Hands-On Gradient Boosting with XGBoost

by Corey Wade & Kevin Glynn

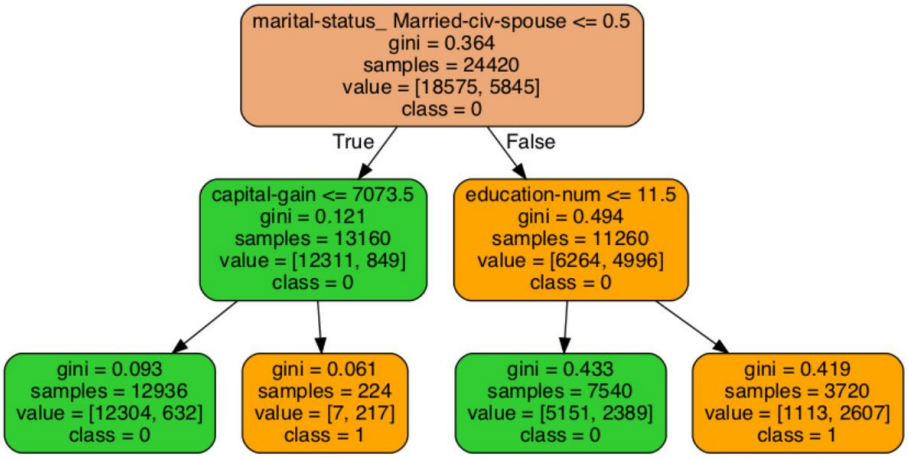
Decision Trees in Depth

XGBoost is an ***ensemble method***, meaning that it is composed of different ML models that combine to work together. The individual models that make up the ensemble are called ***base learners***. Decision trees are the most commonly used XGBoost base learner. They split the data by asking questions about the columns; building decision trees is like playing a game of 20 Questions.

The process of splitting data into new groups via branching continues until the algorithm reaches a desired level of accuracy. A decision tree can create thousands of branches until it uniquely maps each sample to the correct target in the training set – so the training set can have 100% accuracy. Such a model, however, will not generalize well to new data. Decision trees are prone to overfitting the data. A way to avoid this is to aggregate the predictions of many trees, a strategy employed by Random Forests and XGBoost.

Decision trees

So, decision trees work by splitting the data into *branches*. The branches are followed down to *leaves* where predictions are made.



The top of the tree is the root, the true/false arrows are branches, and the data points are nodes. At the end of the tree, the nodes are classified as trees. In the example above, at the root of the tree, marital status is a binary column – all values are 0 (negative) or 1 (positive). The first split is based on whether someone is married or not. The left side of the tree is the *true* branch, meaning the user is unmarried, etc.

Gini criterion

The second line of the root in the example above reads **gini=0.364**. This is the error method the decision tree uses to decide how splits should be made. The goal is to find a split that leads to the lowest error. A gini index of 0 means 0 errors, while a gini index of 1 means all errors. A gini index of 0.5, which shows an equal distribution of elements, means the

predictions are no better than random guessing. So, the closer to 0, the lower the error. At the root, a gini of 0.364 means the training set is imbalanced with 36.4% of class 1. The equation for the gini index is:

$$gini = 1 - \sum_{i=1}^{c} (pi)^2$$

Pi is the probability that the split results in the correct value, and c is the total number of classes: 2 in the preceding example. Another way of looking at this is that pi is the fraction of items in the set with the correct output label.

Samples, values, class

The root of the tree states that there are 24,420 samples (the total of the training set). The following line reads [18575, 5845]; since the ordering is 0 then 1, 18,575 samples have a value of 0 then 1, so 18,575 samples have a value of 0 (they make less than 50K) and 5,845 have a value of 1 (they make more than 50K). *[The set-up for this example was to predict whether someone makes over 50K].*

Following the *true* branch down the tree, in the left node in the second row, the split *capital_gain <= 7073.5* is applied to subsequent nodes. Of the 13,160 unmarried people (who came down this true branch), 12,311 have an income of less than 50K, while 849 have an income of more. The gini index is 0.121.

Leaves

The nodes at the end of the trees, the leaves, contain all final predictions. The far-left leaf has a gini index of 0.093, correctly predicting 12,304 of 12,938 cases, which is 95%. We are 95% confident that unmarried users with capital gains of less than 7,073.50 do not make more than 50K. Other leaves may be interpreted similarly.

Tuning decision tree hyperparameters

Hyperparameters are not the same as parameters. In ML, parameters are adjusted when the model is being tuned; the weights in linear and logistic regression are parameters adjusted during the build phase to minimize errors. Hyperparameters, by contrast, are chosen in advance of the build phase. **Note that hyperparameters should not be chosen in isolation**. Common hyperparameters:

Max_depth: Defines the depth of the tree, determined by the number of times splits are made. By default, there is no limit to this, which can lead to overfitting. By limiting max_depth to smaller numbers, variance is reduced, and the model generalizes better to new data.

Min_samples_leaf: Provides a restriction by increasing the number of samples that a leaf may have. When there are no restrictions, the default is 1, meaning that leaves may consist of unique samples (leading to overfitting). Increasing this reduces variance.

Max_leaf_nodes: Similar to above, except is specifies the total number of leaves. E.g., if this is 10, the model cannot have more than 10 leaves, but could have fewer.

Max_features: Instead of considering every possible feature for a split, it chooses from a select number of features each round. 'Auto' is the default, which provides no limitations. 'Sqrt' is the square root of the total number of features. 'Log2' is the log of the total number of features in base 2; 32 columns resolves to 5 since 2^5=32.

Min_samples_split: A limit to the number of samples required before a split is made. The default is 2, since two samples may be split into one sample each, ending as single leaves. If increased to 5, for example, no further splits are permitted for nodes with 5 samples or fewer.

Criterion: This provides the method the ML model uses to determine how splits should be made – it is the scoring method for splits. For each possible split, the criterion calculates a number for a possible split and compares it to other options. The split with the best score wins. Options for decision tree regressors are 'mse' and 'mae'. For classifiers, 'gini' and 'entropy' usually give similar results.

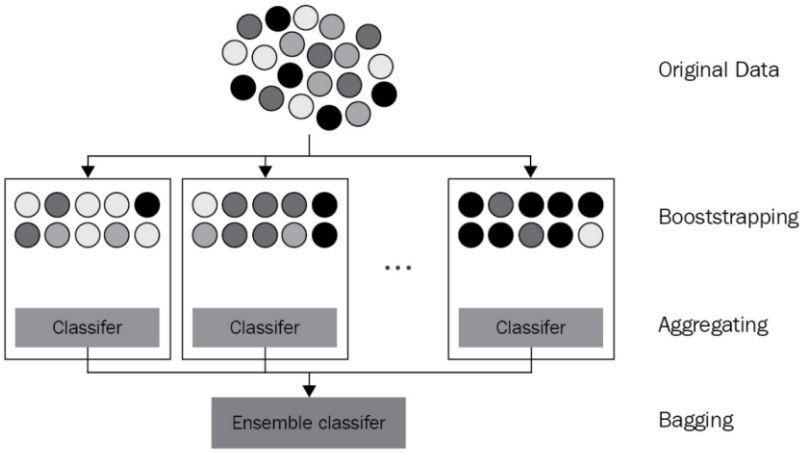
There are too many hyperparameters to consistently use them all. In the author's experience, ***max_depth, max_features, min_samples_leaf, max_leaf_nodes,*** and ***min_samples_split*** are often sufficient.

Bagging with Random Forests

Random forests are also ensembles of decision trees. The difference is that random forests combine via ***bagging***, while XGBoost combines trees via ***boosting***.

Bootstrap aggregation

Bootstrapping means sampling with replacement. Imagine you have a bag of 20 marbles. You are going to select 10, one at a time. Each time you select one, you put it back in the bag. It is extremely unlikely that you will pick the same marble 10 times. You will pick some marbles more than once, and some not at all.



With random forests, the bootstrapping occurs when each decision tree is made. If the decision trees all consisted of the same samples, the trees would give similar predictions, making the aggregate result similar to the individual tree. Instead, with random forests, the trees are built using bootstrapping. Mathematical estimations are that two-thirds of the samples for each tree are unique, and one-third include duplicates.

After the bootstrapping phases, each decision tree makes its own individual predictions. The result is a forest of trees whose predictions are aggregated into one final predictions using majority rules for classifiers and the average for regressors. In summary, a random forest aggregates the predictions of bootstrapped decision trees, which is known as bagging.

From Gradient Boosting to XGBoost

XGBoost is a unique form of gradient boosting, so we should learn how traditional gradient boosting works.

Boosting, as opposed to bagging, learns from the mistakes of individual trees. The general idea is to adjust new trees based on the errors of previous trees. In boosting, correcting errors for each new tree is a distinct approach from bagging. In a bagging model, new trees pay no attention to previous trees; new trees are built from scratch using bootstrapping, and the final model aggregates all individual trees. In boosting, however, each new tree is built from the previous tree. The trees do not operate in isolation; instead, they are built on top of one another.

AdaBoost

In AdaBoost, each new tree adjusts its weights based on the errors of previous trees. More attention is paid to predictions that went wrong by adjusting weights that affect those samples at a higher percentage. By learning from its mistakes, AdaBoost can transform weak learners into strong learners. A weak learner is an algorithm that performs barely better than chance, whereas a strong learner has learned a considerable amount from data and performs well.

The general idea behind boosting algorithms is to transform weak learners into strong learners. There is a purpose behind the weak start. Boosting works by focusing on iterative

error correction, *not* by establishing a strong baseline model. If the base model is too strong, the learning process is necessarily limited, thereby undermining the general strategy behind boosting models. Weak learners are transformed into strong learners through hundreds of iterations.

Gradient boosting

Gradient boosting uses a different approach than AdaBoost. While gradient boosting also adjusts based on incorrect predictions, it takes this idea one step further: gradient boosting fits each new tree entirely based on the errors of the previous tree's predictions. So, for each new tree, gradient boosting looks at the mistakes and then builds a new tree completely around these mistakes. The new tree doesn't care about the predictions that are already correct.

Building an ML algorithm that solely focuses on the errors requires a comprehensive method that sums errors to make accurate final predictions. This method leverages residuals, the difference between the model's predictions and actual values. The general idea:

"Gradient boosting computes the residuals of each tree's predictions and sums all the residuals to score the model."

Computing and *summing residuals* are at the core of XGBoost.

Residuals

The residuals are the difference between the errors and the predictions of a given model. In statistics, residuals are analyzed to determine how good a given linear regression model fits the data.

1. Bike rentals
   a) *Prediction*: 759
   b) *Result*: 799
   c) *Residual*: 799 - 759 = 40

So, residuals tell you how far the model's predictions are from reality. They can be positive or negative. In linear regression, the goal is to minimize the square of the residuals.

Building a gradient boosting model from scratch

1. Fit the data to the decision tree. This decision tree should not be fine-tuned for accuracy. We want a model that focuses on learning from errors, not a model that relies heavily on the base learner.
2. Make predictions with the training set (rather than the test set). This is done because to compute the residuals we need to compare the predictions while still in the training phase. The test phase of the model build comes at the end, after all the trees have been constructed.
3. Compute the residuals: the difference between the predictions and the target column.

```
y_train_pred = tree_1.predict(X_train)
```

4. Fit a new tree on the residuals, which is different than fitting a model on the training set. The primary difference is in the predictions. So, you initialize a new decision tree and fit it on X_train and the residuals, y2_train (as opposed to y_train) *So the residuals of each tree become the new target column for the next tree*.
```
y2_train = y_train - y_train_pred
```
5. Repeat steps 2-4: as the process continues, the residuals should gradually approach 0 from the positive and negative direction. The iteration continue for the number of estimators, ***n_estimators***.
6. Sum the results: Summing the results requires making predictions for each new tree with the test set as follows:
```
y1_pred = tree_1.predict(X_test)
y2_pred = tree_2.predict(X_test)
y3_pred = tree_3.predict(X_test)
```
Since the predictions are positive and negative differences, summing the predictions should result in predictions that are closer to the target column:
```
y_pred = y1_pred + y2_pred + y3_pred
```
7. Compute the mean squared error (MSE) to obtain the results.
MSE(y_test, y_pred).

Gradient boosting hyperparameters

The most important are ***learning_rate*** and ***n_estimators***.

Learning_rate: Also known as *shrinkage*, it shrinks the contribution of individual trees so that no tree has too much influence when building the model. If an entire ensemble is built from the errors of one base learner, without careful adjustment of hyperparameters, early trees in the model can have too much influence on subsequent development. Generally speaking, as n_estimators, the number of trees, goes up, learning_rate should go down. So, determining an optimal learning_rate requires varying n_estimators.

XGBoost Unveiled

Extreme Gradient Boosting. XGBoost is a significant upgrade from gradient boosting. It includes built-in regularization and gains in speed. The *extreme* means pushing computational limits to the extreme. This requires knowledge not just of model-building, but also of disk-reading, compression, cache, and cores.

Handling missing values*:* XGBoost is capable of handling missing values for you. There is a 'missing' hyperparameter that can be set to any value. When given a missing data point, XGBoost scores different split options and chooses the one with the best results.

Gaining speed*:* XGBoost was specifically designed for speed. Speed gains allow ML models to build more quickly, which is important when dealing with millions of rows of data. Design features which give XGBoost an edge in speed over other ensemble algorithms:

- *Approximate split-finding algorithm*: Decision trees need optimal splits to produce optimal results. A *greedy algorithm* selects the best split at each step and does not backtrack to look at previous branches (decision tree splitting is usually performed in this manner). XGBoost presents a greedy algorithm in addition to a new approximate split-finding algorithm. This algorithm uses ***quantiles***, percentages that split data, to propose candidate splits. Globally, the same quantiles are used through the entire training, and locally new quantiles are provided for each round of splitting.
- *Sparsity aware split-finding*: Sparse data occurs when the majority of entries are 0 or null. This can happen when the dataset has been one-hot encoded. Sparse matrices are designed to only store data points with non-zero and non-null values, which saves valuable space. So, XGBoost is faster(50x when compared to standard approach) when looking for splits because its matrices are sparse.
- *Parallel computing*: Boosting is not ideal for parallel computing since each tree depends on the results of the previous tree. Parallel computing occurs when multiple computational units are working together on the same problem at the same time. XGBoost sorts and compresses the data into blocks. These blocks may be distributed to multiple machines, or to external memory. Sorting the data is faster with blocks, and the split-finding algorithm takes advantage of blocks, while the search for quantiles is faster due to blocks.
- *Cache-aware access*: The data on your computer is separated into *cache* and *main memory*. The cache, what you use most often, is reserved for high-speed memory. The data you use less often is held back for lower-speed memory. When it comes to gradient statistics, XGBoost uses ***cache-aware prefetching***. It allocates an internal buffer, fetches the gradient statistics, and performs accumulation with mini batches. Prefetching lengthens read/write dependency and reduces run-times by approximately 50% for large datasets.
- *Block compression and sharding*: Block compression helps with computationally expensive disk reading by compressing columns. Block sharding decreases read times by sharding the data into multiple disks that alternate when reading the data.

Accuracy gains

XGBoost adds in built-in regularization to achieve accuracy gains beyond gradient boosting. ***Regularization*** is the process of adding information to reduce variance and prevent overfitting. Data can be regularizing through hyperparameter tuning, though regularized algorithms can also be attempted (Ridge and Lasso are regularized alternatives to linear regression).

XGBoost includes regularization as part of the learning objective, as contrasted with gradient boosting and random forests. The regularized parameters penalize complexity and smooth

out the final weights to prevent overfitting. XGBoost is a regularized version of gradient boosting.

Analyzing XGBoost parameters

Learning objective: This determines how well the model fits the data. In the case of XGBoost, the learning objective consists of two parts: the ***loss function*** and the ***regularization term***. Mathematically, this is defined as follows:

$$obj(\theta) = l(\theta) + \Omega(\theta)$$

Here, l(θ) is the loss function, which is the ***mean squared error (MSE)*** for regression, or the log loss for classification, and Ω(θ) is the regularization function, a penalty term to prevent overfitting. It is this regularization term as part of the objective function that distinguishes XGBoost from most tree ensembles.

Loss function

Defined as the MSE for regression, the loss function can be written as follows:

$$l(\theta) = \sum_{i=1}^{n} (y_i - yhat_i)^2$$

Here, y_i is the target value for the i^{th} row and $yhat_i$ is the value predicted by the ML model for the i^{th} row. Σ indicates that all rows are summed starting with i = 1 and ending with i = n, the number of rows.

The prediction, $yhat_i$ for a given tree requires a function that starts at the tree root and ends at a leaf. This can be expressed as follows:

$$yhat_i = f(x_i), f \in F$$

Here, x_i is a vector whose entries are the columns of the i_{th} row and f ϵ F means that the function *f* is a member of F, the set of all possible CART (Classification and Regression Trees) functions. In gradient boosting, the function that determines the prediction for the i^{th} row includes the sum of all previous functions:

$$yhat_i = \sum_{t=1}^{T} f_t(x_i| \quad), f_t \in F$$

T is the number of boosted trees. So, to obtain the prediction of the i^{th} tree, sum the predictions of the previous trees in addition to the prediction of the new tree.

Regularization function

Let *w* be the vector space of leaves. Then, *f*, the function mapping the tree root to the leaves, can be recast in terms of *w*, as follows:

$$f_t(x) = w_{q(x)}, w \in R^T, q: R^d \rightarrow \{1,2,\dots,T\}$$

Here, *q* is the function assigning data points to leaves and T is the number of leaves. XGBoost settled on the following regularization function where ϒ and λ are penalty constants to reduce overfitting:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

Skipping over lots of the derivation, the result of what XGBoost uses to determine how well the model fits the data, the objective function:

$$obj^{(t)} = \frac{-1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T$$

XGBoost Hyperparameters

XGBoost base learner hyperparameters incorporate all decision tree hyperparameters as a starting point. There are also gradient boosting hyperparameters, since XGBoost is an enhanced version of gradient boosting. Hyperparameters unique to XGBoost are designed to improve upon accuracy and speed.

N_estimators: The number of trees in the ensemble; i.e., the number of trees trained on the residuals.

Learning_rate: This shrinks the weight of trees for each round of boosting. By lowering *learning_rate,* more trees are required to produce better scores. This prevents overfitting because the size of the weights carried forward is smaller.

Gamma: This provides a threshold that nodes must surpass before making further splits according to the loss function. There is no upper limit, though anything over 10 is considered very high (the default is 0). Increasing gamma results in a more conservative model.

Min_child_weight: The minimum sum of weights required for a node to split into a child. If the sum of the weights is less than the value of min_child_weight, no further splits are made. This again reduces overfitting.

Subsample: This limits the percentage of training instances for each boosting round. Decreasing this from 100% reduces overfitting.

Colsample_bytree: Randomly selects particular columns according to the given percentage. It is useful for limiting the influence of columns and reducing variance. It takes a percentage as input, not the number of columns.

Early Stopping

A method to limit the number of training rounds in iterative ML algorithms. Instead of predefining the number of training rounds, early stopping allows training to continue until *n* consecutive rounds fail to produce any gains. ***Early_stopping_rounds*** is the key parameter here. It is not a hyperparameter, but a strategy for optimizing the n_estimators hyperparameter.

Normally when choosing hyperparameters, a test score is given after all boosting rounds are complete. To use early stopping, we need a test score after each round. ***Eval_metric*** and ***eval_set*** may be used as parameters for '.fit' to generate test scores for each training round. The former provides the scoring method (typically 'error' for classification and 'rmse' for regression), while the latter provides the test to be evaluated (X_test and y_test).

XGBoost Alternative Base Learners

Base learners are the individual models, most commonly trees, that are iterated upon for each boosting round. Along with the default decision tree (***gbtree***), additional options include ***gblinear*** and ***dart***.

Decision trees are the preferred base learner on account that boosted trees consistently produce excellent results. ***Gblinear*** is a gradient boosted linear model, and ***dart*** is a variation of decision trees that includes a dropout technique based on neural networks. There are also XGBoost random forests.

Gblinear: Decision trees are optimal for non-linear data as they can easily access points by splitting the data as many times as needed. They are often preferable as base learners because real data is usually non-linear.

There may be cases where a linear model is ideal. If the real data has a linear relationship, a decision tree is probably not the best option; *gblinear* may be better. The general idea behind boosted linear models is the same as boosted tree models. A base model is built, and each subsequent model is trained upon the residuals. At the end, the individual models are summed for the final result. The primary distinction in this case is that each model in the ensemble is linear.

Gblinear also adds regularization terms to linear regression. It can also be used for classification problems via logistic regression. This works because logistic regression is also built by finding optimal coefficients (weighted inputs), as in linear regression, and summed via the sigmoid equation.

Dart: Dropouts meet Multiple Additive Regression Trees. The dropout technique eliminates nodes (mathematical points) from each layer of learning in a neural network, thereby reducing overfitting. So, the dropout technique slows down the learning process by eliminating information from each round.

In each new round of boosting, instead of summing the residuals from all previous trees to build a new model, DART selects a random sample of previous trees and normalizes the leaves by a scaling factor of 1/k, where k is the number of trees dropped.

XGBoost Kaggle Masters

Implementing XGBoost isn't enough, nor is fine-tuning its hyperparameters. It is equally important to engineer new data and to combine optimal models to attain higher scores.

Furthermore, the purpose of building ML models is to make accurate predictions using unknown data. The distinction between validating models on test sets and testing models on hold-out sets is important. A general approach for validating and testing ML models:

1. Split data into a training set and a hold-out set. Keep the hold-out set away.
2. Split the training set into a training and test set, or use cross-validation. Fit new models on the training set and validate the model, going back and forth to improve scores.
3. After obtaining a final model, test it on the hold-out set: this is the real test of the model. If the score is below expectations, return to step 2 and repeat. Do not use the hold-out set as the new validation set, otherwise the model will adjust itself to match the hold-out set, which defeats the purpose of a hold-out set in the first place.

Engineering new columns

ML models are as good as the data that they train on. When data is insufficient, building a robust ML model is impossible. Can data be improved? Yes, by extracting new data from columns. Feature engineering is the process of developing new columns of data from the original columns. The question is not whether you should feature engineer, but how much feature engineering you should implement. Look at:

- How you treat null or missing values (do you just drop, impute the mean, etc.?);
- How you treat timestamp/date columns (extract the month, is it rush hour, etc.?);
- How you treat categorical columns (one hot encode, target/mean encoding, etc.?);
- How you treat frequency columns (turn into percentages?)

Building non-correlated ensembles

Consider ensembles of distinct models, including a range of diverse and tuned models.

Correlation is a statistical measure between -1 and 1 that indicates the strength of the linear relationship between two sets of points. A correlation of 1 is a perfectly straight line, while a correlation of 0 shows no linear relationship whatsoever. A high correlation between ML models is undesirable in an ensemble. It is desirable to have a diversity of models that score well but give different predictions. If most models give the same prediction, the correlation is high and there is little value in adding the new model. Finding differences in predictions where a strong model may be wrong gives the ensemble the chance to produce better results.

To compute correlations between ML models, we need data points to compare. The different data points that ML models produce are their predictions. After obtaining predictions, we concatenate them into a dataframe, and then apply the '.corr' method to obtain all correlations at once.

4. Concatenate the predictions into a new DataFrame using **`np.c`** (the **`c`** is short for concatenation):

```
df_pred = pd.DataFrame(data=
np.c_[y_pred_gbtree, y_pred_dart,
y_pred_forest, y_pred_logistic,
y_pred_xgb], columns=['gbtree',
'dart','forest', 'logistic', 'xgb'])
```

5. Run correlations on the DataFrame using the **`.corr()`** method:

```
df_pred.corr()
```

You should see the following output:

	gbtree	dart	forest	logistic	xgb
gbtree	1.000000	0.971146	0.884584	0.914111	0.971146
dart	0.971146	1.000000	0.913438	0.914111	0.971146
forest	0.884584	0.913438	1.000000	0.943308	0.913438
logistic	0.914111	0.914111	0.943308	1.000000	0.914111
xgb	0.971146	0.971146	0.913438	0.914111	1.000000

There is no clear cut-off to obtain a non-correlated threshold. It ultimately depends on the values of correlation and the number of models to choose from. For this example, we could pick the next two least correlated models with our best model, xgb – random forest and logistic regression.

We can then ***stack*** the models. Stacking combines ML models at two different levels: the base level, whose models make predictions on all the data, and the meta level, which takes the predictions of the base levels as input and uses them to generate final predictions. In other words, the final model does not take the original data as input, but rather takes the predictions of the base ML models as input. This is different from a standard ensemble.