XGBoost for Multi-Class Classification

Spencer Ebert

December 9, 2019

1 Introduction

In the past decade, machine learning techniques for analysis have become more prevalent and popular. Algorithms like neural networks, support vector machines, random forests, and boosting are used to predict values from complicated datasets. These algorithms are effective in finding complicated relationships between predictors and find their effectiveness in big data. Of the machine learning techniques, Extreme Gradient Boosting (XGBoost) has seen a surge in popularity. XGBoost effectively utilizes gradient boosting to combine weak learners for an overall prediction. XGBoost is lauded as one of the fastest techniques compared to the others and has been used to win competitions in Kaggle.

My purpose in this project is to utilize XGboost for multi-class classification and compare its prediction performance to random forests and linear classification functions using a simulation study. For the analysis, I use the famous Iris dataset from Ronald Fisher. It contains four morphological measurements for three different species of iris. The measurements given are sepal and petal, length and width. There are 50 observations for each species of iris: setosa, virginica, and versicolor. This dataset is ideal for machine learning techniques on multiple classes because it contains three responses for species. I use XGBoost, random forests, and linear classification functions to train on the data and predict a particular species based on the four predictors. Overall, my goal in using these techniques is to examine prediction performance, not a description on the variables that best separate the classes. Also, my main focus is on XGBoost so I will focus most of the methodology on the XGBoost algorithm and give short descriptions for the others.

2 Methods

2.1 Regression Trees

To understand the XGBoost algorithm, one must first have a basic understanding of regression trees. The main idea behind a regression tree is to split the predictor space $\boldsymbol{X}$ into K partitions and assign a value to each observation in different partitions that predicts the response Y. The process of splitting the space into K partitions uses recursive binary splitting. Basically, all observations are grouped together at the top and will get split into two spaces based on some criteria for the first step. This process of splitting into two groups (binary) continues for the remaining steps (recursive).

To predict a response y_i, each partition of the space is given a constant c_k. Each observation x_i that lands in that particular partition is assigned c_k as the constant, which is the prediction for the response. This is represented in equation 1 where $f(x_i)$ is the predicted value given x_i and R_k is the k^{th} partition.

$$f(x_i) = \sum_{k=1}^{K} c_k I(x_i \in R_k) \tag{1}$$

The question now comes on how to best split the observations based on the predictor space into K partitions. Since it is computationally infeasible to consider every possible partition of the predictor space, we utilize recursive binary splitting to find an optimum partition. We start with every observation in the same space and then split into two spaces R_1 and R_2 based on a criteria for the j^{th} predictor of $\boldsymbol{X}$ based on its value s.

$$R_1(j,s) = \{\boldsymbol{X}|\boldsymbol{X}_j \leq s\} \quad \text{and} \quad R_2(j,s) = \{\boldsymbol{X}|\boldsymbol{X}_j < s\} \tag{2}$$

To decide on the best splitting criteria, you minimize the squared error loss for all possible combinations of j and s by following equation 3.

$$\min_{j,s}\left[\min_{c_1}\sum_{x_i\in R_1(j,s)}(y_i - c_1)^2 + \min_{c_2}\sum_{x_i\in R_2(j,s)}(y_i - c_2)^2\right] \tag{3}$$

c_1 and c_2 are minimized at the average of each point in the partition R_1 and R_2 respectively.

This process is then repeated for resulting partitions R_1 and R_2 and continued until you have reached the desired number of partitions. One of the main challenges of regression trees is the tendency to overfit the data. Methods like pruning and penalties for complexity to the model are used to help mitigate the risk. For the purposes of XGBoost, we utilize a specified number of splits for each tree so further information on methods are not discussed here.

2.2 Gradient Boosting

The main idea behind gradient boosting for classification is to combine weak learners to create a strong learner. Weak learners are algorithms that do better at predicting values than by pure chance. For the Iris data, that would result in prediction performance that is better than 33%. Based on the previous weak learners, we build another weak learner and combine the two. The process of building on the previous process is referred to as boosting. The overall goal is to train an algorithm that can take a new observation and predict the class based on the gradient boosted trees.

Other methods also use the combination of multiple trees to come up with a prediction. Adaptive boosting is an example where each successive tree is built off of the resulting tree by increasing weights for observations that were missclassified in the previous tree. Gradient boosting instead builds trees based on the residuals of the previous tree. Hence I discussed the use of regression trees instead of classification trees. The residuals in the case for classification is the observed value minus the predicted probability. By building a new tree based on the previous trees residuals, the algorithm focuses on where the previous step didn't do too well. By recursively building trees the residuals will be reduced and the model will fit the data better.

Gradient boosting starts with an initial prediction value $F_0(x)$ that is calculated by minimizing the loss function for the current model. In the case for multiple classification, the loss function is defined in equation 4.

$$L(y, p(x)) = -\sum_{g=1}^{3} I(y = W_g)\log(p_g(x)) \tag{4}$$

This loss function is the deviance of a multinomial distribution where $I(y = W_g)$ is the indicator function if the response y is in the subset W_g and $p_g(k)$ is the probability of being in that space. For the case of Iris data, W_g would represent the 3 species and $p_g(k)$ would represent the probability of the observation being in that space.

From this initial step $F_0(x)$, we want to move in a direction that minimizes the loss and improve where the initial guess didn't perform well. To do this, pseudo residuals are calculated. The residuals are equal to the negative partial derivative of the loss function with respect to $F_0(x)$ and probability $p(x)$ in the loss function equal to the previous prediction $F_0(x)$ (See step 2.a.i in the following algorithm). It can be shown that the resulting partial derivative equals $I(y_i = W_g) - p_g(k)$. We fit a regression tree to the residuals as discussed earlier to partition the predictor space into R_k groups. From these groups we assign a value $\gamma_{j,m}$ to each resulting partition in the residual tree, and it is calculated in equation 5.

$$\hat{\gamma}_{j,m} = \arg\min_{\gamma_{j,m}} \sum_{x_i \in R_{j,m}} L(y_i, F_{m-1}(x_i) + \gamma_{j,m}) \tag{5}$$

$F_{m-1}(x_i)$ is the predicted algorithm for the previous step given a value x_i.

With that new value calculated we update the prediction function and update again. A technical description of the algorithm is given as follows.

1. Initialize model with a constant value $F_0(x) = \arg\min_\gamma L(y_i, \gamma)$
2. for m in $1, ..., M$

(a) for g in $1, ..., 3$ (Note: Typically only one run through is necessary in boosting here. But since there are 3 classes these next steps have to be run 3 times for M iterations.)

   i. Compute $-h_{igm} = \left[\frac{\partial L(y_i, F_1(x_i), F_2(x_i), F_3(x_i))}{\partial F_g(x_i)}\right]_{F(x)=F_{m-1}(x)} = I(y_i = W_g) - p_g(x_i)$ for $i = 1, ..., N$

   ii. Fit a regression tree to the targets $-h_{igm}$ giving terminal regions $R_{jm}, j = 1, 2, ..., J_m$

   iii. For $j = 1, ..., J_m$ compute $\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$

   iv. Update $F_{m,g}(x) = F_{m-1,g}(x) + \nu \sum_{j=1}^{Jm} \gamma_{jm} I(x \in R_{jm})$

3. Output $\hat{F}_g(x) = F_{M,g}(x)$

For our case of three classes, we output three different tree expansions $\hat{F}_k(x)$. In the algorithm ν is the learning rate for each iteration. We don't want to overfit the data so instead of taking big steps in the right direction we take a lot of small steps of size ν in the correct direction. ν is often referred to as the learning rate in the algorithm. Typically you want values around 0.1.

By taking the gradient with respect to the previous function at each step (Step 2.a.ii), we move in the direction that minimizes the current loss function. This gradient calculation is considered as the pseudo-residuals. XGBoost follows the gradient boosting framework, and utilizes some techniques to speed up the process. XGBoost utilizes regulization, parallel computing, and the second derivative of the loss function to speed up the process and help reduce the chance of overfitting.

2.3 Random Forests

To observe how XGBoost compares to other methods, I compared it to random forests and linear classification function methods. Random forest methods use a collection of decision trees to come up with a prediction. It is considered an ensemble method. A single classification trees tends to overfit data so random forests utilize a large number of single classification trees and assign the class as the majority vote from the trees. Instead of just using the same data every time for building the tree, you build the tree off of bootstrapped samples from the training set and you also take a random subset of predictors from the predictor space. By following these two procedures, you minimize correlation between each trees results and you look at many different combinations of the predictor space. More detailed information on random forests and how they can be scaled to multiple classification can be found in further reading.

2.4 Linear Classification Function

The last method I used to compare against XGBoost is the linear classification function. The main idea is that the data for each class (setosa, virginica, and versicolor in iris data) follows a multivariate normal distribution all with equal covariance matrices Σ and different mean vectors μ_i.

$$X_i \sim N(\mu_i, \Sigma) \quad \text{for} \quad i = 1, 2, 3 \tag{6}$$

Assigning a new observation to one of three classes is based on the Mahalanobis distance of that observation to each of the estimated mean vectors. The observation is assigned to the class with the minimum distance from the estimated mean vector. If the observation is closest to the mean of the first class, then it is assigned to that class. Adjusting the formula for Mahalanobis distance gives equation 7.

$$L_i(\boldsymbol{x}) = \bar{\boldsymbol{x}}_i' S_{pl}^{-1} \boldsymbol{x} - \frac{1}{2} \bar{\boldsymbol{x}}_i' S_{pl}^{-1} \bar{\boldsymbol{x}}_i \quad \text{for} \quad i = 1, 2, 3 \tag{7}$$

Since this equation is multiplied by a negative, we assign the observation $\boldsymbol{x}$ to the class that gives it the highest value, rather than the minimum. Further classification rules are used if the proportion of observations in each classes differs. The number of each class is similar for all of the classes thus the priors are equal in the analysis.

3 Iris Analysis

I performed XGBoost, random forests, and linear classification function to the iris data by randomly splitting the data into test and training datasets, with the test set being 50 observations and the training 100 observations. I fit each model to the training data and used the model fit to predict the test responses. I repeated the process 1000 times with different test and training datasets and averaged over the correct prediction rate for each method.

4 Simulation Study

To compare the three methods, I performed a simulation study that builds datasets based on a set of four parameters and I performed all three methods on the datasets.

4.1 Creating the Data

I wanted to see how increasing noise σ^2, number of predictors x_n, correlation between the predictors ρ, and number of observations n affected the performance for each method. For each class, I took n draws from a multivariate normal distribution with different mean vectors and similar covariance matrices. I label the three groups 0, 1, and 2. I then combine all of the groups for that particular combination into one dataset. For group 0, I create a mean vector of 0's with length x_n. For group 1 the mean vector is 1's with length x_n. Likewise group 2 has same length but instead 2's as the mean. By choosing these values, I make the distance between the mean vectors similar, with group 2 and 0 the farthest from each other. I created a similar covariance matrix for each of the draws from the multivariate normal distribution. The covariance matrix is given in equation 8. The number of rows and columns for the covariance matrix is x_n.

$$\Sigma_0 = \Sigma_1 = \Sigma_2 = \sigma^2 \begin{bmatrix} 1 & \rho & \rho \\ \rho & 1 & \rho \\ \rho & \rho & 1 \end{bmatrix} \tag{8}$$

I also wanted to look at how the methods performed if the covariance matrices differed from each other. To do this, I kept the same combination of parameters but I adjusted the calculation for the covariance matrix. The different covariance matrices are given in 9

$$(\sigma_0^2, \sigma_1^2, \sigma_2^2) = \sigma^2(0.5, 1, 3)$$

$$\Sigma_0 = \sigma_0^2 \begin{bmatrix} 1 & \rho & \rho \\ \rho & 1 & \rho \\ \rho & \rho & 1 \end{bmatrix}$$

$$\Sigma_1 = \sigma_1^2 \begin{bmatrix} 1 & -\rho & -\rho \\ -\rho & 1 & -\rho \\ -\rho & -\rho & 1 \end{bmatrix}$$

$$\Sigma_2 = \sigma_2^2 \begin{bmatrix} 1 & \rho & \rho \\ \rho & 1 & \rho \\ \rho & \rho & 1 \end{bmatrix} \quad (9)$$

4.2 Analysis

I looked at all 144 possible combinations for $x_n = 2, 4, 8$; $\sigma^2 = 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 20$; $\rho = 0.01, 0.5, 0.99$; and $n = 50, 100$. The process for the simulation analysis is as follows:

1. Generate data from the i^{th} combination of parameters
2. Split data into test and training sets; 2/3 of the data in training and the remaining in test
3. Fit all 3 models on the training dataset
4. Predict class for the testing dataset
5. Compute overall prediction performance for each method
6. Repeat steps 1-5 100 times using the same dataset parameters and average out the overall prediction performance for each method
7. Save the average performance for each method
8. Repeat steps 1-7 144 times using each combination of the parameters

This simulation study took approximately 5 hours to run because I am generating 14400 datasets and fitting 3 models to those datasets.

The process is repeated for the different covariance matrices, but instead of using similar covariance matrices I use the covariance matrices found in equation 9. Results from the similar matrices should favor the linear discriminant function, but with the second method the relationship between the classes is more complicated.

5 Results

5.1 Iris Data Results

The correct classification results from fitting all three methods to the iris data are found in table 1.

Table 1: Correct classification on Iris data

XGBoost	Random Forest	Lin Function
0.9535	0.9504	0.9780

As can be seen in the results for the iris data, the linear discriminant function performed the best out of the three methods. This may be due to the predictors following a similar covariance between the classes. Overall, the prediction results for all three methods are pretty high which implies that the classes are fairly separated in these data.

5.2 Simulation Results

There are 144 combinations of parameters for both methods of simulating data. Because of the high number of possible combinations I only report the top findings in table 2.

Some of the key findings from the simulation results in table 2 are shown below.

1. As σ^2 increases, the overall prediction performance for all three methods goes down. When σ^2 is small, the data don't overlap as much, which gives a strong separation between the classes. This is consistent for both methods of constructing the covariance matrices.

Table 2: Simulation study results for multiple combinations of covariance matrices. There are a total of 144 possible combinations, but these combinations are sufficient for this purpose.

Parameters				Similar Covariances			Different Covariances		
x_n	σ^2	ρ	n	Boost	RF	LC	Boost	RF	LC
4.00	0.05	0.50	50.00	0.98	1.00	1.00	0.97	0.99	0.98
4.00	0.50	0.50	50.00	0.69	0.71	0.74	0.77	0.83	0.83
4.00	10.00	0.50	50.00	0.36	0.37	0.41	0.61	0.65	0.46
2.00	0.50	0.50	100.00	0.67	0.67	0.71	0.77	0.77	0.79
4.00	0.50	0.50	100.00	0.71	0.72	0.75	0.82	0.86	0.86
8.00	0.50	0.50	100.00	0.72	0.75	0.75	0.81	0.88	0.85
8.00	1.00	0.50	100.00	0.61	0.64	0.64	0.74	0.82	0.75
8.00	5.00	0.50	100.00	0.43	0.45	0.46	0.65	0.75	0.54

2. As the number of predictors x_n goes up, the prediction performance for all three methods seems to get a little better, except for when x_n goes from 4 to 8. The number of predictors in this case doesn't seem to have a strong impact on the performance of these methods.

3. As the number of observations n goes up, the prediction performance gets better for both constructions of the matrix. This makes sense because we have more observations in helping train the model to better understand the data.

4. For large σ^2, boosting and random forests do better than linear classification for the different covariance constructions. As the noise increases in a complex model linear classification does a poor job at separating the data, whereas machine learning techniques are able to pick up some of the intricacies in the data structure.

6 Conclusion

Overall, I examined the predictive effectiveness for XGboosting, random forests, and linear discriminant classification, and found that these methods performed about the same. For the iris data, linear discriminant classification performed the best compared to the other two. This implies the underlying iris data structure is fairly simple. The simulation study confirms this.

None of the modeling are better than the other in all situations. Choosing a model for multi-class prediction requires understanding of the nature in the data. If one believes that the underlying data structure is fairly simple (equal covariance matrices), then using the simple algorithm for linear classification is very effective and simple compared to boosting, and random forests. Whereas if the data are complicated, then machine learning techniques outperform simpler models. Popular machine learning techniques definitely have their place, but they are not necessarily a magic bullet for all problems.

Moving forward, I want to experiment with different group sizes. How do these methods compare if the proportion in each group is different? I also want to do a stronger cross validation for XGBoost parameters to choose the most optimal. learning rate and tree depth.