Learning XGBoost with R: A Practical Step-by-Step Guide

Authored by

Mohammed looti

November 6, 2025

Boosting is a highly effective and widely adopted technique in the field of machine learning, consistently producing models known for their superior predictive accuracy. This ensemble method sequentially combines numerous weak learners (typically decision trees) to form a powerful final model.

The most popular and efficient implementation of boosting today is **XGBoost**, which stands for "eXtreme Gradient Boosting." XGBoost is renowned for its speed, scalability, and optimization features, making it the go-to algorithm for structured data challenges.

This comprehensive tutorial guides you through a practical, step-by-step example of how to implement **XGBoost** to fit a robust boosted regression model using the statistical programming language, **R**.

We will cover everything from data preparation and model training to hyperparameter tuning and final performance evaluation.

Step 1: Initializing the R Environment and Loading Dependencies

Before we begin the modeling process, it is essential to ensure that the necessary libraries are loaded into the **R** environment. These packages provide the core functionalities required for data handling and model execution.

We require two primary packages for this demonstration: the specialized **xgboost** package for model fitting, and the general-purpose **caret** package, which simplifies many data preparation and model evaluation tasks.

library(xgboost) #for fitting the xgboost model
library(caret) #for general data preparation and model fitting

If these packages are not yet installed on your system, you must run the installation command (e.g., `install.packages("xgboost")`) before proceeding with the `library()` calls.

Step 2: Acquiring and Inspecting the Dataset

For this regression example, we will utilize the classic **Boston** dataset, which is conveniently housed within the **MASS** package. This dataset is frequently used for benchmarking regression techniques in machine learning.

The **Boston** dataset includes 13 distinct predictor variables, such as crime rate and proximity to employment centers, which we will use to forecast a single response variable named **medv**. The **medv** variable quantifies the median value of owner-occupied homes in thousands of dollars

across various census tracts near Boston.

Loading and viewing the structure of the data is a crucial preliminary step to understand the variable types and overall dimensions.

```
#load the data
data = MASS::Boston

#view the structure of the data
str(data)
```

```
'data.frame': 506 obs. of 14 variables:
$ crim : num 0.00632 0.02731 0.02729 0.03237 0.06905 ...
$ zn : num 18 0 0 0 0 0 12.5 12.5 12.5 12.5 ...
$ indus : num 2.31 7.07 7.07 2.18 2.18 2.18 7.87 7.87 7.87 7.87 ...
$ chas : int 0 0 0 0 0 0 0 0 0 0 ...
$ nox : num 0.538 0.469 0.469 0.458 0.458 0.458 0.524 0.524 0.524 0.524 ...
$ rm : num 6.58 6.42 7.18 7 7.15 ...
$ age : num 65.2 78.9 61.1 45.8 54.2 58.7 66.6 96.1 100 85.9 ...
$ dis : num 4.09 4.97 4.97 6.06 6.06 ...
$ rad : int 1 2 2 3 3 3 5 5 5 5 ...
$ tax : num 296 242 242 222 222 222 311 311 311 311 ...
$ ptratio: num 15.3 17.8 17.8 18.7 18.7 18.7 15.2 15.2 15.2 15.2 ...
$ black : num 397 397 393 395 397 ...
$ lstat : num 4.98 9.14 4.03 2.94 5.33 ...
$ medv : num 24 21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 ...
```

From the output structure, we confirm that the dataset comprises 506 observations (rows) and 14 total variables (columns). This setup is ideal for our regression analysis task.

Step 3: Preparing Data for XGBoost Training

To build a reliable machine learning model, we must partition the dataset into distinct training and testing sets. This step prevents overfitting the training data and allows us to rigorously evaluate the model's generalization ability on unseen data.

We use the powerful **createDataPartition()** function from the **caret** package to perform an 80/20 stratified split, allocating 80% of the data to the training set and reserving 20% for testing.

A critical requirement for the **xgboost** package is that the input data must be converted into a specific format: the predictor variables must be represented as a matrix structure. We achieve this

conversion using the **data.matrix()** function and, finally, structure the data into the specialized **xgb.DMatrix** object required for efficient XGBoost computation.

```
#make this example reproducible
set.seed(0)

#split into training (80%) and testing set (20%)
parts = createDataPartition(data$medv, p = .8, list = F)
train = data
test = data

#define predictor and response variables in training set
train_x = data.matrix(train)
train_y = train

#define predictor and response variables in testing set
test_x = data.matrix(test)
test_y = test

#define final training and testing sets (DMatrix format)
xgb_train = xgb.DMatrix(data = train_x, label = train_y)
xgb_test = xgb.DMatrix(data = test_x, label = test_y)
```

Step 4: Training the XGBoost Regression Model

With the data properly structured, we can now proceed to train the XGBoost model using the **xgb.train()** function. This process involves defining key hyperparameters that govern the structure and duration of the boosting process.

Two vital parameters are **nrounds** (the number of boosting iterations, or the number of trees to build) and **max.depth**. While we select 70 rounds here for simplicity, large-scale datasets often require hundreds or thousands of rounds, significantly impacting execution time. Monitoring performance across these rounds is crucial.

For **max.depth**, which controls the complexity of the individual decision trees, a low value (typically 2 or 3) is preferred. This constraint ensures that the individual base learners are weak, which is a foundational principle of the boosting algorithm and tends to yield more robust and accurate ensemble models. We also utilize a **watchlist** to monitor the performance (via RMSE) on both the training and testing sets during the iterative process.

```
#define watchlist
```

watchlist = list(train=xgb_train, test=xgb_test)

#fit XGBoost model and display training and testing data at each round
model = xgb.train(data = xgb_train, max.depth = 3, watchlist=watchlist, nrounds = 70)

train-rmse:10.167523 test-rmse:10.839775
train-rmse:7.521903 test-rmse:8.329679
train-rmse:5.702393 test-rmse:6.691415
train-rmse:4.463687 test-rmse:5.631310
train-rmse:3.666278 test-rmse:4.878750
train-rmse:3.159799 test-rmse:4.485698
train-rmse:2.855133 test-rmse:4.230533
train-rmse:2.603367 test-rmse:4.099881
train-rmse:2.445718 test-rmse:4.084360
train-rmse:2.327318 test-rmse:3.993562
train-rmse:2.267629 test-rmse:3.944454
train-rmse:2.189527 test-rmse:3.930808
train-rmse:2.119130 test-rmse:3.865036
train-rmse:2.086450 test-rmse:3.875088
train-rmse:2.038356 test-rmse:3.881442
train-rmse:2.010995 test-rmse:3.883322
train-rmse:1.949505 test-rmse:3.844382
train-rmse:1.911711 test-rmse:3.809830
train-rmse:1.888488 test-rmse:3.809830
train-rmse:1.832443 test-rmse:3.758502
train-rmse:1.816150 test-rmse:3.770216
train-rmse:1.801369 test-rmse:3.770474
train-rmse:1.788891 test-rmse:3.766608
train-rmse:1.751795 test-rmse:3.749583
train-rmse:1.713306 test-rmse:3.720173
train-rmse:1.672227 test-rmse:3.675086
train-rmse:1.648323 test-rmse:3.675977
train-rmse:1.609927 test-rmse:3.745338
train-rmse:1.594891 test-rmse:3.756049
train-rmse:1.578573 test-rmse:3.760104
train-rmse:1.559810 test-rmse:3.727940
train-rmse:1.547852 test-rmse:3.731702
train-rmse:1.534589 test-rmse:3.729761
train-rmse:1.520566 test-rmse:3.742681
train-rmse:1.495155 test-rmse:3.732993

```
train-rmse:1.467939 test-rmse:3.738329
train-rmse:1.446343 test-rmse:3.713748
train-rmse:1.435368 test-rmse:3.709469
train-rmse:1.401356 test-rmse:3.710637
train-rmse:1.390318 test-rmse:3.709461
train-rmse:1.372635 test-rmse:3.708049
train-rmse:1.367977 test-rmse:3.707429
train-rmse:1.359531 test-rmse:3.711663
train-rmse:1.335347 test-rmse:3.709101
train-rmse:1.331750 test-rmse:3.712490
train-rmse:1.313087 test-rmse:3.722981
train-rmse:1.284392 test-rmse:3.712840
train-rmse:1.257714 test-rmse:3.697482
train-rmse:1.248218 test-rmse:3.700167
train-rmse:1.243377 test-rmse:3.697914
train-rmse:1.231956 test-rmse:3.695797
train-rmse:1.219341 test-rmse:3.696277
train-rmse:1.207413 test-rmse:3.691465
train-rmse:1.197197 test-rmse:3.692108
train-rmse:1.171748 test-rmse:3.683577
train-rmse:1.156332 test-rmse:3.674458
train-rmse:1.147686 test-rmse:3.686367
train-rmse:1.143572 test-rmse:3.686375
train-rmse:1.129780 test-rmse:3.679791
train-rmse:1.111257 test-rmse:3.679022
train-rmse:1.093541 test-rmse:3.699670
train-rmse:1.083934 test-rmse:3.708187
train-rmse:1.067109 test-rmse:3.712538
train-rmse:1.053887 test-rmse:3.722480
train-rmse:1.042127 test-rmse:3.720720
train-rmse:1.031617 test-rmse:3.721224
train-rmse:1.016274 test-rmse:3.699549
train-rmse:1.008184 test-rmse:3.709522
train-rmse:0.999220 test-rmse:3.708000
train-rmse:0.985907 test-rmse:3.705192
```

Observing the output log, we track the testing RMSE (Root Mean Squared Error) across the 70 rounds. We notice a critical point: the minimum testing RMSE of **3.674458** is achieved specifically at round **56**.

After round 56, the training RMSE continues to decrease (meaning the model fits the training data better), but the testing RMSE begins to stagnate or slightly increase. This inflection point is the classic indicator of overfitting, where the model learns noise specific to the training set rather than general patterns.

Therefore, to obtain the most generalizable model, we define our final **XGBoost** model using 56 rounds, employing early stopping implicitly by selecting the optimal iteration count. We also set **verbose = 0** to suppress the iteration output during the final training run.

```
#define final model
final = xgboost(data = xgb_train, max.depth = 3, nrounds = 56, verbose = 0)
```

Step 5: Generating Predictions and Evaluating Performance

The final stage involves leveraging our optimized boosted model to generate predictions on the previously unseen testing set. These predictions represent the model's best estimate for the median house value of the Boston homes based on the input features.

To quantify the model's accuracy, we calculate three standard regression metrics:

MSE: Mean Squared Error, which penalizes larger errors more heavily.
MAE: Mean Absolute Error, representing the average magnitude of the errors.
RMSE: Root Mean Squared Error, the square root of MSE, which expresses the error in the same units as the response variable.

The following R code executes the prediction and calculates these three critical error measures:

```
mean((test_y - pred_y)^2) #mse
caret::MAE(test_y, pred_y) #mae
caret::RMSE(test_y, pred_y) #rmse
```

13.50164
2.409426
3.674457

The final Root Mean Squared Error value is calculated as **3.674457**. Given that the **medv** variable is measured in thousands of dollars, this means that, on average, the model's predictions differ from the actual observed median house values in the test set by approximately $3,674.46.

This **RMSE** serves as a benchmark for the model's performance. For a comprehensive analysis, one should compare this result against other regression techniques, such as multiple linear

regression, ridge regression, or principal components regression, to determine which modeling approach offers the most accurate and reliable predictions for the Boston housing market data.

The complete R source code utilized throughout this example is available for review and implementation here.