Machine Learning

Sources Used
https://xgboost.readthedocs.io/en/stable/
https://en.wikipedia.org/wiki/XGBoost
https://www.nvidia.com/en-us/glossary/xgboost/
https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/
https://www.ibm.com/topics/random-forest#:~:text=Random%20forest%20is%20a%20commonly,both%20classification%20and%20regression%20problems.
https://scijinks.gov/forecast-reliability/#:~:text=The%20Short%20Answer%3A,right%20about%20half%20the%20time.
https://www.datacamp.com/tutorial/xgboost-in-python

XGBoost is an optimized distributed gradient boosting library designed to be highly **efficient**, **flexible** and **portable**. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

XGBoost[2] (eXtreme Gradient Boosting) is an open-source software library which provides a regularizing gradient boosting framework for C++, Java, Python,[3] R,[4] Julia,[5] Perl,[6] and Scala. It works on Linux, Microsoft Windows,[7] and macOS.[8] From the project description, it aims to provide a "Scalable, Portable and Distributed Gradient Boosting (GBM, GBRT, GBDT) Library". It runs on a single machine, as well as the distributed processing frameworks Apache Hadoop, Apache Spark, Apache Flink, and Dask.[9][10]

XGBoost gained much popularity and attention in the mid-2010s as the algorithm of choice for many winning teams of machine learning competitions.[11]

History[edit]

XGBoost initially started as a research project by Tianqi Chen[12] as part of the Distributed (Deep) Machine Learning Community (DMLC) group. Initially, it began as a terminal application which could be configured using a libsvm configuration file. It became well known in the ML competition circles after its use in the winning solution of the Higgs Machine Learning Challenge. Soon after, the Python and R packages were built, and XGBoost now has package implementations for Java, Scala, Julia, Perl, and other languages. This brought the library to more developers and contributed to its popularity among the Kaggle community, where it has been used for a large number of competitions.[11]

It was soon integrated with a number of other packages making it easier to use in their respective communities. It has now been integrated with scikit-learn for Python users and with the caret package for R users. It can also be integrated into Data Flow frameworks like Apache Spark, Apache Hadoop, and Apache Flink using the abstracted Rabit[13] and XGBoost4J.[14] XGBoost is also available on OpenCL for FPGAs.[15] An efficient, scalable implementation of XGBoost has been published by Tianqi Chen and Carlos Guestrin.[16]

While the XGBoost model often achieves higher accuracy than a single decision tree, it sacrifices the intrinsic interpretability of decision trees. For example, following the path that a decision tree takes to make its decision is trivial and self-explained, but following the paths of hundreds or thousands of trees is much harder.

XGBoost

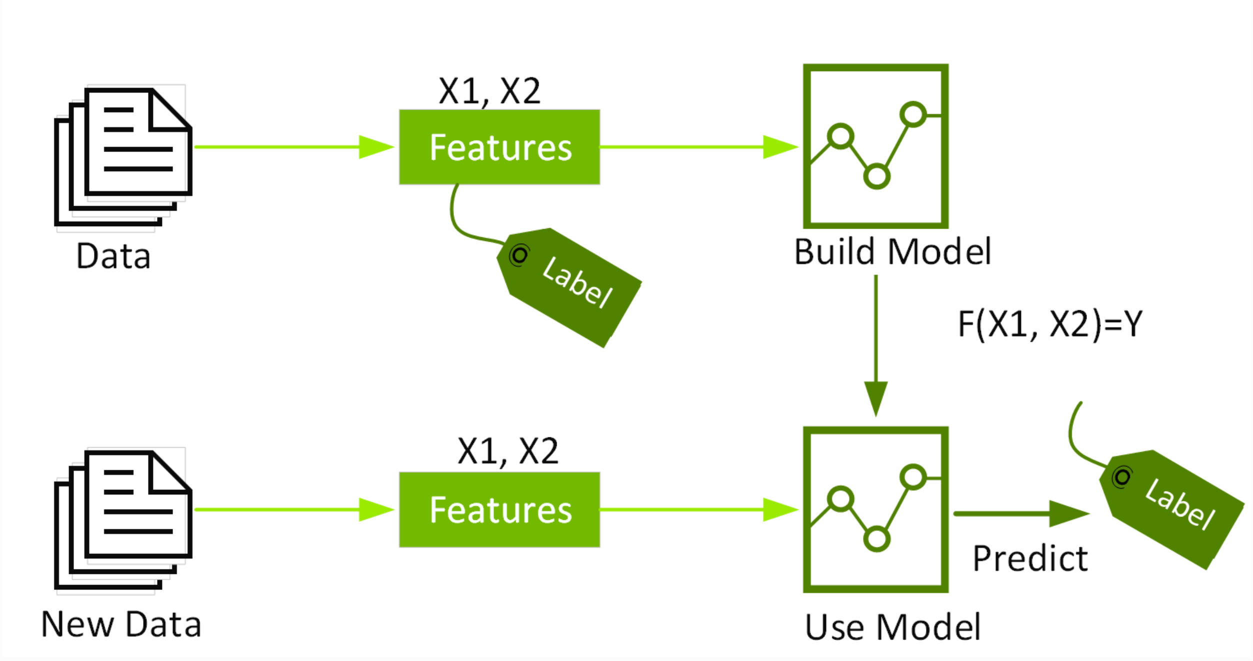
XGBoost is an open-source software library that implements optimized distributed gradient boosting machine learning algorithms under the Gradient Boosting framework.

What is XGBoost?

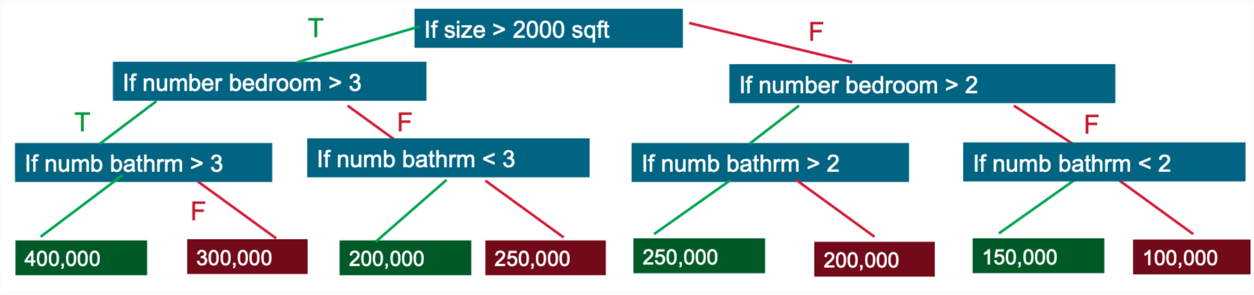
XGBoost, which stands for Extreme Gradient Boosting, is a scalable, distributed gradient-boosted decision tree (GBDT) machine learning library. It provides parallel tree boosting and is the leading machine learning library for regression, classification, and ranking problems.

It's vital to an understanding of XGBoost to first grasp the machine learning concepts and algorithms that XGBoost builds upon: supervised machine learning, decision trees, ensemble learning, and gradient boosting.

Supervised machine learning uses algorithms to train a model to find patterns in a dataset with labels and features and then uses the trained model to predict the labels on a new dataset's features.

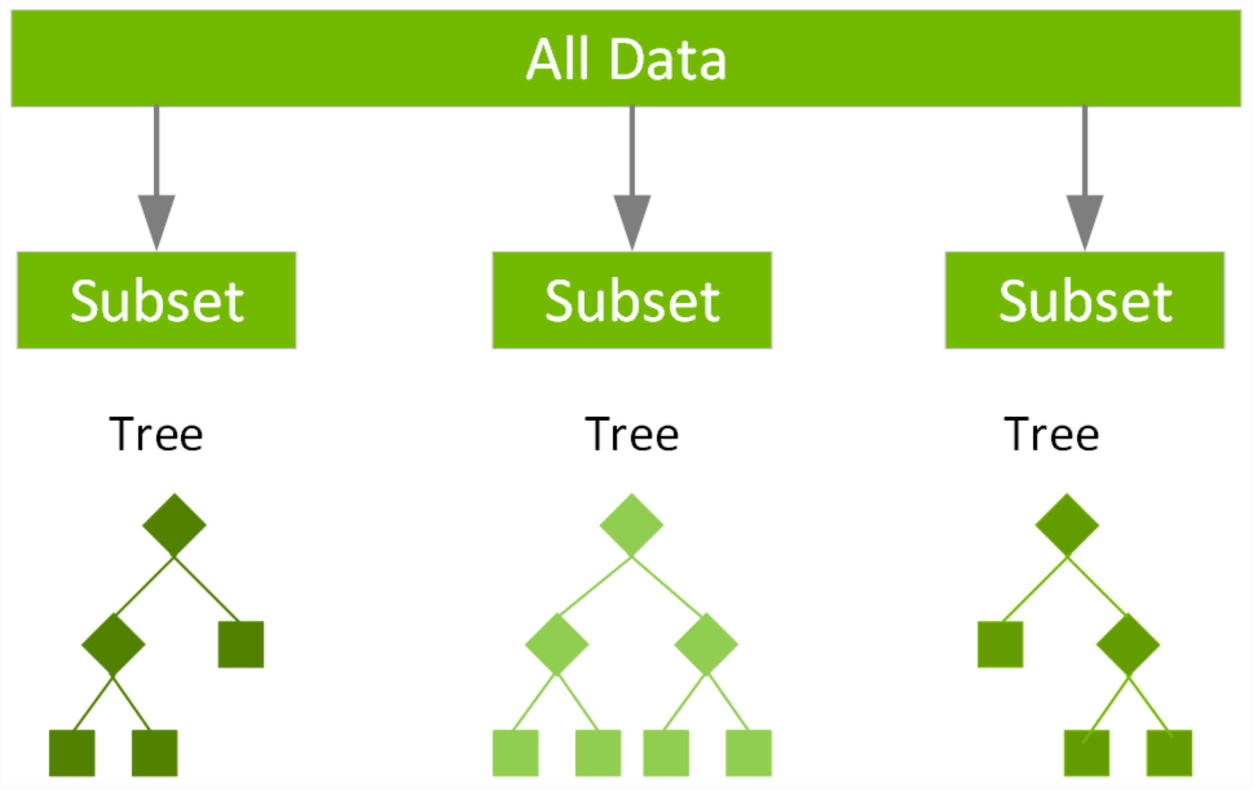


Decision trees create a model that predicts the label by evaluating a tree of if-then-else true/false feature questions, and estimating the minimum number of questions needed to assess the probability of making a correct decision. Decision trees can be used for classification to predict a category, or regression to predict a continuous numeric value. In the simple example below, a decision tree is used to estimate a house price (the label) based on the size and number of bedrooms (the features).



A Gradient Boosting Decision Trees (GBDT) is a decision tree ensemble learning algorithm similar to random forest, for classification and regression. Ensemble learning algorithms combine multiple machine learning algorithms to obtain a better model.

Both random forest and GBDT build a model consisting of multiple decision trees. The difference is in how the trees are built and combined.



Random forest uses a technique called bagging to build full decision trees in parallel from random bootstrap samples of the data set. The final prediction is an average of all of the decision tree predictions.

The term “gradient boosting” comes from the idea of “boosting” or improving a single weak model by combining it with a number of other weak models in order to generate a collectively strong model. Gradient boosting is an extension of boosting where the process of additively generating weak models is formalized as a gradient descent algorithm over an objective function. Gradient boosting sets targeted outcomes for the next model in an effort to minimize errors. Targeted outcomes for each case are based on the gradient of the error (hence the name gradient boosting) with respect to the prediction.

GBDTs iteratively train an ensemble of shallow decision trees, with each iteration using the error residuals of the previous model to fit the next model. The final prediction is a weighted sum of all of the tree predictions. Random forest “bagging” minimizes the variance and overfitting, while GBDT “boosting” minimizes the bias and underfitting.

XGBoost is a scalable and highly accurate implementation of gradient boosting that pushes the limits of computing power for boosted tree algorithms, being built largely for energizing machine learning model performance and computational speed. With XGBoost, trees are built in parallel, instead of sequentially like GBDT. It follows a level-wise strategy, scanning across gradient values and using these partial sums to evaluate the quality of splits at every possible split in the training set.

Why XGBoost?

XGBoost gained significant favor in the last few years as a result of helping individuals and teams win virtually every Kaggle structured data competition. In these competitions, companies and researchers post data after which statisticians and data miners compete to produce the best models for predicting and describing the data.

Initially both Python and R implementations of XGBoost were built. Owing to its popularity, today XGBoost has package implementations for Java, Scala, Julia, Perl, and other languages. These implementations have opened the XGBoost library to even more developers and improved its appeal throughout the Kaggle community.

XGBoost has been integrated with a wide variety of other tools and packages such as scikit-learn for Python enthusiasts and caret for R users. In addition, XGBoost is integrated with distributed processing frameworks like Apache Spark and Dask.

In 2019 XGBoost was named among InfoWorld's coveted Technology of the Year award winners.

XGBoost Benefits and Attributes

The list of benefits and attributes of XGBoost is extensive, and includes the following:

- A large and growing list of data scientists globally that are actively contributing to XGBoost open source development
- Usage on a wide range of applications, including solving problems in regression, classification, ranking, and user-defined prediction challenges
- A library that's highly portable and currently runs on OS X, Windows, and Linux platforms
- Cloud integration that supports AWS, Azure, Yarn clusters, and other ecosystems
- Active production use in multiple organizations across various vertical market areas
- A library that was built from the ground up to be efficient, flexible, and portable

XGBoost is a machine learning algorithm that belongs to the ensemble learning category, specifically the gradient boosting framework. It utilizes decision trees as base learners and employs regularization techniques to enhance model generalization. XGBoost is famous for its computational efficiency, offering efficient processing, insightful feature importance analysis, and seamless handling of missing values. It's the go-to algorithm for a wide range of tasks, including regression, classification, and ranking. In this article, we will give you an overview of XGBoost, along with a use-case!

We recommend going through the below article as well to fully understand the various terms and concepts mentioned in this article:

- A Comprehensive Guide to Ensemble Modeling (with Python codes)
- Ensemble Learning Course: Ensemble Learning and Ensemble Learning Techniques

Table of contents

What is XGBoost in Machine Learning?

XGBoost, or eXtreme Gradient Boosting, is a machine learning algorithm under ensemble learning. It is trendy for supervised learning tasks, such as regression and classification. XGBoost builds a predictive model by combining the predictions of multiple individual models, often decision trees, in an iterative manner.

The algorithm works by sequentially adding weak learners to the ensemble, with each new learner focusing on correcting the errors made by the existing ones. It uses a gradient descent optimization technique to minimize a predefined loss function during training.

Key features of XGBoost Algorithm include its ability to handle complex relationships in data, regularization techniques to prevent overfitting and incorporation of parallel processing for efficient computation.

Why Ensemble Learning?

XGBoost is an ensemble learning method. Sometimes, it may not be sufficient to rely upon the results of just one machine learning model. Ensemble learning offers a systematic solution to combine the predictive power of multiple learners. The resultant is a single model which gives the aggregated output from several models.

The models that form the ensemble, also known as base learners, could be either from the same learning algorithm or different learning algorithms. Bagging and boosting are two widely used ensemble learners. Though these two techniques can be used with several statistical models, the most predominant usage has been with decision trees.

Let's briefly discuss bagging before taking a more detailed look at the concept of gradient boosting.

Bagging

While decision trees are one of the most easily interpretable models, they exhibit highly variable behavior. Consider a single training dataset that we randomly split into two parts. Now, let's use each part to train a decision tree in order to obtain two models.

When we fit both these models, they would yield different results. Decision trees are said to be associated with high variance due to this behavior. Bagging or boosting aggregation helps to reduce the variance in any learner. Several decision trees which are generated in parallel, form the base learners of bagging technique. Data sampled with replacement is fed to these learners for training. The final prediction is the averaged output from all the learners.

Boosting

In boosting, the trees are built sequentially such that each subsequent tree aims to reduce the errors of the previous tree. Each tree learns from its predecessors and updates the residual errors. Hence, the tree that grows next in the sequence will learn from an updated version of the residuals.

The base learners in boosting are weak learners in which the bias is high, and the predictive power is just a tad better than random guessing. Each of these weak learners contributes some vital information for prediction, enabling the boosting technique to produce a strong learner by effectively combining these weak learners. The final strong learner brings down both the bias and the variance.

In contrast to bagging techniques like Random Forest, in which trees are grown to their maximum extent, boosting makes use of trees with fewer splits. Such small trees, which are not very deep, are highly interpretable. Parameters like the number of trees or iterations, the rate at which the gradient boosting learns, and the depth of the tree, could be optimally selected through validation techniques like k-fold cross validation. Having a large number of trees might lead to overfitting. So, it is necessary to carefully choose the stopping criteria for boosting.

Gradient Boosting Ensemble Technique

The gradient boosting ensemble technique consists of three simple steps:

- An initial model F0 is defined to predict the target variable y. This model will be associated with a residual (y – F0)
- A new model h1 is fit to the residuals from the previous step
- Now, F0 and h1 are combined to give F1, the boosted version of F0. The mean squared error from F1 will be lower than that from F0:

$$F_1(x) <- F_0(x) + h_1(x)$$

To improve the performance of F1, we could model after the residuals of F1 and create a new model F2:

$$F_2(x) <- F_1(x) + h_2(x)$$

This can be done for *'m'* iterations, until residuals have been minimized as much as possible:

$$F_m(x) <- F_{m-1}(x) + h_m(x)$$

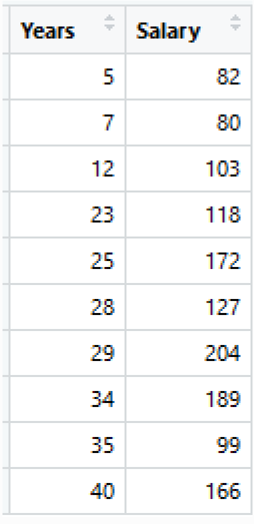
Here, the additive learners do not disturb the functions created in the previous steps. Instead, they impart information of their own to bring down the errors.

Demonstrating the Potential of Gradient Boosting

In this section, we will explore the power of gradient boosting, a machine learning technique, by building an ensemble model to predict salary based on years of experience. By utilizing regression trees and optimizing loss functions, we aim to showcase the significant reduction in error that gradient boosting can achieve.

Introduction to the Predictive Model

Consider the following data where the years of experience is predictor variable and salary (in thousand dollars) is the target. Using regression trees as base learners, we can create an ensemble model to predict the salary. For the sake of simplicity, we can choose square loss as our loss function and our objective would be to minimize the square error.



Years	Salary
5	82
7	80
12	103
23	118
25	172
28	127
29	204
34	189
35	99
40	166

Initializing the Model and Understanding Residuals

As the first step, the model should be initialized with a function F0(x). F0(x) should be a function which minimizes the loss function or MSE (mean squared error), in this case:

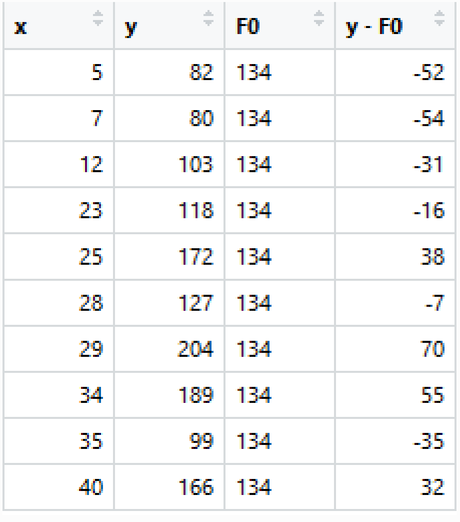
$$F_0(x) = argmin_\gamma \sum_{i=1}^{n} L(y_{i,} \ \gamma)$$

$$argmin_\gamma \sum_{i=1}^{n} L(y_{i,} \ \gamma) \ = \ argmin_\gamma \sum_{i=1}^{n} (y_i - \gamma)^2$$

Taking the first differential of the above equation with respect to γ, it is seen that the function minimizes at the mean i=1nyin. So, the boosting model could be initiated with:

$$F_0(x) = \frac{\sum_{i=1}^{n} y_i}{n}$$

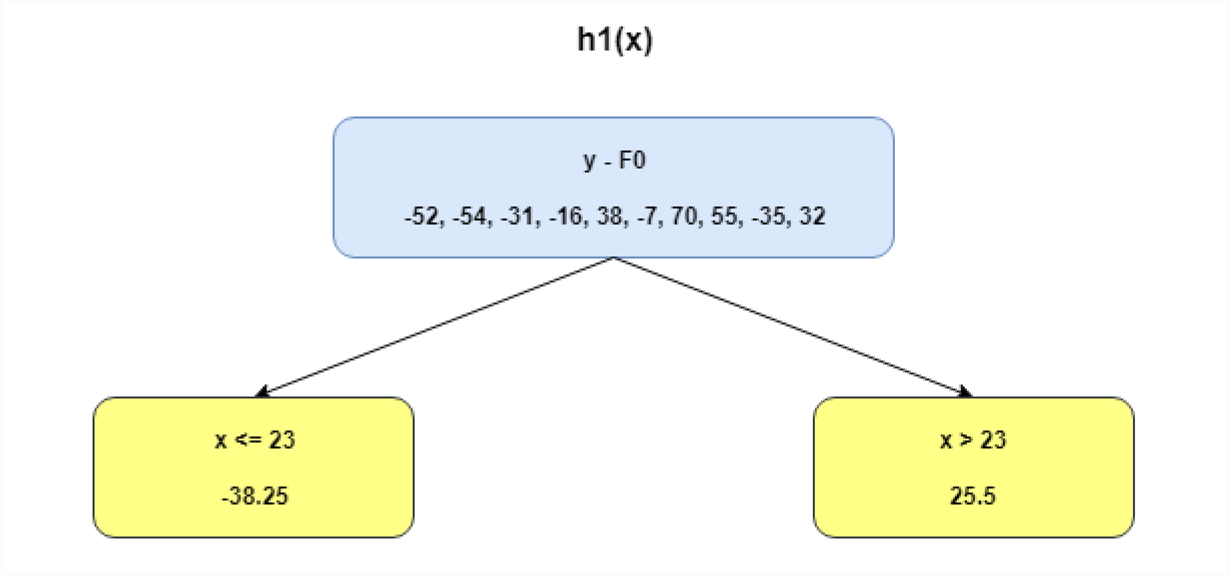
F0(x) gives the predictions from the first stage of our model. Now, the residual error for each instance is (yi – F0(x)).



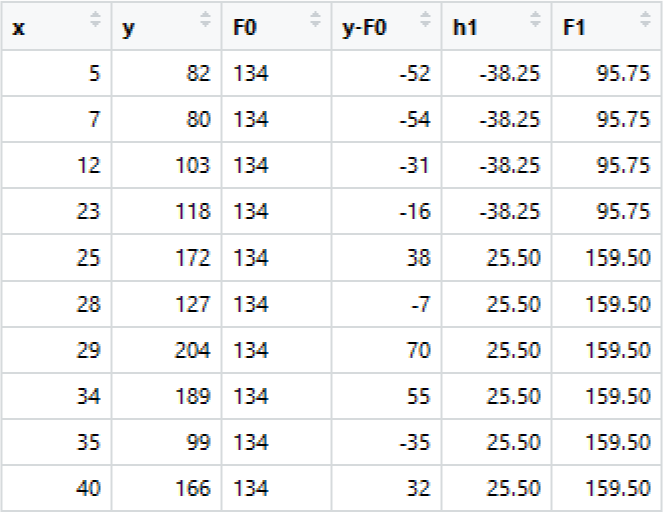
x	y	F0	y - F0
5	82	134	-52
7	80	134	-54
12	103	134	-31
23	118	134	-16
25	172	134	38
28	127	134	-7
29	204	134	70
34	189	134	55
35	99	134	-35
40	166	134	32

Building Additive Learners

We can use the residuals from F0(x) to create h1(x). h1(x) will be a regression tree which will try and reduce the residuals from the previous step. The output of h1(x) won’t be a prediction of y; instead, it will help in predicting the successive function F1(x) which will bring down the residuals.

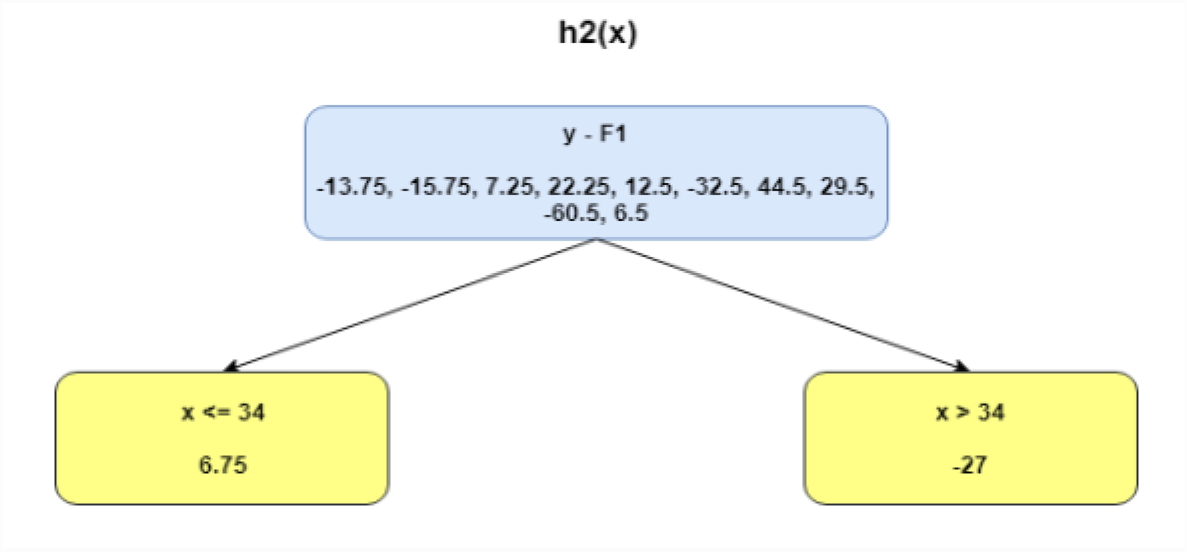


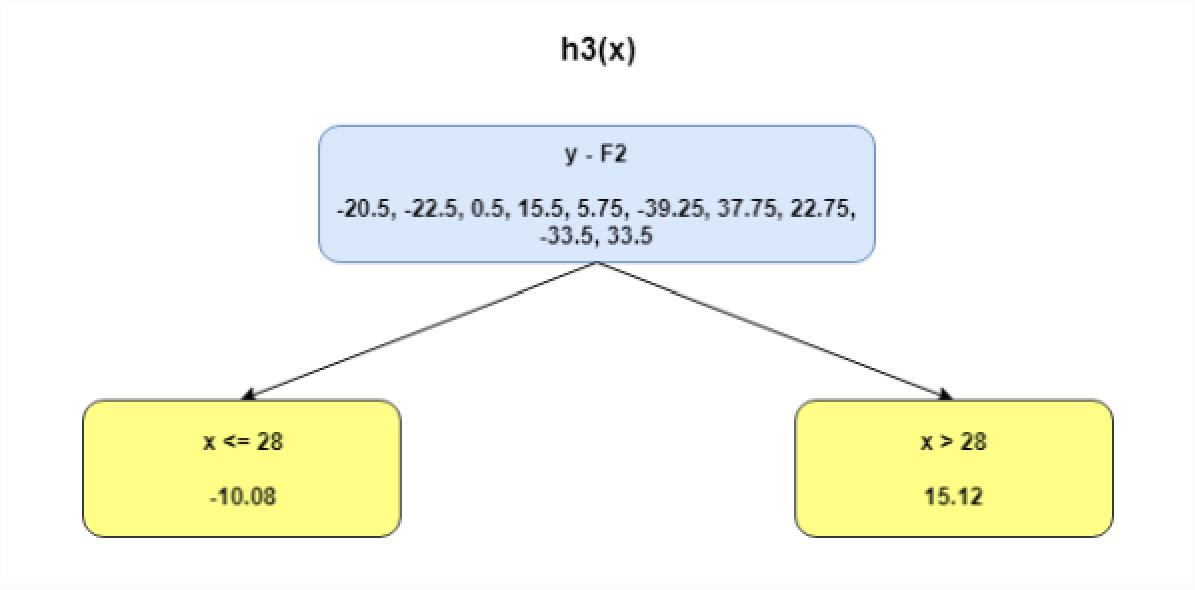
The additive model h1(x) computes the mean of the residuals (y – F0) at each leaf of the tree. The boosted function F1(x) is obtained by summing F0(x) and h1(x). This way h1(x) learns from the residuals of F0(x) and suppresses it in F1(x).

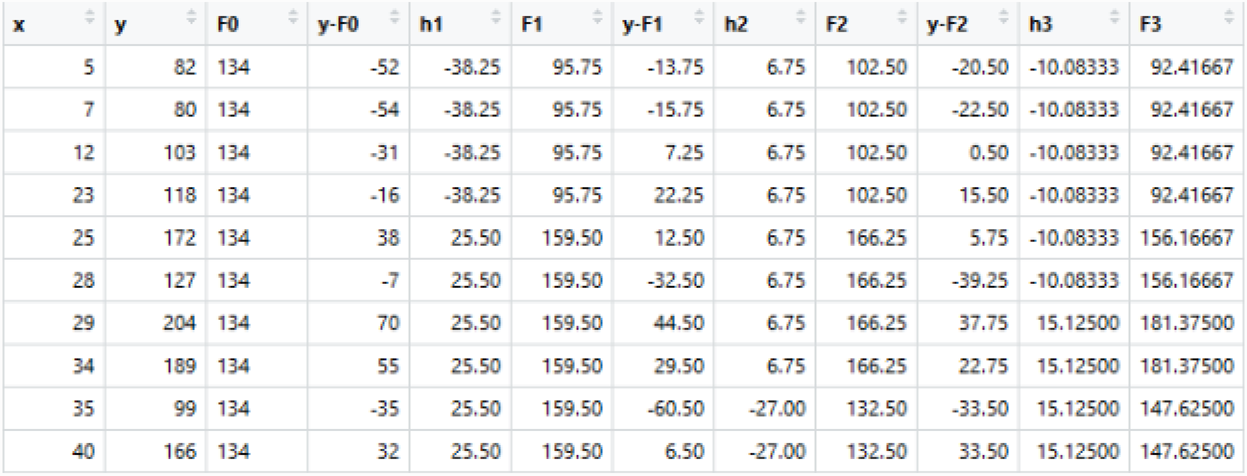


x	y	F0	y-F0	h1	F1
5	82	134	-52	-38.25	95.75
7	80	134	-54	-38.25	95.75
12	103	134	-31	-38.25	95.75
23	118	134	-16	-38.25	95.75
25	172	134	38	25.50	159.50
28	127	134	-7	25.50	159.50
29	204	134	70	25.50	159.50
34	189	134	55	25.50	159.50
35	99	134	-35	25.50	159.50
40	166	134	32	25.50	159.50

This can be repeated for 2 more iterations to compute h2(x) and h3(x). Each of these additive learners, hm(x), will make use of the residuals from the preceding function, Fm-1(x).





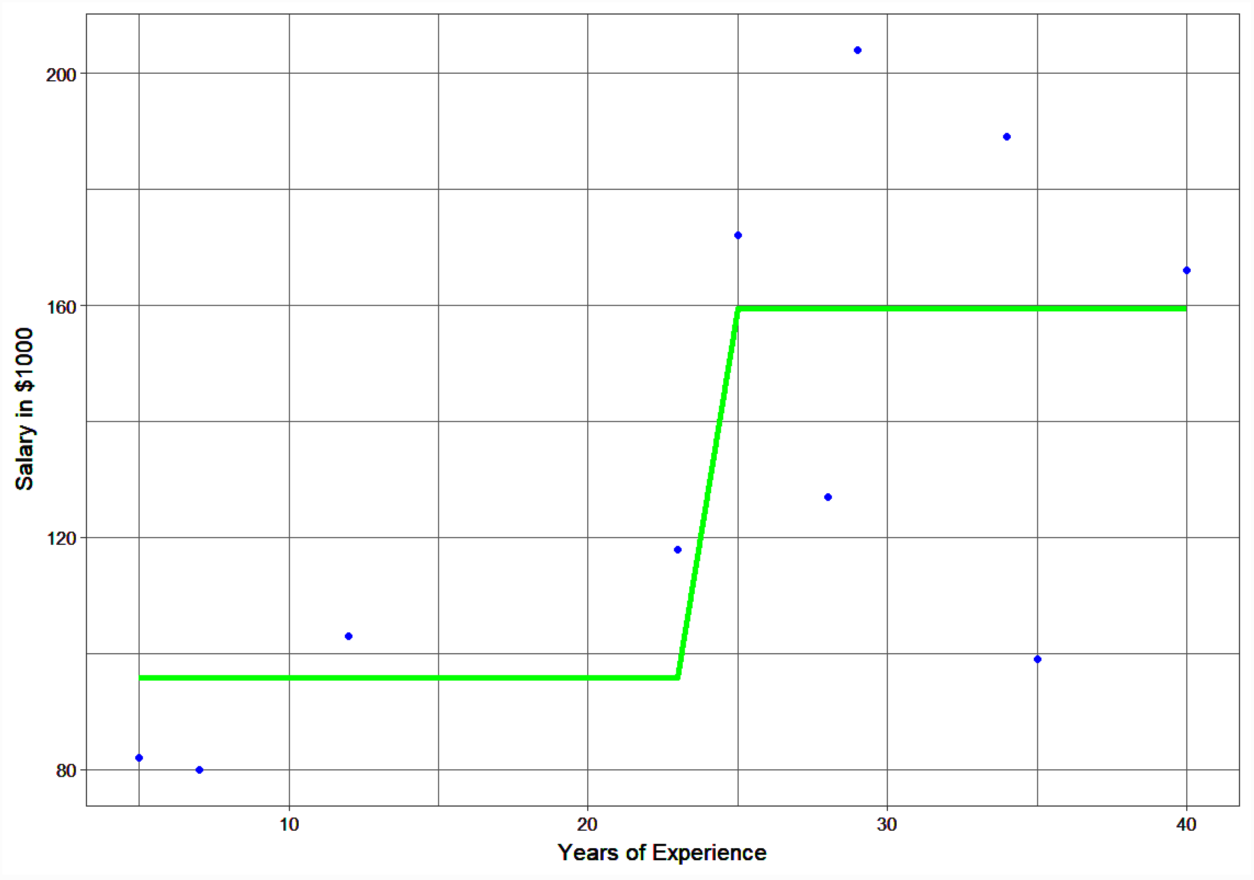


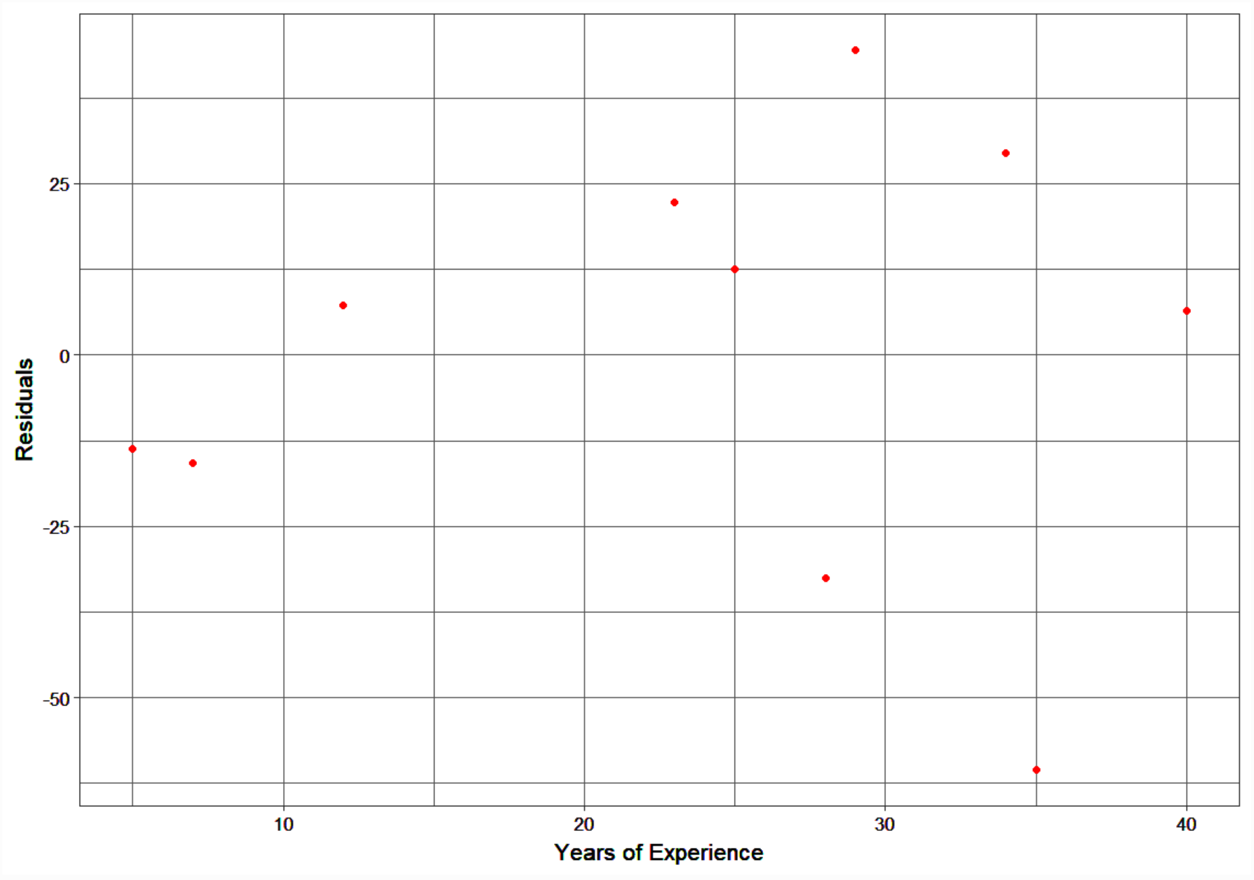
x	y	F0	y-F0	h1	F1	y-F1	h2	F2	y-F2	h3	F3
5	82	134	-52	-38.25	95.75	-13.75	6.75	102.50	-20.50	-10.08333	92.41667
7	80	134	-54	-38.25	95.75	-15.75	6.75	102.50	-22.50	-10.08333	92.41667
12	103	134	-31	-38.25	95.75	7.25	6.75	102.50	0.50	-10.08333	92.41667
23	118	134	-16	-38.25	95.75	22.25	6.75	102.50	15.50	-10.08333	92.41667
25	172	134	38	25.50	159.50	12.50	6.75	166.25	5.75	-10.08333	156.16667
28	127	134	-7	25.50	159.50	-32.50	6.75	166.25	-39.25	-10.08333	156.16667
29	204	134	70	25.50	159.50	44.50	6.75	166.25	37.75	15.12500	181.37500
34	189	134	55	25.50	159.50	29.50	6.75	166.25	22.75	15.12500	181.37500
35	99	134	-35	25.50	159.50	-60.50	-27.00	132.50	-33.50	15.12500	147.62500
40	166	134	32	25.50	159.50	6.50	-27.00	132.50	33.50	15.12500	147.62500

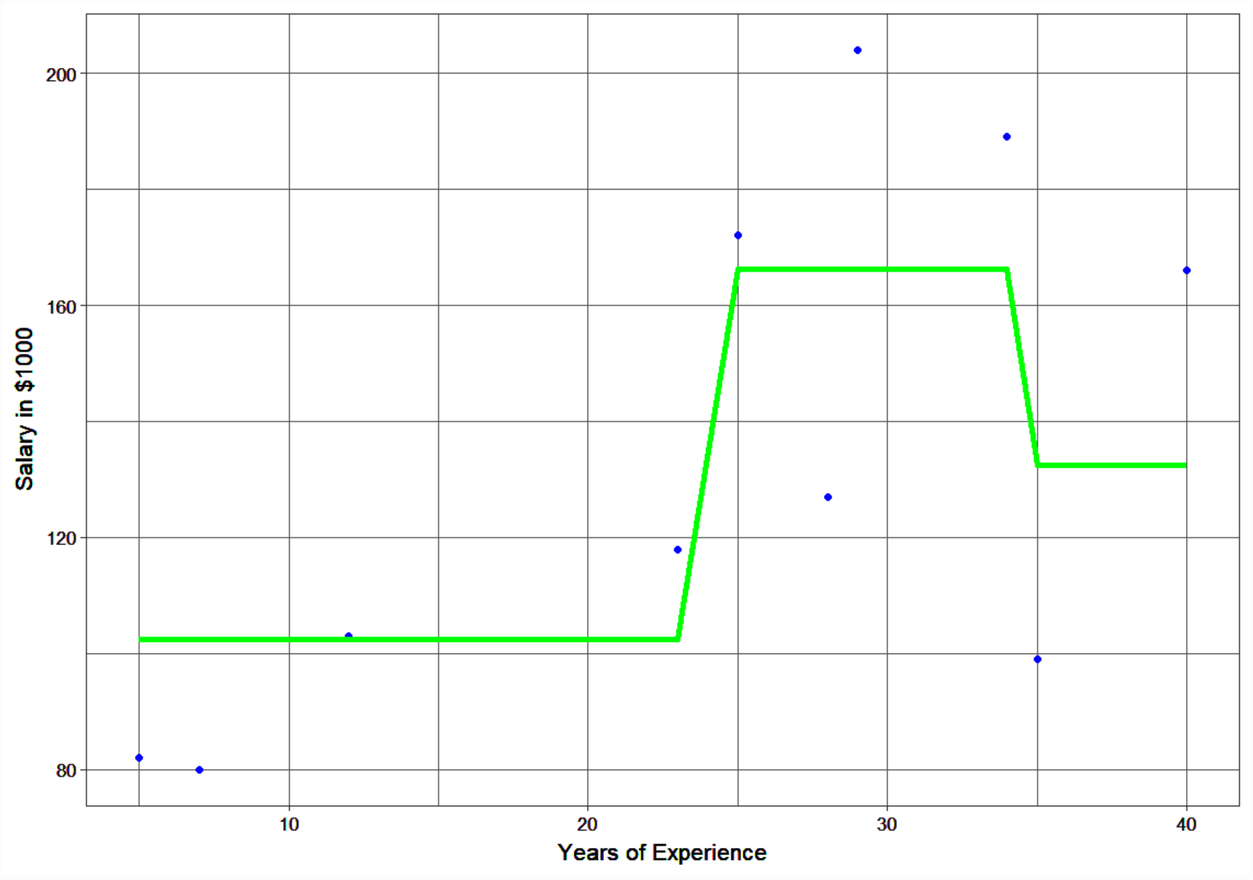
The MSEs for F0(x), F1(x) and F2(x) are 875, 692 and 540. It’s amazing how these simple weak learners can bring about a huge reduction in error!

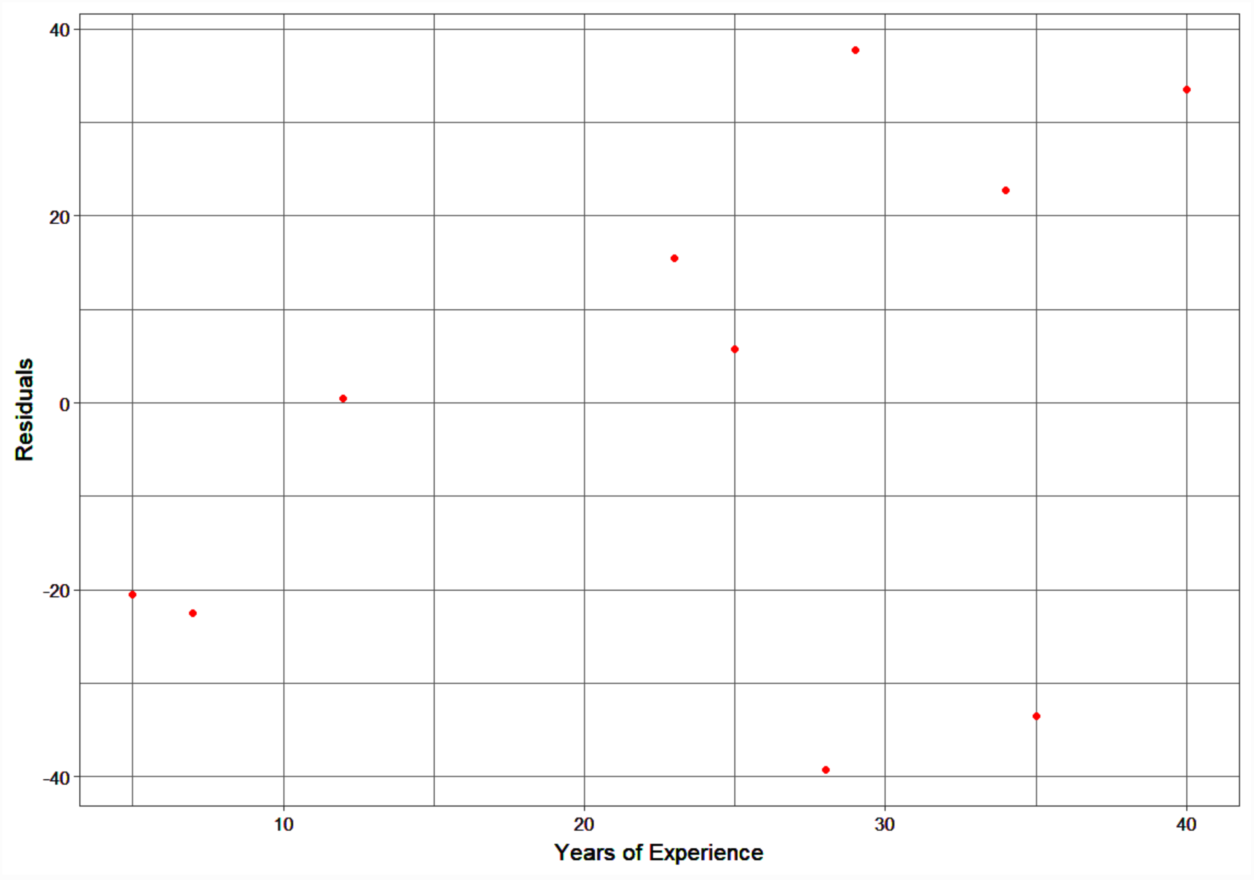
Observing the Reduction in Error

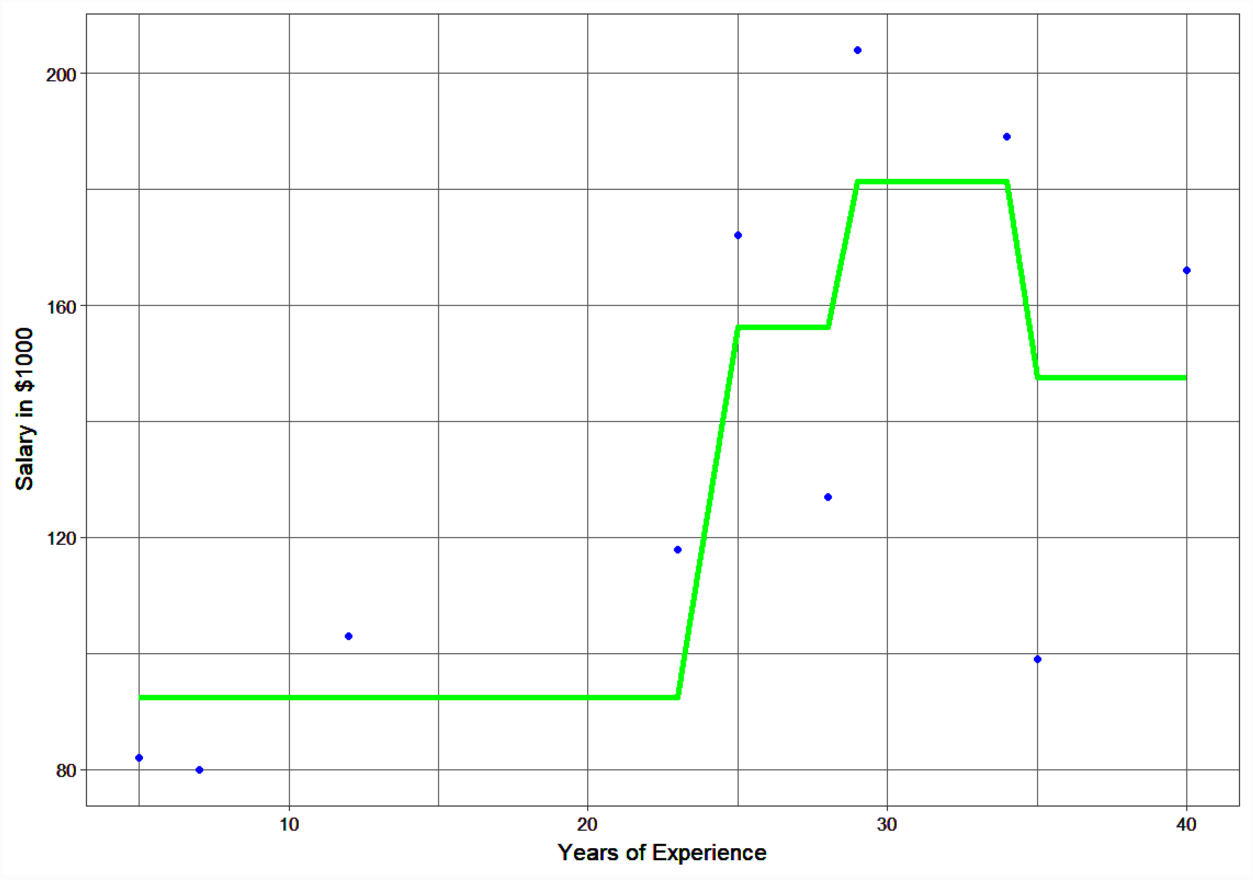
Note that each learner, hm(x), is trained on the residuals. All the additive learners in boosting are modeled after the residual errors at each step. Intuitively, it could be observed that the boosting learners make use of the patterns in residual errors. At the stage where maximum accuracy is reached by boosting, the residuals appear to be randomly distributed without any pattern.

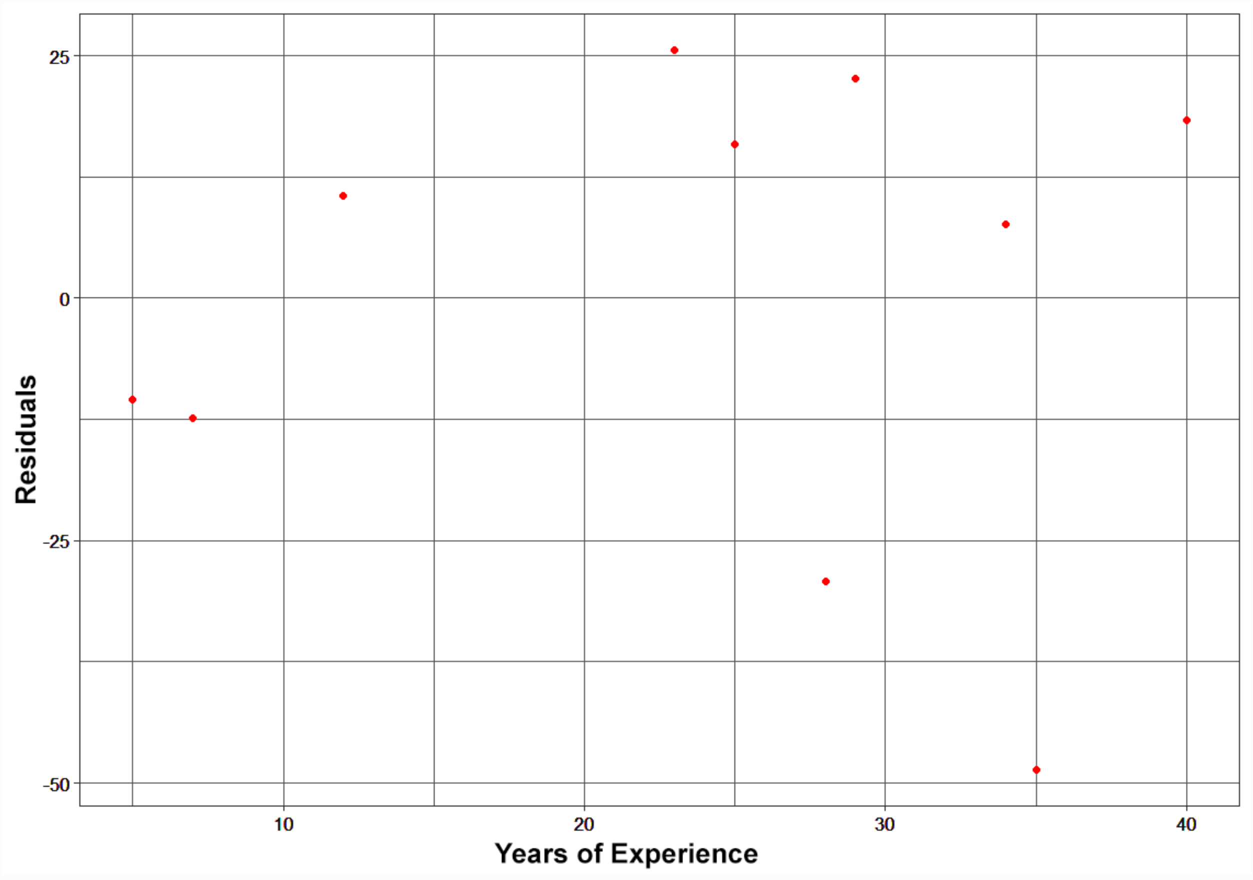

200
160
120
80
10
20
30
40
Salary in $1000
Years of Experience


25
0
-25
-50
Residuals
10
20
30
40
Years of Experience


Salary in $1000
200
160
120
80
10
20
30
40
Years of Experience


40
20
0
-20
-40
Residuals
10
20
30
40
Years of Experience


Salary in $1000
200
160
120
80
10
20
30
40
Years of Experience



Plots of Fn and hn

Using Gradient d=Descent for Optimizing the Loss Function

In the case discussed above, MSE was the loss function. The mean minimized the error here. When MAE (mean absolute error) is the loss function, the median would be used as $F_0(x)$ to initialize the model. A unit change in y would cause a unit change in MAE as well. Using scikit-learn, you can implement various models, including tree boosting algorithms and linear regression models to analyze the differences in loss functions and their impact on the model’s performance.

For MSE, the change observed would be roughly exponential. Instead of fitting hm(x) on the residuals, fitting it on the gradient of loss function, or the step along which loss occurs, would make this process generic and applicable across all loss functions.

Gradient descent helps us minimize any differentiable function. Earlier, the regression tree for hm(x) predicted the mean residual at each terminal node of the tree. In gradient boosting, the average gradient component would be computed.

For each node, there is a factor γ with which hm(x) is multiplied. This accounts for the difference in impact of each branch of the split. Gradient boosting helps in predicting the optimal gradient for the additive model, unlike classical gradient descent techniques which reduce error in the output at each iteration.

The following steps are involved in gradient boosting:

- F0(x) – with which we initialize the boosting algorithm – is to be defined:

$$F_0(x) = argmin_\gamma \sum_{i=1}^{n} L(y_i, \gamma)$$

- The gradient of the loss function is computed iteratively:

$$r_{im} = -\alpha\left[\frac{\partial(L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}, \textit{ where } \alpha \textit{ is the learning rate}$$

- Each hm(x) is fit on the gradient obtained at each step
- The multiplicative factor γm for each terminal node is derived and the boosted model Fm(x) is defined:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

Unique Features of XGBoost Model

XGBoost is a popular implementation of gradient boosting. Let’s discuss some features or metrics of XGBoost that make it so interesting:

- Regularization: XGBoost has an option to penalize complex models through both L1 and L2 regularization. Regularization helps in preventing overfitting
- Handling sparse data: Missing values or data processing steps like one-hot encoding make data sparse. XGBoost Classifier incorporates a

sparsity-aware split finding algorithm to handle different types of sparsity patterns in the data

- Weighted quantile sketch: Most existing tree based algorithms can find the split points when the data points are of equal weights (using quantile sketch algorithm). However, they are not equipped to handle weighted data. XGBoost has a distributed weighted quantile sketch algorithm to effectively handle weighted data
- Block structure for parallel learning: For faster computing, XGBoost Classifier can make use of multiple cores on the CPU. This is possible because of a block structure in its system design. Data is sorted and stored in in-memory units called blocks. Unlike other algorithms, this enables the data layout to be reused by subsequent iterations, instead of computing it again. This feature also serves useful for steps like split finding and column sub-sampling
- Cache awareness: In XGBoost, Scala non-continuous memory access is required to get the gradient statistics by row index. Hence,tianqi chen XGBoost has been designed to make optimal use of hardware. This is true and done by allocating internal buffers in each thread, where the gradient

statistics can be stored its Workflow. And these parallel tree make better XGboost algorithms with the help of julia and java lanuages.

- Out-of-core computing: This feature optimizes the available disk space and maximizes its usage when handling huge datasets that do not fit into memory

Python Code for XGBoost

Here’s a live coding window to see how XGBoost works and play around with the code without leaving this article!

XGBoost Model Benefits and Attributes

1. High accuracy: Xgboost Classifier is known for its accuracy and has been shown to outperform other machine learning algorithms in many predictive modeling tasks.
2. Scalability: It is highly scalable and can handle large datasets with millions of rows and columns.
3. Efficiency: It is designed to be computationally efficient and can quickly train models on large datasets.
4. Flexibility: It supports a variety of data types and objectives, including regression, classification, and ranking problems.

5. Regularization: It incorporates regularization techniques to avoid overfitting and improve generalization performance.
6. Interpretability: It provides feature importance scores that can help users understand which features are most important for making predictions.
7. Open-source: XGBoost Model is an open-source library that is widely used and supported by the data science community.

XGBoost vs Gradient Boosting

Feature	XGBoost	Gradient Boosting
Description	Advanced implementation of gradient boosting	Ensemble technique using weak learners
Optimization	Regularized objective function	Error gradient minimization
Efficiency	Highly optimized, efficient	Computationally intensive
Missing Values	Built-in support	Requires preprocessing
Regularization	Built-in L1 and L2	Requires external steps
Feature Importance	Built-in measures	Limited, needs external calculation
Interpretability	Complex, less interpretable	More interpretable models

Difference between XGBoost and Random Forest

Feature	XGBoost	Random Forest
Description	Improves mistakes from previous trees	Builds trees independently
Algorithm Type	Boosting	Bagging
Handling of Weak Learners	Corrects errors sequentially	Combines predictions of independently built trees
Regularization	Uses L1 and L2 regularization to prevent overfitting	Usually doesn't employ regularization techniques
Performance	Often performs better on structured data but needs more tuning	Simpler and less prone to overfitting

Conclusion

So that was all about the mathematics that power the popular XGBoost algorithm. If your basics are solid, this article must have been a breeze for you. It's such a powerful algorithm and while there are other techniques that have spawned from it (like CATBoost), XGBoost Model remains a game changer in the machine learning community. We highly recommend you to take up this course to sharpen your skills in machine learning and learn all the state-of-the-art techniques used in the field with our Applied Machine Learning – Beginner to

Professional course. Also, these Algorithm helps you for training data. and help

you for learning rate that will help you for lightgbm the algorithms.

What is random forest?
Random forest is a commonly-used machine learning algorithm, trademarked by Leo Breiman and Adele Cutler, that combines the output of multiple decision trees to reach a single result. Its ease of use and flexibility have fueled its adoption, as it handles both classification and regression problems.

Decision trees

Since the random forest model is made up of multiple decision trees, it would be helpful to start by describing the decision tree algorithm briefly. Decision trees start with a basic question, such as, “Should I surf?” From there, you can ask a series of questions to determine an answer, such as, “Is it a long period swell?” or “Is the wind blowing offshore?”. These questions make up the decision nodes in the tree, acting as a means to split the data. Each question helps an individual to arrive at a final decision, which would be denoted by the leaf node. Observations that fit the criteria will follow the “Yes” branch and those that don’t will follow the alternate path. Decision trees seek to find the best split to subset the data, and they are typically trained through the Classification and Regression Tree (CART) algorithm. Metrics, such as Gini impurity, information gain, or mean square error (MSE), can be used to evaluate the quality of the split.

This decision tree is an example of a classification problem, where the class labels are "surf" and "don't surf."

While decision trees are common supervised learning algorithms, they can be prone to problems, such as bias and overfitting. However, when multiple decision trees form an ensemble in the random forest algorithm, they predict more accurate results, particularly when the individual trees are uncorrelated with each other.

Ensemble methods

Ensemble learning methods are made up of a set of classifiers—e.g. decision trees—and their predictions are aggregated to identify the most popular result.

The most well-known ensemble methods are bagging, also known as bootstrap aggregation, and boosting. In 1996, Leo Breiman (link resides outside ibm.com) introduced the bagging method; in this method, a random sample of data in a training set is selected with replacement—meaning that the individual data points can be chosen more than once. After several data samples are generated, these models are then trained independently, and depending on the type of task—i.e. regression or classification—the average or majority of those predictions yield a more accurate estimate. This approach is commonly used to reduce variance within a noisy dataset.

Random forest algorithm

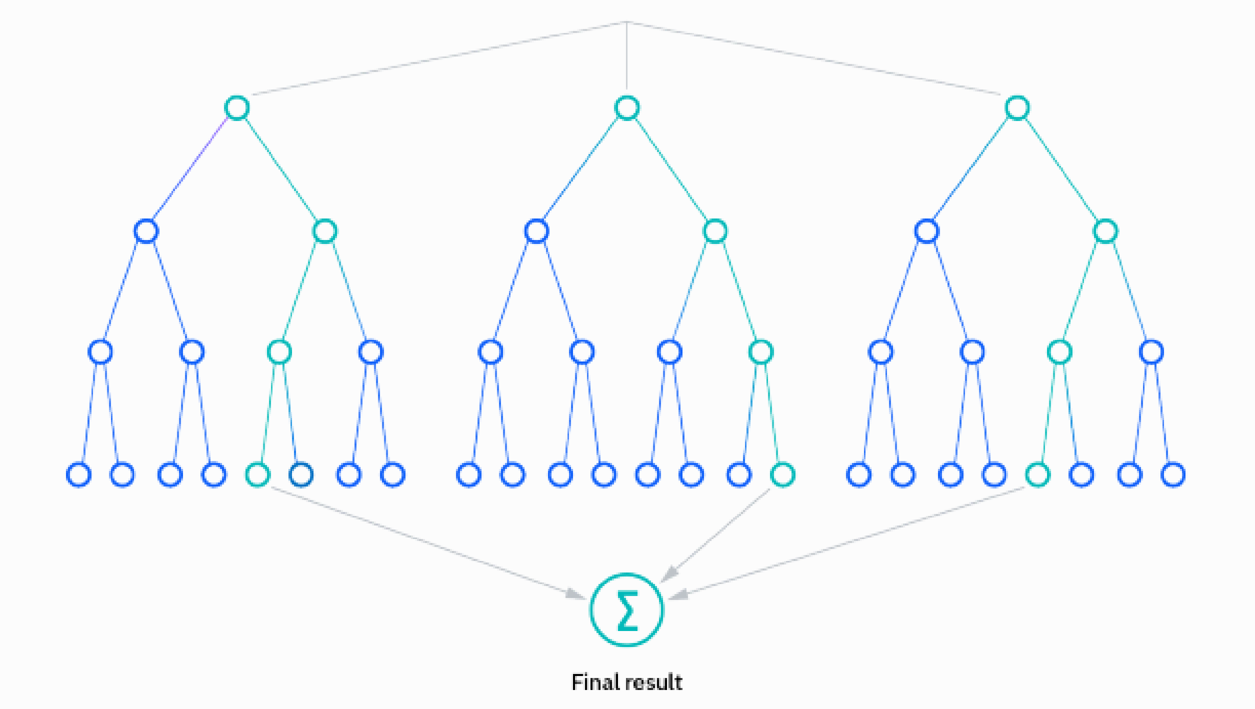
The random forest algorithm is an extension of the bagging method as it utilizes both bagging and feature randomness to create an uncorrelated forest of decision trees. Feature randomness, also known as feature bagging or "the random subspace method"(link resides outside ibm.com), generates a random subset of features, which ensures low correlation among decision trees. This is a key difference between decision trees and random forests. While decision trees consider all the possible feature splits, random forests only select a subset of those features.

If we go back to the "should I surf?" example, the questions that I may ask to determine the prediction may not be as comprehensive as someone else's set of questions. By accounting for all the potential variability in the data, we can reduce the risk of overfitting, bias, and overall variance, resulting in more precise predictions.

How it works

Random forest algorithms have three main hyperparameters, which need to be set before training. These include node size, the number of trees, and the number of features sampled. From there, the random forest classifier can be used to solve for regression or classification problems.

The random forest algorithm is made up of a collection of decision trees, and each tree in the ensemble is comprised of a data sample drawn from a training set with replacement, called the bootstrap sample. Of that training sample, one-third of it is set aside as test data, known as the out-of-bag (oob) sample, which we'll come back to later. Another instance of randomness is then injected through feature bagging, adding more diversity to the dataset and reducing the correlation among decision trees. Depending on the type of problem, the determination of the prediction will vary. For a regression task, the individual decision trees will be averaged, and for a classification task, a majority vote—i.e. the most frequent categorical variable—will yield the predicted class. Finally, the oob sample is then used for cross-validation, finalizing that prediction.



Benefits and challenges of random forest

There are a number of key advantages and challenges that the random forest algorithm presents when used for classification or regression problems. Some of them include:

Key Benefits

Reduced risk of overfitting: Decision trees run the risk of overfitting as they tend to tightly fit all the samples within training data. However, when there's a robust number of decision trees in a random forest, the classifier won’t overfit the model since the averaging of uncorrelated trees lowers the overall variance and prediction error.

Provides flexibility: Since random forest can handle both regression and classification tasks with a high degree of accuracy, it is a popular method among data scientists. Feature bagging also makes the random forest classifier an effective tool for estimating missing values as it maintains accuracy when a portion of the data is missing.

Easy to determine feature importance: Random forest makes it easy to evaluate variable importance, or contribution, to the model. There are a few ways to evaluate feature importance. Gini importance and mean decrease in impurity (MDI) are usually used to measure how much the model’s accuracy decreases when a given variable is excluded. However, permutation importance, also known as mean decrease accuracy (MDA), is another importance measure. MDA identifies the average decrease in accuracy by randomly permutating the feature values in oob samples.

Key Challenges

Time-consuming process: Since random forest algorithms can handle large data sets, they can be provide more accurate predictions, but can be slow to process data as they are computing data for each individual decision tree.

Requires more resources: Since random forests process larger data sets, they’ll require more resources to store that data.

More complex: The prediction of a single decision tree is easier to interpret when compared to a forest of them.

Random forest applications

The random forest algorithm has been applied across a number of industries, allowing them to make better business decisions. Some use cases include:

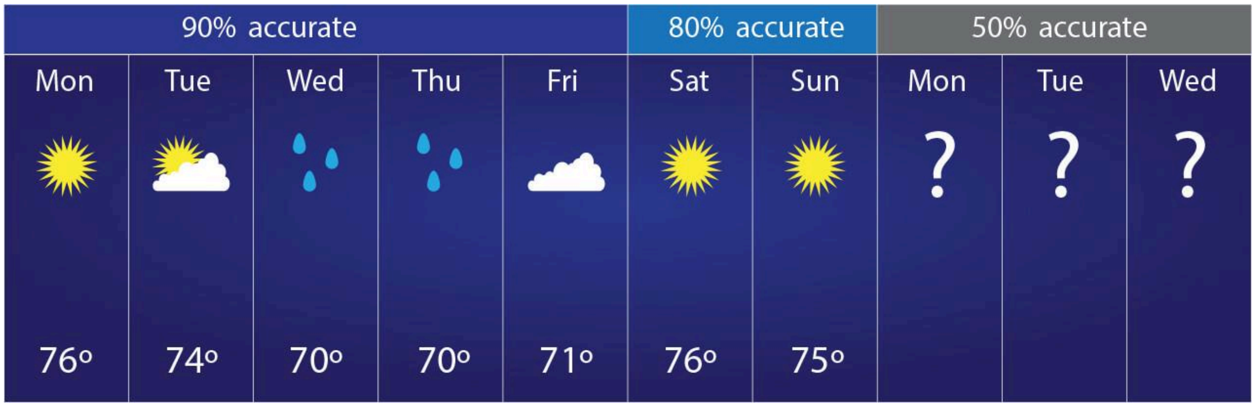
Finance: It is a preferred algorithm over others as it reduces time spent on data management and pre-processing tasks. It can be used to evaluate customers with high credit risk, to detect fraud, and option pricing problems.

Healthcare: The random forest algorithm has applications within computational biology (link resides outside ibm.com), allowing doctors to tackle problems such as gene expression classification, biomarker discovery, and sequence annotation. As a result, doctors can make estimates around drug responses to specific medications.

E-commerce: It can be used for recommendation engines for cross-sell purposes.

If you want to know what the weather will be like within the next week, a weather forecast can give you a really good idea of what to expect. A seven-day forecast can accurately predict the weather about 80 percent of the time and a five-day forecast can accurately predict the weather approximately 90 percent of the time.

However, a 10-day—or longer—forecast is only right about half the time. Meteorologists use computer programs called weather models to make forecasts. Since we can't collect data from the future, models have to use estimates and assumptions to predict future weather. The atmosphere is changing all the time, so those estimates are less reliable the further you get into the future.



90% accurate					80% accurate		50% accurate		
Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed
☀	⛅	💧	💧	☁	☀	☀	?	?	?
76º	74º	70º	70º	71º	76º	75º			

A seven-day forecast is fairly accurate, but forecasts beyond that range are less reliable.

How Weather Forecasts Are Made

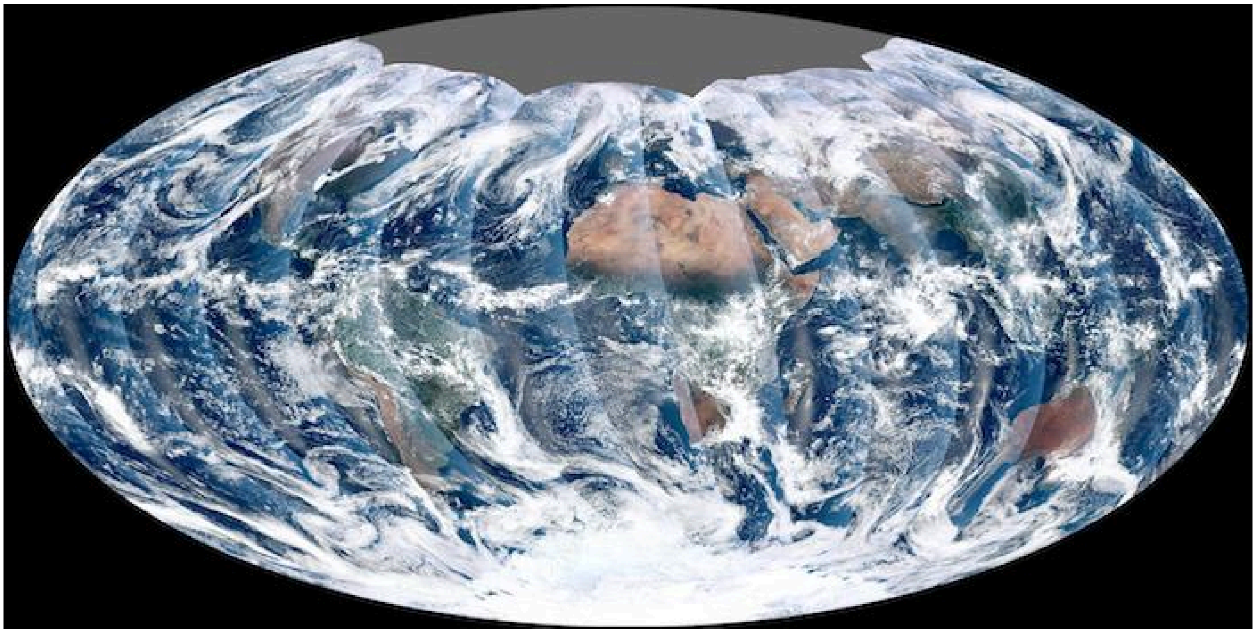
Some of the information needed to make a weather forecast comes from environmental satellites. NOAA, the National Oceanic and Atmospheric Administration, operates three types of environmental satellites that monitor Earth's weather:

Geostationary satellites: NOAA's Geostationary Environmental Operational Satellite-R (GOES-R) series satellites orbit approximately 22,000 miles above Earth and they provide a picture of what the weather is like right now.

“Geostationary” means that the satellites orbit at the same rate that the Earth rotates. This means they can collect near-continuous images over the same area. Because they focus on one spot, they can provide up-to-the-minute information about severe weather. This information helps forecasters understand how quickly a storm, such as a hurricane, is growing and moving.

An image of a Nor'easter off the coast of New England captured by a NOAA geostationary satellite called GOES-East. Credit: NOAA

Polar-orbiting satellites: Satellites as part of NOAA’s Joint Polar Satellite System (JPSS) orbit approximately 500 miles above Earth. They zip around our planet from pole to pole 14 times per day. Because they orbit while the Earth is rotating below, these satellites can see every part of Earth twice each day. Polar orbiting satellites can monitor the entire Earth’s atmosphere, clouds and oceans at high resolution. By watching these global weather patterns, polar orbiting satellites can help meteorologists accurately predict long-term forecasts—up to 7 days in the future.



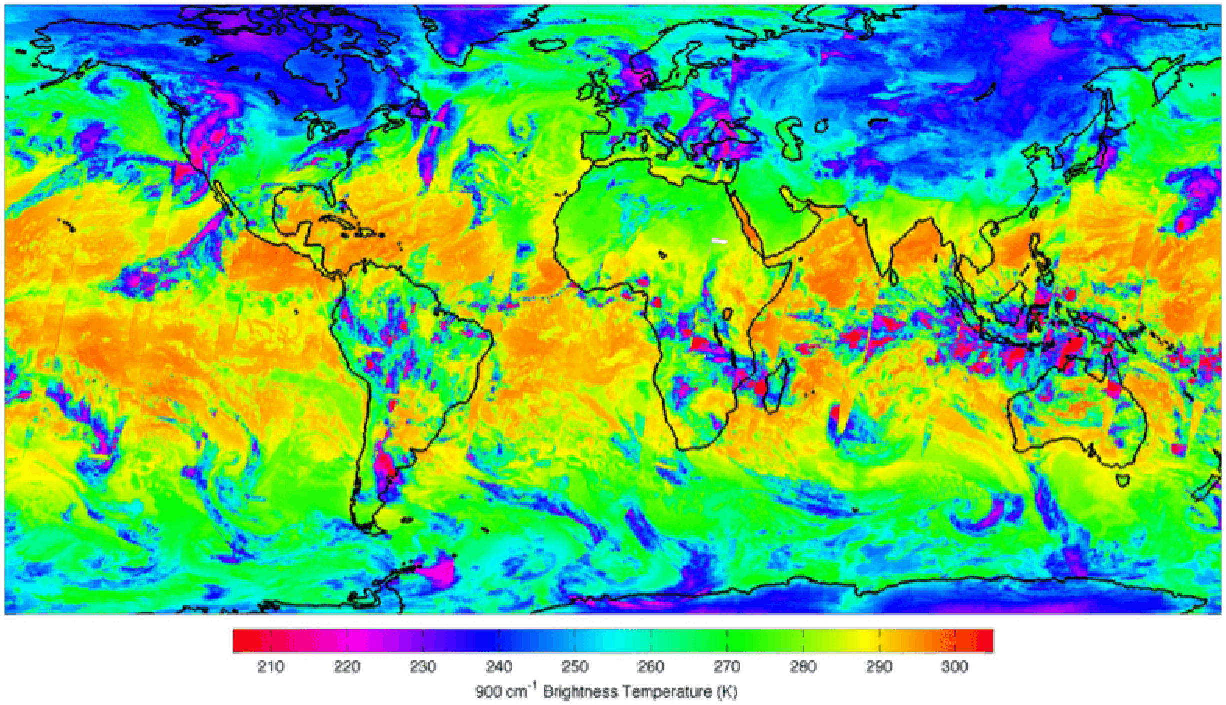
Polar orbiting satellites get a complete view of Earth each day by orbiting from pole to pole. Because the Earth spins, the satellite sees a different part of Earth with each orbit. It captures a picture of the entire planet as a series of wedges that then be pieced back together, as in the image above. Credit: NASA's NPP Land Product Evaluation and Testing Element

Deep space satellite: NOAA's Deep Space Climate Observatory (DSCOVR) orbits one million miles from Earth. It provides space weather alerts and forecasts while also monitoring the amounts of solar energy absorbed by Earth every day. DSCOVR also makes observations about ozone and aerosols in Earth's atmosphere. These factors are important in making air quality forecasts.

What is Space Weather?

Space weather describes the variations in the space environment between the Sun and Earth. Learn all about it in this video!

Polar orbiting satellites provide the information most useful for long-term weather forecasting. These satellites use instruments to measure energy, called radiation, emitted by the Earth and atmosphere. This information is incorporated into weather models, which in turn leads to more accurate weather forecasts. Other instruments can also be used to map sea surface temperature—an important factor in long-term weather forecasting.



Polar orbiting satellites monitor the whole Earth. This map, created with data from a polar orbiting satellite called Suomi-NPP, shows warm sea surface temperatures in orange and cold temperatures and high cloud tops in magenta. This information is important for long-term forecasting. Credit: NOAA

The satellite performs these accurate measurements all around the globe twice per day. This flood of data helps weather forecasters reliably predict the weather up to 7 days in advance. These measurements can also help forecasters predict seasonal weather patterns, such as El Niño and La Niña.

Polar orbiting satellites collect essential information for the models that forecast severe weather like hurricanes, tornadoes and blizzards days in advance. The information they collect is also needed to assess environmental hazards such as droughts, forest fires, poor air quality, and harmful coastal waters.

Using XGBoost in Python Tutorial

Discover the power of XGBoost, one of the most popular machine learning frameworks among data scientists, with this step-by-step tutorial in Python.

Updated Feb 2023 · 16 min read

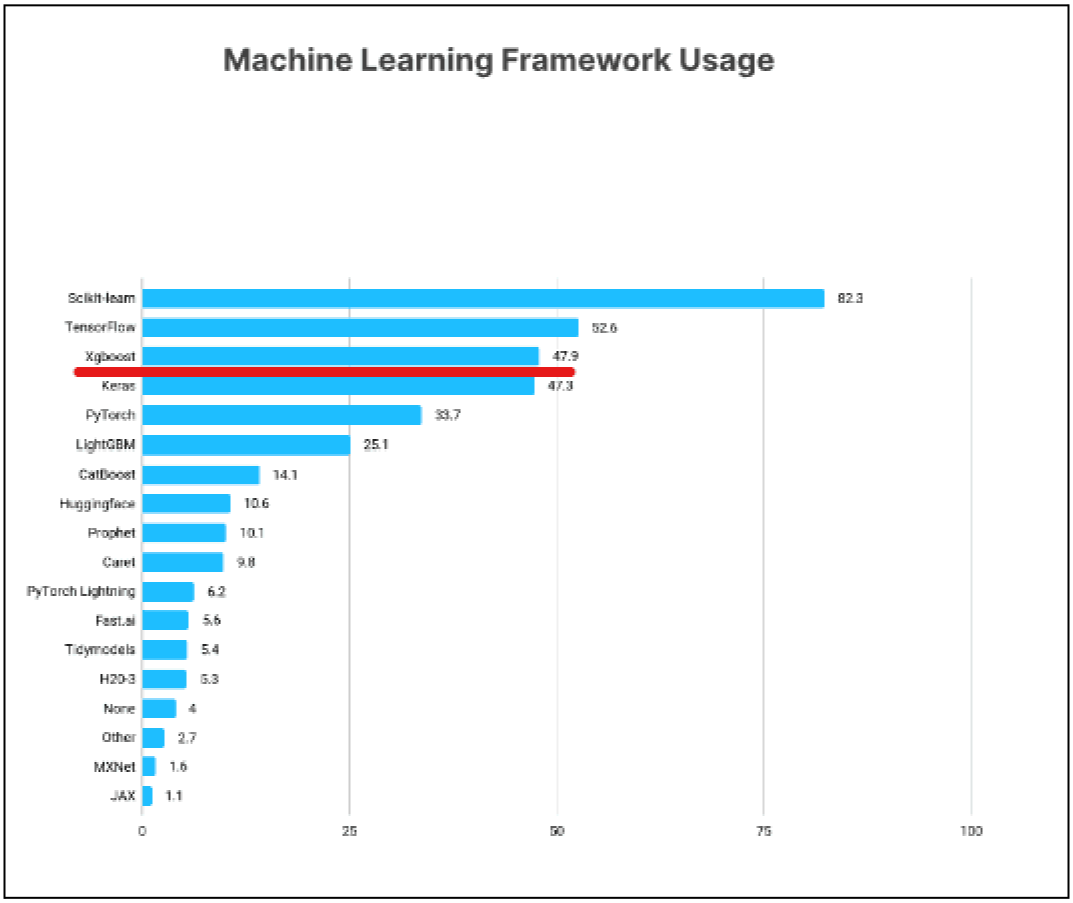
Contents

Experiment with this code in

Run Code

Share

XGBoost is one of the most popular machine learning frameworks among data scientists. According to the Kaggle **State of Data Science Survey 2021**, almost 50% of respondents said they used XGBoost, ranking below only TensorFlow and Sklearn.



https://www.kaggle.com/kaggle-survey-2021

This XGBoost tutorial will introduce the key aspects of this popular Python framework, exploring how you can use it for your own machine learning projects.

Watch and learn more about using XGBoost in Python in this video from our course.

What You Will Learn in This Python XGBoost Tutorial

Throughout this tutorial, we will cover the key aspects of XGBoost, including:

- Installation
- XGBoost DMatrix class
- XGBoost regression
- Objective and loss functions in XGBoost
- Building training and evaluation loops
- Cross-validation in XGBoost
- Building an XGBoost classifier
- Changing between Sklearn and native APIs of XGBoost

Let’s get started!

Run and edit the code from this tutorial online

Run code

XGBoost Installation

You can **install XGBoost like any other library through pip**. This method of installation will also include support for your machine's NVIDIA GPU. If you want to install the CPU-only version, you can go with conda-forge:

```
$ pip install --user xgboost

# CPU only

$ conda install -c conda-forge py-xgboost-cpu

# Use NVIDIA GPU

$ conda install -c conda-forge py-xgboost-gpu
```

Run code

Powered By

It’s recommended to install XGBoost in a virtual environment so as not to pollute your base environment.

We recommend running through the examples in the tutorial with a GPU-enabled machine. If you don’t have one, you can check out alternatives like **DataLab** or Google Colab.

If you decide to go with Colab, it has the old version of XGBoost installed, so you should call `pip install --upgrade xgboost` to get the latest version.

Loading and Exploring the Data

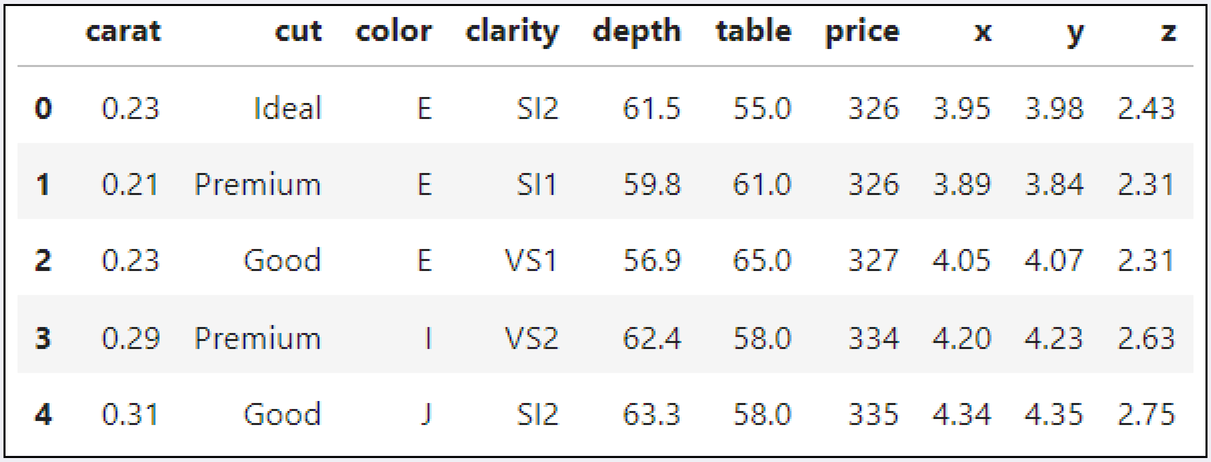
We will be working with the Diamonds dataset throughout the tutorial. It is built into the Seaborn library, or alternatively, you can also **download it from**

Kaggle. It has a nice combination of numeric and categorical features and over 50k observations that we can comfortably showcase all the advantages of XGBoost.

```
import seaborn as sns

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import warnings


warnings.filterwarnings(”ignore”)


diamonds = sns.load_dataset(”diamonds”)

diamonds.head()
```

Run code

Powered By



	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

```
>>> diamonds.shape
(53940, 10)
```

Run code

Powered By

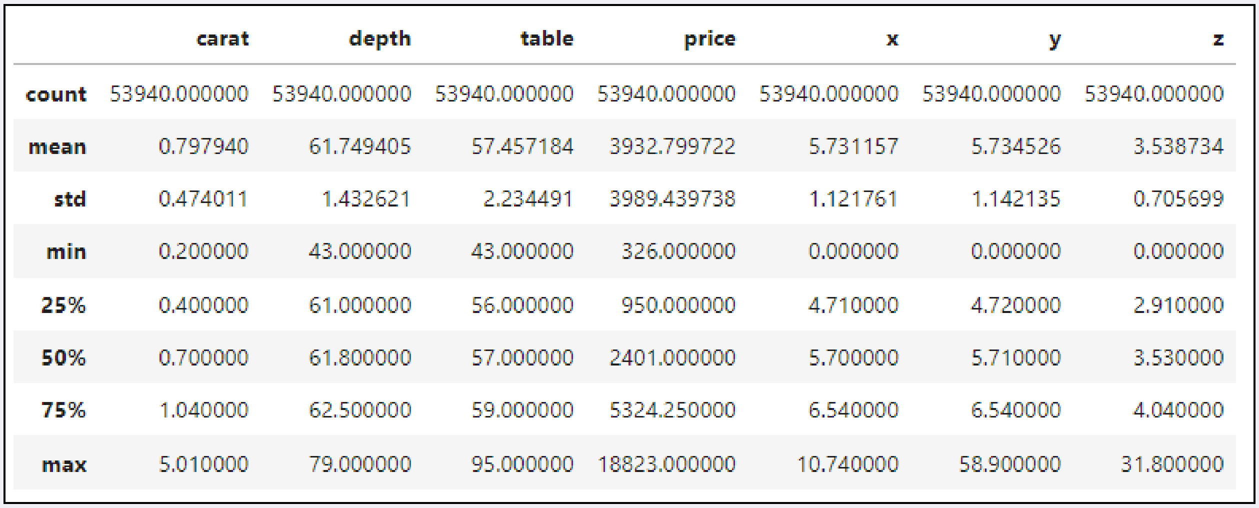
In a typical real-world project, you would want to spend a lot more time exploring the dataset and visualizing its features. But since this data comes built-in to Seaborn, it is relatively clean.

So, we will just look at the 5-number summary of the numeric and categorical features and get going. You can spend a few moments to familiarize yourself with the dataset.

```
diamonds.describe()
```

Run code

Powered By

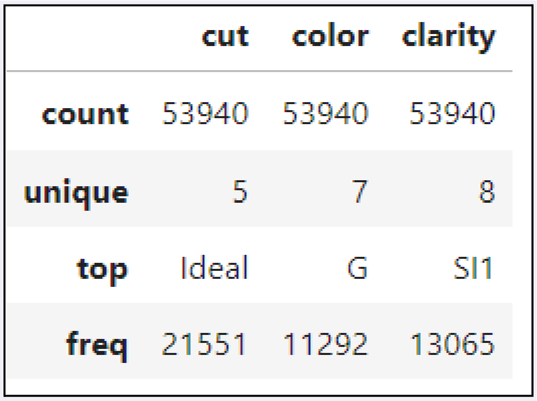


	carat	depth	table	price	x	y	z
count	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000
mean	0.797940	61.749405	57.457184	3932.799722	5.731157	5.734526	3.538734
std	0.474011	1.432621	2.234491	3989.439738	1.121761	1.142135	0.705699
min	0.200000	43.000000	43.000000	326.000000	0.000000	0.000000	0.000000
25%	0.400000	61.000000	56.000000	950.000000	4.710000	4.720000	2.910000
50%	0.700000	61.800000	57.000000	2401.000000	5.700000	5.710000	3.530000
75%	1.040000	62.500000	59.000000	5324.250000	6.540000	6.540000	4.040000
max	5.010000	79.000000	95.000000	18823.000000	10.740000	58.900000	31.800000

```
diamonds.describe(exclude=np.number)
```

Run code

Powered By



	cut	color	clarity
count	53940	53940	53940
unique	5	7	8
top	Ideal	G	SI1
freq	21551	11292	13065

How to Build an XGBoost DMatrix

After you are done with exploration, the first step in any project is framing the machine learning problem and extracting the feature and target arrays based on the dataset.

In this tutorial, we will first try to predict diamond prices using their physical measurements, so our target will be the price column.

So, we are isolating the features into X and the target into y:

```
from sklearn.model_selection import train_test_split

# Extract feature and target arrays
X, y = diamonds.drop('price', axis=1), diamonds[['price']]
```

Run code

Powered By

The dataset has three categorical columns. Normally, you would encode them with ordinal or one-hot encoding, but XGBoost has the ability to internally deal with categoricals.

The way to enable this feature is to cast the categorical columns into Pandas `category` data type (by default, they are treated as text columns):

```
# Extract text features
cats = X.select_dtypes(exclude=np.number).columns.tolist()

# Convert to Pandas category
for col in cats:
   X[col] = X[col].astype('category')
```

Run code

Powered By

Now, when you print the dtypes attribute, you'll see that we have three category features:

```
>>> X.dtypes
carat      float64
cut       category
color     category
clarity   category
depth      float64
table      float64
x          float64
y          float64
z          float64
dtype: object
```

Run code

Powered By

Let’s split the data into train, and test sets (0.25 test size):

```
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
```

Run code

Powered By

Now, the important part: XGBoost comes with its own class for storing datasets called DMatrix. It is a highly optimized class for memory and speed. That's why converting datasets into this format is a requirement for the native XGBoost API:

```
import xgboost as xgb

# Create regression matrices
dtrain_reg = xgb.DMatrix(X_train, y_train, enable_categorical=True)
dtest_reg = xgb.DMatrix(X_test, y_test, enable_categorical=True)
```

Run code

Powered By

The class accepts both the training features and the labels. To enable automatic encoding of Pandas category columns, we also set `enable_categorical` to True.

Note:

Why are we going with the native API of XGBoost, rather than its Scikit-learn API? While it might be more comfortable to use the Sklearn API at first, later, you'll realize that the native API of XGBoost contains some excellent features that the former doesn't support. So, better get used to it from the beginning. However, there is a section at the end where we show how to switch between APIs in a single line of code even after you have trained models.

Python XGBoost Regression

After building the DMatrices, you should choose a value for the `objective` parameter. It tells XGBoost the machine learning problem you are trying to solve and what metrics or loss functions to use to solve that problem.

For example, to predict diamond prices, which is a regression problem, you can use the common `reg:squarederror` objective. Usually, the name of the objective also contains the name of the loss function for the problem. For regression, it is common to use Root Mean Squared Error, which minimizes the square root of the squared sum of the differences between actual and predicted values. Here is how the metric would look like when implemented in NumPy:

```
import numpy as np

mse = np.mean((actual - predicted) ** 2)
rmse = np.sqrt(mse)
```

Run code

Powered By

We’ll learn classification objectives later in the tutorial.

A note on the difference between a loss function and a performance metric: A loss function is used by machine learning models to minimize the *differences* between the actual (ground truth) values and model predictions. On the other hand, a **metric** (or metrics) is chosen by the machine learning engineer to measure the *similarity* between ground truth and model predictions.

In short, a loss function should be minimized while a metric should be maximized. A loss function is used during training to guide the model on where to improve. A metric is used during evaluation to measure overall performance.

Training

The chosen objective function and any other hyperparameters of XGBoost should be specified in a dictionary, which by convention should be called params:

```
# Define hyperparameters
params = {"objective": "reg:squarederror", "tree_method": "gpu_hist"}
```

Run code

Powered By

Inside this initial params, we are also setting tree_method to gpu_hist, which enables GPU acceleration. If you don't have a GPU, you can omit the parameter or set it to hist.

Now, we set another parameter called num_boost_round, which stands for *number of boosting rounds*. Internally, XGBoost minimizes the loss function RMSE in small incremental rounds (more on this later). This parameter specifies the amount of those rounds.

The ideal number of rounds is found through hyperparameter tuning. For now, we will just set it to 100:

```
# Define hyperparameters
params = {"objective": "reg:squarederror", "tree_method": "gpu_hist"}

n = 100
model = xgb.train(
  params=params,
  dtrain=dtrain_reg,
  num_boost_round=n,
)
```

Run code

Powered By

When XGBoost runs on a GPU, it is blazing fast. If you didn’t receive any errors from the above code, the training was successful!

Evaluation

During the boosting rounds, the model object has learned all the patterns of the training set it possibly can. Now, we must measure its performance by testing it on unseen data. That's where our `dtest_reg` DMatrix comes into play:

```
from sklearn.metrics import mean_squared_error

preds = model.predict(dtest_reg)
```

Run code

Powered By

This step of the process is called model evaluation (or inference). Once you generate predictions with predict, you pass them inside `mean_squared_error` function of Sklearn to compare against `y_test`:

```
rmse = mean_squared_error(y_test, preds, squared=False)

print(f"RMSE of the base model: {rmse:.3f}")
RMSE of the base model: 543.203
```

Run code

Powered By

We’ve got a base score ~543$, which was the performance of a base model with default parameters. There are two ways we can improve it—by performing cross-validation and hyperparameter tuning. But before that, let’s see a quicker way of evaluating XGBoost models.

Using Validation Sets During Training

Training a machine learning model is like launching a rocket into space. You can control everything about the model up to the launch, but once it does, all you can do is stand by and wait for it to finish.

But the problem with our current training process is that we can’t even watch where the model is going. To solve this, we will use evaluation arrays that allow us to see model performance as it gets improved incrementally across boosting rounds.

First, let’s set up the parameters again:

```
params = {"objective": "reg:squarederror", "tree_method": "gpu_hist"}
n = 100
```

Run code

Powered By

Next, we create a list of two tuples that each contain two elements. The first element is the array for the model to evaluate, and the second is the array’s name.

```
evals = [(dtrain_reg, "train"), (dtest_reg, "validation")]
```

Powered By

When we pass this array to the `evals` parameter of `xgb.train`, we will see the model performance after each boosting round:

```
evals = [(dtrain_reg, "train"), (dtest_reg, "validation")]

model = xgb.train(
   params=params,
   dtrain=dtrain_reg,
   num_boost_round=n,
   evals=evals,
)
```

Run code

Powered By

You should get an output similar to the one below (shortened here to just 10 rows). You can see how the model minimizes the score from a whopping ~3931$ to just 543$.

What’s best is that we can see the model’s performance on both our training and validation sets. Usually, the training loss will be lower than validation since the model has already seen the former.

```
[0] train-rmse:3985.18329 validation-rmse:3930.52457
```

```
[1] train-rmse:2849.72257 validation-rmse:2813.20828
[2] train-rmse:2059.86648 validation-rmse:2036.66330
[3] train-rmse:1519.32314 validation-rmse:1510.02762
[4] train-rmse:1153.68171 validation-rmse:1153.91223
...
[95] train-rmse:381.93902 validation-rmse:543.56526
[96] train-rmse:380.97024 validation-rmse:543.51413
[97] train-rmse:380.75330 validation-rmse:543.36855
[98] train-rmse:379.65918 validation-rmse:543.42558

[99] train-rmse:378.30590 validation-rmse:543.20278
```

Run code

Powered By

In real-world projects, you usually train for thousands of boosting rounds, which means that many rows of output. To reduce them, you can use the `verbose_eval` parameter, which forces XGBoost to print performance updates every `vebose_eval` rounds:

```
params = {"objective": "reg:squarederror", "tree_method": "gpu_hist"}
n = 100

evals = [(dtest_reg, "validation"), (dtrain_reg, "train")]
```

```
model = xgb.train(
   params=params,
   dtrain=dtrain_reg,
   num_boost_round=n,
   evals=evals,
   verbose_eval=10 # Every ten rounds
)
```

```
[OUT]:

[0] train-rmse:3985.18329 validation-rmse:3930.52457
[10] train-rmse:550.08330 validation-rmse:590.15023
[20] train-rmse:488.51248 validation-rmse:551.73431
[30] train-rmse:463.13288 validation-rmse:547.87843
[40] train-rmse:447.69788 validation-rmse:546.57096
[50] train-rmse:432.91655 validation-rmse:546.22557
[60] train-rmse:421.24046 validation-rmse:546.28601
[70] train-rmse:408.64125 validation-rmse:546.78238
[80] train-rmse:396.41125 validation-rmse:544.69846
[90] train-rmse:386.87996 validation-rmse:543.82192
[99] train-rmse:378.30590 validation-rmse:543.20278
```

Run code

Powered By

XGBoost Early Stopping

By now, you must have realized how important boosting rounds are. Generally, the more rounds there are, the more XGBoost tries to minimize the loss. But this doesn’t mean the loss will always go down. Let’s try with 5000 boosting rounds with the verbosity of 500:

```
params = {"objective": "reg:squarederror", "tree_method": "gpu_hist"}
n = 5000

evals = [(dtest_reg, "validation"), (dtrain_reg, "train")]


model = xgb.train(
   params=params,
   dtrain=dtrain_reg,
   num_boost_round=n,
   evals=evals,
   verbose_eval=250
)
```

```
[OUT]:

[0] train-rmse:3985.18329 validation-rmse:3930.52457
[500] train-rmse:195.89184 validation-rmse:555.90367
[1000] train-rmse:122.10746 validation-rmse:563.44888
[1500] train-rmse:84.18238 validation-rmse:567.16974
[2000] train-rmse:61.66682 validation-rmse:569.52584
[2500] train-rmse:46.34923 validation-rmse:571.07632
[3000] train-rmse:37.04591 validation-rmse:571.76912
[3500] train-rmse:29.43356 validation-rmse:572.43196
[4000] train-rmse:24.00607 validation-rmse:572.81287
[4500] train-rmse:20.45021 validation-rmse:572.89062
[4999] train-rmse:17.44305 validation-rmse:573.13200
```

Run code

Powered By

We get the lowest loss before round 500. After that, even though training loss keeps going down, the validation loss (the one we care about) keeps increasing.

When given an unnecessary number of boosting rounds, XGBoost starts to overfit and memorize the dataset. This, in turn, leads to validation performance drop because the model is memorizing instead of generalizing.

Remember, we want the **golden middle**: a model that learned just enough patterns in training that it gives the highest performance on the validation set. So, how do we find the perfect number of boosting rounds, then?

We will use a technique called **early stopping.** Early stopping forces XGBoost to watch the validation loss, and if it stops improving for a specified number of rounds, it automatically stops training.

This means we can set as high a number of boosting rounds as long as we set a sensible number of early stopping rounds.

For example, let’s use 10000 boosting rounds and set the `early_stopping_rounds` parameter to 50. This way, XGBoost will automatically stop the training if validation loss doesn't improve for 50 consecutive rounds.

```
n = 10000
```

```
model = xgb.train(
   params=params,
   dtrain=dtrain_reg,
   num_boost_round=n,
   evals=evals,
   verbose_eval=50,
   # Activate early stopping
   early_stopping_rounds=50
)
```

[OUT]:

```
[0] train-rmse:3985.18329 validation-rmse:3930.52457
[50] train-rmse:432.91655 validation-rmse:546.22557
[100] train-rmse:377.66173 validation-rmse:542.92457
[150] train-rmse:334.27548 validation-rmse:542.79733
[167] train-rmse:321.04059 validation-rmse:543.35679
```

Run code

Powered By

As you can see, the training stopped after the 167th round because the loss stopped improving for 50 rounds before that.

XGBoost Cross-Validation

At the beginning of the tutorial, we set aside 25% of the dataset for testing. The test set would allow us to simulate the conditions of a model in production, where it must generate predictions for unseen data.

But only a single test set would not be enough to measure how a model would perform in production accurately. For example, if we perform hyperparameter tuning using only a single training and a single test set, knowledge about the test set would still “leak out.” How?

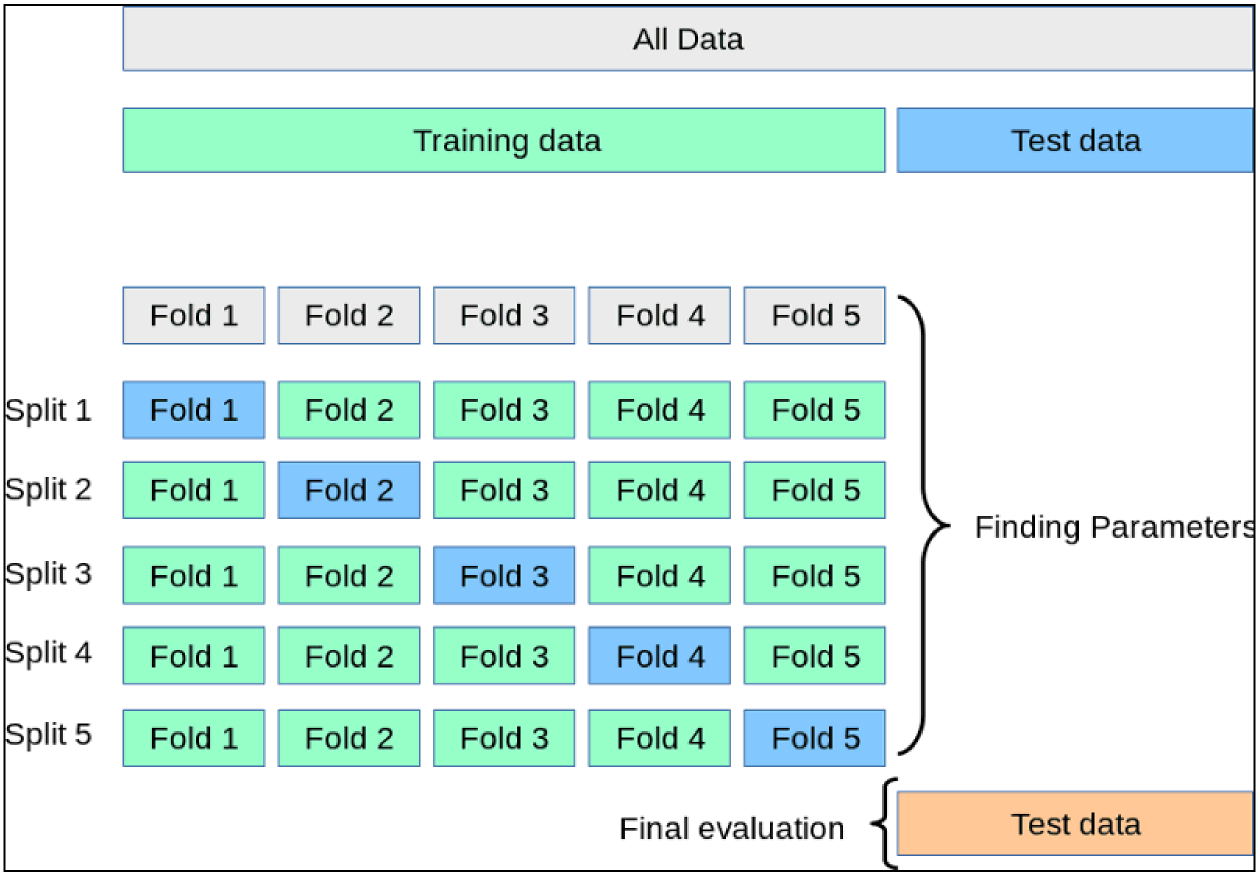
Since we try to find the best value of a hyperparameter by comparing the validation performance of the model on the test set, we will end up with a model that is configured to perform well *only* on that particular test set. Instead, we want a model that performs well across the board—on any test set we throw at it.

A possible workaround is splitting the data into three sets. The model trains on the first set, the second set is used for evaluation and hyperparameter tuning, and the third is the final one we test the model before production.

But when data is limited, splitting data into three sets will make the training set sparse, which hurts model performance.

The solution to all these problems is cross-validation. In cross-validation, we still have two sets: training and testing.

While the test set waits in the corner, we split the training into 3, 5, 7, or *k* splits or folds. Then, we train the model *k* times. Each time, we use *k-1* parts for training and the final *k*th part for validation. This process is called k-fold cross-validation:



Source: https://scikit-learn.org/stable/modules/cross_validation.html

Above is a visual depiction of a 5-fold cross-validation. After all folds are done, we can take the mean of the scores as the final, most realistic performance of the model.

Let’s perform this process in code using the `cv` function of XGB:

```
params = {"objective": "reg:squarederror", "tree_method": "gpu_hist"}
n = 1000

results = xgb.cv(
   params, dtrain_reg,
   num_boost_round=n,
   nfold=5,
   early_stopping_rounds=20
)
```

Run code

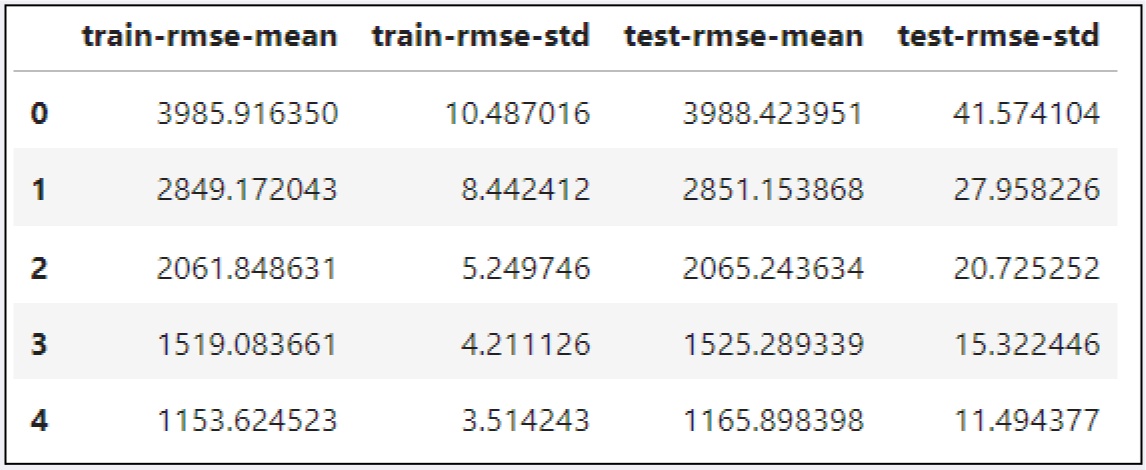
Powered By

The only difference with the train function is adding the nfold parameter to specify the number of splits. The results object is now a DataFrame containing each fold's results:

```
results.head()
```

Run code

Powered By



	train-rmse-mean	train-rmse-std	test-rmse-mean	test-rmse-std
0	3985.916350	10.487016	3988.423951	41.574104
1	2849.172043	8.442412	2851.153868	27.958226
2	2061.848631	5.249746	2065.243634	20.725252
3	1519.083661	4.211126	1525.289339	15.322446
4	1153.624523	3.514243	1165.898398	11.494377

It has the same number of rows as the number of boosting rounds. Each row is the average of all splits for that round. So, to find the best score, we take the minimum of the `test-rmse-mean` column:

```
best_rmse = results['test-rmse-mean'].min()

best_rmse
```

```
550.8959336674216
```

Run code

Powered By

Note that this method of cross-validation is used to see the true performance of the model. Once satisfied with its score, you must retrain it on the full data before deployment.

XGBoost Classification

Building an XGBoost classifier is as easy as changing the objective function; the rest can stay the same.

The two most popular classification objectives are:

- `binary:logistic` - binary classification (the target contains only two classes, i.e., cat or dog)

- multi:softprob - multi-class classification (more than two classes in the target, i.e., apple/orange/banana)

Performing binary and multi-class classification in XGBoost is almost identical, so we will go with the latter. Let’s prepare the data for the task first.

We want to predict the cut quality of diamonds given their price and their physical measurements. So, we will build the feature/target arrays accordingly:

```
from sklearn.preprocessing import OrdinalEncoder

X, y = diamonds.drop(”cut”, axis=1), diamonds[[’cut’]]

# Encode y to numeric
y_encoded = OrdinalEncoder().fit_transform(y)

# Extract text features
cats = X.select_dtypes(exclude=np.number).columns.tolist()

# Convert to pd.Categorical
for col in cats:
   X[col] = X[col].astype(’category’)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, random_state=1,
stratify=y_encoded)
```

Run code

Powered By

The only difference is that since XGBoost only accepts numbers in the target, we are encoding the text classes in the target with OrdinalEncoder of Sklearn.

Now, we build the DMatrices…

```
# Create classification matrices
dtrain_clf = xgb.DMatrix(X_train, y_train, enable_categorical=True)
dtest_clf = xgb.DMatrix(X_test, y_test, enable_categorical=True)
```

Run code

Powered By

…and set the objective to multi:softprob. This objective also requires the number of classes to be set by us:

```
params = {"objective": "multi:softprob", "tree_method": "gpu_hist", "num_class": 5}
n = 1000

results = xgb.cv(
  params, dtrain_clf,
  num_boost_round=n,
  nfold=5,
  metrics=["mlogloss", "auc", "merror"],
)
```

Run code

Powered By

During cross-validation, we are asking XGBoost to watch three classification metrics which report model performance from three different angles. Here is the result:

```
results.keys()

Index(['train-mlogloss-mean', 'train-mlogloss-std', 'train-auc-mean',

      'train-auc-std', 'train-merror-mean', 'train-merror-std',

      'test-mlogloss-mean', 'test-mlogloss-std', 'test-auc-mean',

      'test-auc-std', 'test-merror-mean', 'test-merror-std'],

     dtype='object')
```

Run code

Powered By

To see the best AUC score, we take the maximum of test-auc-mean column:

```
>>> results['test-auc-mean'].max()
0.9402233623451636
```

Run code

Powered By

Even the default configuration gave us 94% performance, which is great.

XGBoost Native vs. XGBoost Sklearn

So far, we have been using the native XGBoost API, but its Sklearn API is pretty popular as well.

Sklearn is a vast framework with many machine learning algorithms and utilities and has an API syntax loved by almost everyone. Therefore, XGBoost also offers XGBClassifier and XGBRegressor classes so that they can be integrated into the Sklearn ecosystem (at the loss of some of the functionality).

If you want to only use the Scikit-learn API whenever possible and only switch to native when you need access to extra functionality, there is a way.

After training the XGBoost classifier or regressor, you can convert it using the `get_booster` method:

```
import xgboost as xgb

# Train a model using the scikit-learn API
xgb_classifier = xgb.XGBClassifier(n_estimators=100, objective=’binary:logistic’,
tree_method=’hist’, eta=0.1, max_depth=3, enable_categorical=True)
xgb_classifier.fit(X_train, y_train)

# Convert the model to a native API model
```

```
model = xgb_classifier.get_booster()
```

Powered By

The model object will behave in the exact same way we've seen throughout this tutorial.

Conclusion

We’ve covered a lot of important topics in this XGBoost tutorial, but there are still so many things to learn.

You can check out the **XGBoost parameters page**, which teaches you how to configure the parameters to squeeze out every last performance from your models.

If you are looking for a comprehensive, all-in-one resource to learn the library, check out our **Extreme Gradient Boosting With XGBoost course**.