (2008), pp. 432–441.

[2] J. Heaton, *Introduction to neural networks with Java*, Heaton Research, Inc., 2008.

[3] B. Kégl, *The return of adaboost. mh: multi-class hamming trees*, arXiv preprint arXiv:1312.6086, (2013).

[4] D. Kingma and J. Ba, *Adam: A method for stochastic optimization*, arXiv preprint arXiv:1412.6980, (2014).

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by back-propagating errors*, Cognitive modeling, 5 (1988), p. 1.