XGBoost

MATH 154 PO Final Project

Sam Malik

Friday, Dec. 16th at 11:59pm

Note: I took the opportunity of the final project to dive deeper into the world of gradient boosted trees and the ubiquitous XGBoost

1 Introduction

XGBoost, short for "Extreme Gradient Boosting," is one of the most popular and powerful open-source gradient boosting libraries available today. It is a supervised machine learning algorithm that uses decision tree ensemble learning and gradient boosting to perform either regression or classification. Gradient Boosting is an ensemble learning technique that adds the predictive power of multiple weak decision trees that, when combined, create a strong, accurate model. XGBoost is known for its efficiency, flexibility, regularization, and predictive power on large and sparse datasets.

To truly grasp the predictive power of this model, this paper will compare its abilities to four other popular regressions models: Random Forest Regression, LASSO regression, Gaussian Process Regression, and Support Vector Regression.

2 Gradient Boosted Trees and XGBoost

The following derivation of the core structure of XGBoost is heavily adopted from:

1. "How to Understand XGBoost" https://blog.mattbowers.dev/how-to-understand-xgboost
2. "XGBoost: A Scalable Tree Boosting System" https://arxiv.org/abs/1603.02754

Suppose the dataset with n instances and m features in the form (x_i, y_i). Assume the additive model $\hat{y_i} = F(x_i)$, that it adds K weighted functions, $F(x_i) = b + \eta \sum_{i=1}^{K} h_i(x_i)$, where b is the initial prediction (typically 0.5), and η is the learning rate (by default it is 0.3). In the gradient boosted trees algorithm, we limit each of h_i to be a weak learner that loosely learns the residuals of the previous model it is being added to. Each new addition therefore tries to correct the mistakes of the previous model until the residuals converge or the model reaches a maximum

number of additions. The algorithm behind XGBoost adopts this additive model. Recursively, the formulation for XGBoost becomes $\hat{y_i}^k = F_k(x_i) = F_{k-1}(x_i) + \eta h_k(x_i)$. Each regression tree h_i contains a continuous output, w_j, at each of its T leaves, where w_j is the output at the jth leaf of a specific tree. The final prediction for x_i is the weighted sum of the outputs of all the trees.

To generate the set of at most k trees to use in the final model, the XGBoost algorithm attempts to minimizes the regularized loss function:

$$\mathcal{L} = \sum_i loss(F_K(x_i), y_i) + \sum_k R(h_k) \tag{1}$$

$loss(F_K(x_i), y_i)$ is the loss between the prediction value and the actual y_i and the added $R(h_k)$ is a regularization term to penalize fitting to the residuals exactly, which prevents over-fitting. In the literature, $R(h_k) = \gamma T + \frac{1}{2}\lambda||w||^2$, where T is the number of leaves in each tree, and w is the output value at each leaf. This regularization term attempts to limit the learning ability of each tree by allowing it to fit only part of the signal from the residuals. The effect of the hyper-parameters γ and λ will become clear later in this section.

As the total loss in Eq. 1 is not optimizable using traditional methods, the XGBoost algorithm takes a greedy approach. The model is trained in an additive matter at each time step, adding a new tree residual predictor h_k that most improves the model according to 1. At any given time step k (also known as the kth iteration of the algorithm), the goal is to add a tree h_k that minimizes the objective:

$$\mathcal{L}_k = \sum_i loss(F_k(x_i), y_i) + R(h_k) \tag{2}$$

As $F_k(x_i) = F_{k-1}(x_i) + h_k(x_i)$, we perform the appropriate substitution to obtain

$$\mathcal{L}_k = \sum_i loss(F_{k-1}(x_i) + h_k(x_i), y_i) + R(h_k) \tag{3}$$

As the above loss function is hard to optimize using tradition methods, we approximate the first term using a second order taylor approximation to grant us the ability to optimize it using traditional methods in Euclidean space. The formula for the second order taylor expansion for a function $f(x)$ at $x = a$ is given by

$$f(a+u) \approx f(a) + f'(a)u + \frac{1}{2}f''(a)u^2 \tag{4}$$

Applying this to the first term in $\mathcal{L}_k$ with $a = F_{k-1}(x_i)$ and $u = h_k(x_i)$,

$$loss(F_{k-1}(x_i)+h_k(x_i), y_i) \approx loss(F_{k-1}(x_i), y_i) + \frac{\partial loss(F_{k-1}(x_i), y_i)}{\partial F_{k-1}(x_i)} \cdot h_k(x_i) + \frac{1}{2}\frac{\partial^2 loss(F_{k-1}(x_i), y_i)}{\partial F_{k-1}(x_i)^2} \cdot h_k(x_i)^2 \tag{5}$$

The literature denotes

$$g_i = \frac{\partial loss(F_{k-1}(x_i), y_i)}{\partial F_{k-1}(x_i)} \tag{6}$$

and,

$$h_i = \frac{\partial^2 loss(F_{k-1}(x_i), y_i)}{\partial F_{k-1}(x_i)^2} \tag{7}$$

the first and second order partial derivatives of the loss function with respect to the current predictions. Substituting Eq. 6 and Eq. 7, and the whole approximation into Eq. 3, we obtain

$$\mathcal{L}_k \approx \sum_i loss(F_{k-1}(x_i), y_i) + g_i h_k(x_i) + \frac{1}{2} h_i h_k(x_i)^2 + R(h_k) \tag{8}$$

$loss(F_{k-1}(x_i), y_i)$ is a constant with respect to the choice of the tree we're trying to optimize, $h_k(x_i)$, so we can remove it from the objective function.

$$\mathcal{L}_k \approx \sum_i \big[g_i h_k(x_i) + \frac{1}{2} h_i h_k(x_i)^2 \big] + R(h_k) \tag{9}$$

Also notice, that all instances x_i that are sorted into a given leaf of a tree h_k, the prediction will be the same. Call I_j the set of x_i's that fall in the jth leaf of the tree. Therefore, for all x_i in I_j, $h_k(x_i) = w_j$ for any instance x_i that falls into the leaf that corresponds to the value w_j. As each of the n instances must fall into one of the T leaves, and each leaf predicts a value of w_j, The sum therefore becomes:

$$\mathcal{L}_k \approx \sum_{j=1}^{T} \big[\sum_{i \in I_j} g_i w_j + \frac{1}{2} \sum_{i \in I_j} h_i w_j^2 \big] + R(h_k) \tag{10}$$

Expanding the regularization term $R(h_k) = \gamma T + \frac{1}{2}\lambda ||w||^2$, we see that $||w||^2$ is the sum of the squares of all of the outputs from the T leaves in the tree.

$$\mathcal{L}_k \approx \sum_{j=1}^{T} \big[\sum_{i \in I_j} g_i w_j + \frac{1}{2} \sum_{i \in I_j} h_i w_j^2 \big] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \tag{11}$$

To obtain the optimal output value w_j for each specific leaf node leaf, we take the derivative of this loss function for a specific leaf node and equate it to 0.

$$\begin{aligned} \frac{d}{dw_j} \big[\sum_{i \in I_j} g_i w_j + \frac{1}{2} \sum_{i \in I_j} h_i w_j^2 + \frac{1}{2} \lambda w_j^2 \big] &= \sum_{i \in I_j} g_i + w_j \sum_{i \in I_j} h_i + \lambda w_j \\ &= \sum_{i \in I_j} g_i + w_j \big(\lambda + \sum_{i \in I_j} h_i \big) \\ &= 0 \end{aligned} \tag{12}$$

Solving for this equation yields that the optimal output for each leaf node j in the tree h_k is

$$w_j^* = \frac{-\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{13}$$

In context, it is the sum of the first order derivatives of the loss function with respect to th current predictions divided by the sum of the second order derivative of the loss function plus the regularization term lambda. Here, it is clear to see that if $\lambda = 0$, we have the equivalent of no regularization, and the output for w_j would just be the residual. Increasing the regularization penalty λ pulls the optimal output value closer and closer to 0, which serves the primary goal of regularization. The corresponding optimal loss value by adding a new tree (h_k) is therefore given

by plugging the optimal value for w_j back into the equation, which yields the follow equation for a given tree structure

$$\mathcal{L}_k = -\frac{1}{2}\sum_{j=1}^{T}\frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T \tag{14}$$

In any particular tree, the way to make the optimal split is by comparing the objective function for the tree structure before making the split and after making the split. The optimal split produces the maximum reduction in the objective. Suppose to have a current node with instances x_i in I. The current value of the objective function is

$$L_{bs} = -\frac{1}{2}\frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda} + \gamma \tag{15}$$

Once we make a decision choice for the split at that node, we split the data into two subsets I_L and I_R. As all of the data originally in I not falls into either I_L or I_R, the loss after the split is equal to

$$L_{as} = L(I_L) + L(I_R) = -\frac{1}{2}\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \gamma - \frac{1}{2}\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \gamma \tag{16}$$

The total gain from splitting from I into I_L and I_R is therefore the difference in the loss.

$$\begin{aligned} L_{bs} - L_{as} &= -\frac{1}{2}\frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda} + \gamma + \frac{1}{2}\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \gamma + \frac{1}{2}\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \gamma \\ &= \frac{1}{2}\Big[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\Big] - \gamma \end{aligned} \tag{17}$$

XGBoost defined $L_{bs} - L_{as}$ as the "gain" from splitting. At every node, the algorithm searches for the split that maximizes the "gain". That is, make L_{as} as small as possible. If the loss after splitting is always greater than before splitting, $L_{bs} < L_{as}$, the gain is always negative and the algorithm does not split. It only makes sense to split if it realizes a positive gain. The hyper-parameter γ directly controls this; a larger γ subtracts more from the "gain," which makes splitting into a deeper, more complex tree more difficult. The hyper-parameter γ is therefore a regularizing penalty on the size and complexity of the tree fitting to the residuals of the previous model. γ represents the minimum loss split parameter.

At each of the k iterations of the algorithm, XGBoost will find a tree h_k that fits to the residuals of the previous model. Each tree greedily searches for the best split that maximizes gain at each node, $L_{bs} - L_{as}$, and only splits if the gain is strictly positive.

Interestingly, the loss function was kept general through this whole process, so the user of XGBoost can specify their own loss function. In the specific case where MSE is the loss function, $loss(F_k(x_i), y_i) = \frac{1}{2}(y_i - F_k(x_i))^2$, g_i and h_i become

$$\begin{aligned} g_i &= \frac{\partial}{\partial F_{k-1}(x_i)}\frac{1}{2}(y_i - F_k(x_i))^2 \\ &= -(y_i - F_k(x_i)) \end{aligned} \tag{18}$$

and,

$$\begin{aligned} h_i &= \frac{\partial^2}{\partial F_{k-1}(x_i)^2} \frac{1}{2}(y_i - F_k(x_i))^2 \\ &= 1 \end{aligned} \tag{19}$$

The optimal output value at a given leaf node is therefore

$$\begin{aligned} w_j^* &= \frac{-\sum_{i \in I_j} -(y_i - F_k(x_i))}{\sum_{i \in I_j} 1 + \lambda} \\ &= \frac{\sum_{i \in I_j}(y_i - F_k(x_i))}{|I_j| + \lambda} \\ &= \frac{\text{Sum of Residuals in leaf}}{\text{Number of residuals in leaf} + \lambda} \end{aligned} \tag{20}$$

It is the sum of the residuals in that leaf divided by the number of residuals in that leaf plus λ. With $\lambda = 0$, the output for a given is leaf is just the average residual in that leaf.

Through similar calculation, the gain formula for a node and split reduces to

$$\frac{1}{2}\Big[\frac{\text{Sum of Residuals in } I_L \text{ squared}}{\text{Number of residuals in } I_L + \lambda} + \frac{\text{Sum of Residuals in } I_R \text{ squared}}{\text{Number of residuals in } I_R + \lambda} - \frac{\text{Sum of Residuals in } I \text{ squared}}{\text{Number of residuals in } I + \lambda}\Big] - \gamma \tag{21}$$

To reiterate, the final prediction for a new instance $\mathbf{x}$ will be

$$\hat{y_i} = F(\mathbf{x}) = b + \eta \sum_{k=1}^{K} h_k(\mathbf{x}) \tag{22}$$

where b is the initial prediction (typically 0.5), and η is the learning rate.

There are other idiosyncrasies of the XGBoost algorithm that make it fast that are out of the scope of this paper. These include the approximate greedy algorithm and sparsity aware split finding for large datasets and efficient training. All in all, XGBoost introduces two new regularization functions to prevent the new learners from fitting too rigidly to the signal of the residuals, λ and γ.

3 Data

To examine the effectiveness of XGBoost, I compare it to other regression techniques such as Support Vector Regression, Random Forest Regression, LASSO Regression, and Gaussian Process Regression. A dataset about concrete compressive strength is preprocessed for missing values and outliers, and is then split into training and testing set with an 80/20 split. A model corresponding to each of the aforementioned regression techniques is trained on the training set with 4-fold cross validation, then evaluated on the unseen testing dataset using a normalized root mean squared error (normalized by the standard deviation of the testing set y_i values)

3.1 Dataset

This analysis uses a dataset to predict Concrete Compressive Strength, measured in MPa (Mega Pascals). The dataset can be found here

(https://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength). Concrete is a crucial material in civil engineering. The dataset authors describe the compressive strength of concrete to be a non-linear function of ingredients and its age. The dataset includes eight input features and one target variable. The eight input variables are comprised of measurements of seven different ingredients and the age of the concrete. The seven ingredients are all quantitative, continuous variables measured in kg/m^3, while the age descriptor is discrete value measured in Days, ranging from 1 to 365. The goal of the dataset is to build a regression model to model the relationship between the 8 features and the concrete compressive strength. Before preprocessing, there are 1030 instances with no missing values.

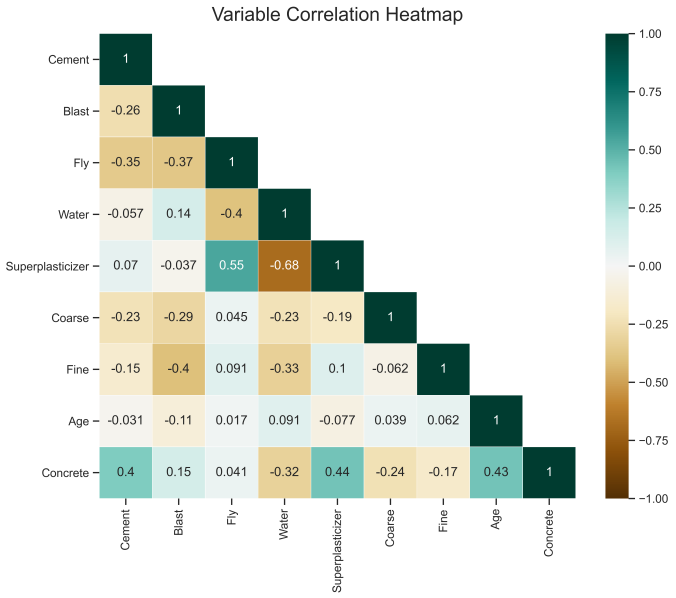

Figure 1: Correlation Heatmap

In Figure 1, we see that most all variables have small correlation, with the exception of Superplasticizer and Water content, which have a clear negative correlation, Superplasticizer and Fly which have a positive correlation. The output variable, Concrete is not strongly correlated with many of the explanatory variables, which suggests that relationship between the input features and the output variable is non-linear in just the input features alone.

3.2 Data pre-processing

Pre-processing the data involved handling missing values and removing outliers from the dataset. As the dataset had no missing values, this step could be skipped. To handle outliers, I fit an Isolation Forest outlier detection model from *sci-kit learn* to the entire dataset. In total, 225

instances were considered outliers, so they were discarded. The cleaned dataset was left with 805 instances to train and test the model on. I split the remaining instance into a training and testing set with an 80/20 split.

4 Results

The following section presents the results from training five different regression models on a given train/test partition of the data. The primary metrics used were the root mean squared error, which is given by $RMSE = \sqrt{\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}}$. One of the models below, support vector regression, requires each feature to range from 0 to 1, so all features were scaled, including the output variable. Therefore, the RMSE error for the SVR model would be artificially low due to the scale of its data. To account for this, we use a standard deviation normalized RMSE to compare the models regression performance, $NRMSE = \frac{RMSE}{\sigma_y}$.

4.1 XGBoost

The hyperparameters for this model included the number of trees to use in the estimate, the learning rate η, the coefficient for the L2 regularization λ, and the regularization parameter γ. After grid searching over a range of these values by performing 4-fold cross validation to minimize the root mean squared error, the optimal cross validation score was 4.54 and the optimal set of parameters was $\eta = 0.4, \lambda = 15, \gamma = 0, \text{n\_estimators} = 400$. When predicting on the test set, the RMSE was 4.44 and the NRMSE was 0.318.

4.2 Random Forest Regression

With the random forest regression model, I performed 4-fold cross validation over the number of decision trees to use in the estimate. Each decision tree in the random forest used the Gini Impurity score to search for the optimal split. The maximum number of features to be considered at each split is the square root of the number of features as a regularization feature. I performed cross validation over the number of trees in the regression model with the increments of 7 starting from 10, ending at 200. After cross validating to find the lowest root mean squared error, the optimal cross validation score was 5.05 and the optimal number of trees was 150. When predicting the test set, the RMSE was 4.75 and the NRMSE was equal to 0.339.

4.3 Gaussian Process Regression

With the Gaussian Process regression, the primary tune-able parameters were the kernels I could use. Alongside the choice of kernel, I also had to consider each of the kernel's hyparameters. However, the data science library I am using, sci-kit learn, uses built in cross validation to locate the optimal hyparameters for a given kernel of the Gaussian process regression: the kernels are optimized during fitting to the training data. So, I primarily focused on the choice of kernel to

yield the best result. I considered the Radial Basis Function Kernel, the Dot Product Kernel, the White Kernel, and the sum of any two those kernels. The following table presents the choice kernel alongside its optimal parameters for the regression and its NRMSE on the testing set.

Kernel	*Optimal Hyperparameters*	*Test Set RMSE*	*Test Set NRMSE*
RBF	$l = 10^{-5}$	31.79	2.27
Dot	$\sigma_0 = 1$	7.88	0.56
White	noise_level=1.33×10^3	34.29	2.45
RBF + Dot	$l = 1, \sigma_0 = 1$	**6.47**	**0.46**
RBF + White	$l = 10^5$, noise_level=385	17.51	1.25
Dot + White	$\sigma_0 = 8.82$, noise_level=71.2	7.73	0.55

Table 1: Optimal Hyperparameters and NRMSE for different Kernels in the Gaussian Process Regression

Here, the combination of kernel that resulted in the lowest NRMSE on the test set was the sum of the Radial Basis Function and the Dot Product Kernel.

4.4 LASSO Regression

When fitting the LASSO regression, the only hyper parameter to cross validate for is the coefficient of the sum of the L1 norms of the OLS β coefficients. I cross validated over the range from 0 to 3 in increments of 0.01. After cross validation, the optimal coefficient was $\alpha = 0.1$ with a cross validated average RMSE of 8.501. On the testing set, the optimal model achieved a RMSE of 7.69 and a NRMSE of 0.55.

4.5 Support Vector Regression

Lastly, as the last regression for comparison, I trained a support vector regression model to the data. For this task, however, the data had to be scaled. Each feature in the training set was independently scaled using a MinMax scaler to force the data between 0 and 1. The model used a polynomial kernel. With this kernel, the hyper-parameters to be cross validated are the degree (for the polynomial kernel), d, the cost coefficient, C, and epsilon, ϵ. The degree coefficient range from 1 to 15 in increments of 3, the cost coefficient ranged from 1 to 25 in increments of 3, and he epsilon coefficient ranges from 0.1 to 0.5 in increments of 0.1. After cross validation, the optimal parameters were $d = 4, C = 1, \epsilon = 0.1$ with a polynomial kernel. The optimal cross validated RMSE from the optimal parameters was 0.0789, but the data was scaled for this model. The RMSE on the testing set was 0.072 and the NRMSE on the testing set was 0.4.

5 Discussion

In Figure 2 we see that all of the models have testing set root mean squared error values that are lower than the cross validation score, implying that each of the models fit to the data's signal, and

Model (Regression)	*Cross Validation RMSE*	*Test set RMSE*	*Test set NRMSE*
XGBoost	**4.54**	**4.44**	**0.318**
Random Forest	5.05	4.75	0.332
Gaussian Process	N/A	6.47	0.46
LASSO	8.501	7.69	0.55
Support Vector Regression	0.0789	0.072	0.4

Table 2: Test NRMSE for each optimized regression model

did not overfit. The Gaussian Process regression CV RMSE was not provided, but the sklearn library performs cross validation on all parameters to prevent overfitting.

Out of all these regression models, XGBoost realized the lowest NRMSE on the testing set; it had the best fit to the signal of the data.

6 Conclusion

Overall, Extreme Gradient Boost, "XGBoost" is a very powerful ensemble learning algorithm that is widely used due to its strong performance, efficiency, and flexibility. Its use of gradient boosted methods and its regularization techniques allow the model to isolate the signal of a data very effective.