Boosting, Optimization, and Explainability

W. Evan Johnson, Ph.D.
Professor, Division of Infectious Disease
Director, Center for Data Science
Associate Director, Center for Biomedical Informatics and Health AI
Rutgers University – New Jersey Medical School

2025-11-27

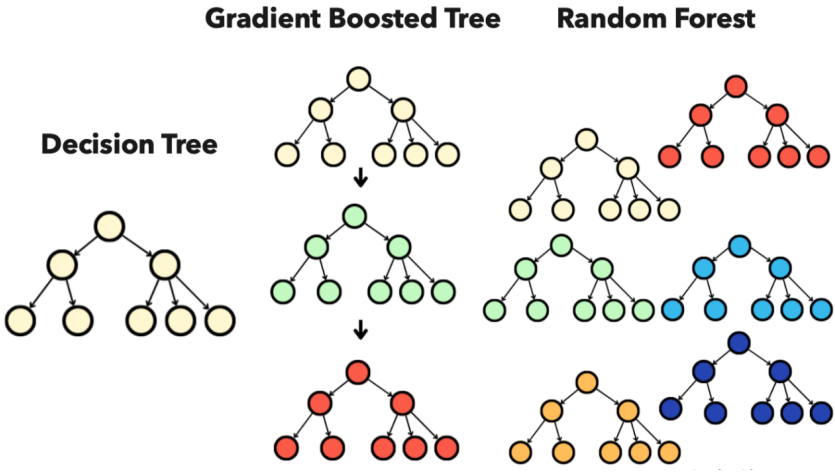
Ensemble Methods: Bagging vs. Boosting

Ensemble methods combine the predictions of multiple individual models (often called **weak learners**, like decision trees) to create a single, highly accurate predictor.

The two main strategies for building ensembles are:

- **Bagging (Bootstrap Aggregation)**: A *parallel* method where models are built independently and then averaged/combined. Used in **Random Forests**.
- **Boosting**: A *sequential* method where models are built iteratively, with each new model focusing on correcting the errors of the previous ones. Used in **XGBoost** (introduced soon).

Ensemble Methods: Bagging vs. Boosting



Bagging (Bootstrap Aggregation)

Bagging is a technique designed primarily to **reduce variance** in high-variance models, such as deep decision trees.

1. **Parallelism**: B independent models are trained simultaneously.
2. **Bootstrap Sampling**: Each model is trained on a unique *bootstrap sample* (sampling with replacement) of the original training data. This ensures the individual trees are diverse (random).
3. **Final Prediction**: The predictions from all B models are combined by:
   - **Averaging** (for regression, e.g., Random Forest).
   - **Majority Vote** (for classification, e.g., Random Forest).

The final averaged model has a significantly lower variance than any single tree, but the bias remains roughly the same.

Boosting (Sequential Error Correction)

Boosting is an iterative technique designed primarily to *reduce bias* and create a single strong learner from a sequence of weak learners.

1. **Sequential Dependence**: Models are trained *one after the other*.
2. **Error Focus**: Each subsequent model is trained to predict the *residuals* (the errors) of the combined ensemble created up to that point.
3. **Weighted Combination**: New models are added to the ensemble using a small *learning rate* (η) to ensure the model learns slowly and prevents dramatic overfitting.

Boosting can achieve very high predictive accuracy but requires careful tuning to avoid overfitting the training data, as seen with the complexity parameters in XGBoost.

Introducing Gradient Boosting Machines (GBMs)

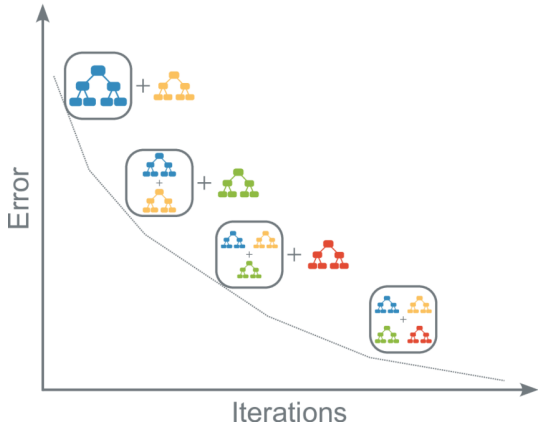
Gradient Boosting Machines (GBMs) is the general class of ensemble methods that utilizes the boosting principle.

How it relates to Boosting and XGBoost:

- Boosting Foundation: GBMs use the sequential error correction strategy (Boosting), where new trees correct the mistakes of the previous ensemble.
- Gradient Descent: The "Gradient" in GBM refers to the use of gradient descent—an optimization algorithm—to precisely identify the errors (residuals) and determine the ideal direction for the next tree to move in to minimize the model's overall loss function.
- XGBoost is an Implementation: XGBoost is simply a modern, highly optimized, and regularized (L1/L2) implementation of the GBM algorithm.

GBMs are generally more complex to train than Random Forests but often achieve superior predictive accuracy by aggressively reducing model bias.

Introducing Gradient Boosting Machines (GBMs)



Gradient Boosting Machines (GBMs)

The core idea is **sequential error correction**:

1. **Tree 1 (Weak Learner):** Makes an initial prediction.
2. **Tree 2:** Is trained to predict and correct the **residual errors** (or gradients of the loss function) from the combined output of all previous trees.
3. This process continues, with each new tree $f_k(\mathbf{x})$ added to the ensemble to minimize the total error.

$$\hat{y} = \hat{y}_0 + \sum_{k=1}^{K} \eta \cdot f_k(\mathbf{x})$$

The final prediction is the sum of the initial prediction ($\hat{y}_0$) and all subsequent tree predictions, scaled by the **learning rate** (η).

GBM Example: Iris

```r
# Load the iris dataset and set up training control
data("iris")
set.seed(42) # Set seed for reproducibility

# 5-fold Cross-Validation
ctrl <- trainControl(method = "cv", number = 5)

# GBM Tuning Grid
gbmGrid <- expand.grid(n.trees = c(50, 100, 150),
                       interaction.depth = c(1, 3, 5),
                       shrinkage = c(0.1),
                       n.minobsinnode = 10)
```

GBM Example: Iris

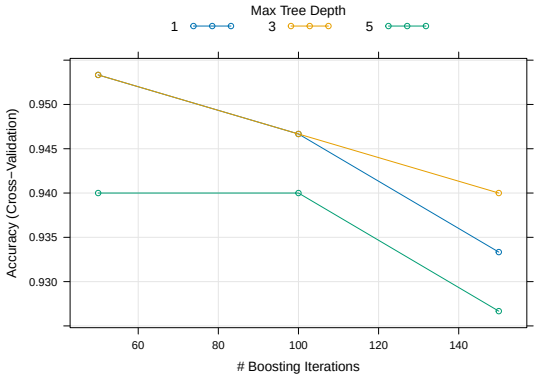
```r
# Train the GBM model
gbm_fit <- train(Species ~ ., data = iris,
  method = "gbm", trControl = ctrl,
  verbose = FALSE, tuneGrid = gbmGrid)

# Print best model results
print(gbm_fit)
```

```
## Stochastic Gradient Boosting
##
## 150 samples
##   4 predictor
##   3 classes: 'setosa', 'versicolor', 'virginica'
##
```

GBM Example: Iris

```
plot(gbm_fit)
```



GBM Example: Iris

```r
gbm_predictions <- predict(gbm_fit, newdata = iris)
confusionMatrix(gbm_predictions, iris$Species)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   setosa versicolor virginica
##   setosa         50          0         0
##   versicolor      0         49         0
##   virginica       0          1        50
##
## Overall Statistics
##
##                Accuracy : 0.9933
##                  95% CI : (0.9634, 0.9998)
##     No Information Rate : 0.3333
##     P-Value [Acc > NIR] : < 2.2e-16
```

Extreme Gradient Boosting (XGBoost)

XGBoost (Extreme Gradient Boosting) is a highly optimized and scalable implementation of **Gradient Boosting Machines (GBMs)**. It builds upon the idea of decision trees and boosting.

XGBoost: Key Enhancements

XGBoost improves upon traditional GBMs with several critical enhancements for performance and generalization:

- **Regularization:** Includes both **L1** (Lasso) and **L2** (Ridge) terms in the objective function to explicitly **penalize complexity** and prevent overfitting, which is common in boosting.
- **Parallel Computing:** Optimized to run tree construction in parallel, offering significant **speed gains** compared to older GBM implementations.

XGBoost: Key Enhancements (continued)

- **Handling Missing Values:** It has a built-in method to intelligently handle missing data by allowing the algorithm to **learn the best path** (left or right split) for null values.
- **Learning Rate (η):** A small η (e.g., 0.01 to 0.3) is used to **shrink** the contribution of each tree, making the learning process slow and more robust, requiring more trees (`nrounds`).

XGBoost often yields superior accuracy compared to Random Forests, but remains relatively interpretable through Feature Importance plots.

XGBoost: Iris Example

```r
data("iris")
set.seed(42)  # reproducibility
ctrl <- trainControl(method = "cv", number = 5) # 5-fold CV
xgbGrid <- expand.grid(
  nrounds = c(50, 100, 150),      # number of boosting iterations
  max_depth = c(1, 3, 5),         # tree depth
  eta = 0.1,                      # learning rate
  gamma = 0,                      # min loss reduction
  colsample_bytree = 1,           # feature subsample
  min_child_weight = 1,           # minimum sum of instance weight
  subsample = 1                   # row subsample
)
```

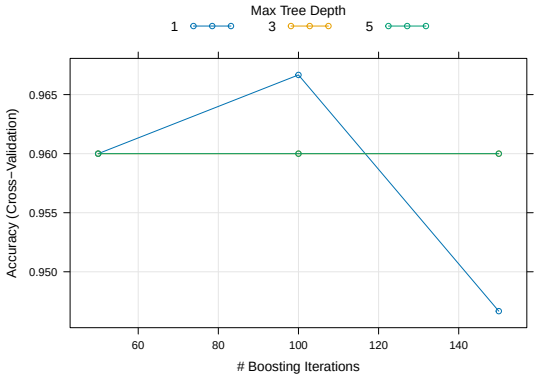
XGBoost: Iris Example

```r
xgb_fit <- train(
  Species ~ ., data = iris,
  method = "xgbTree",
  trControl = ctrl,
  tuneGrid = xgbGrid,
  verbose = FALSE
)
```

```
## [20:31:54] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated
## [20:31:54] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated
## [20:31:54] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated
## [20:31:54] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated
## [20:31:54] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated
## [20:31:54] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated
```

XGBoost: Iris Example

```
plot(xgb_fit)
```



XGBoost: Iris Example

```r
xgb_predictions <- predict(xgb_fit, newdata = iris)
confusionMatrix(xgb_predictions, iris$Species)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   setosa versicolor virginica
##   setosa         50          0         0
##   versicolor      0         47         2
##   virginica       0          3        48
##
## Overall Statistics
##
##                Accuracy : 0.9667
##                  95% CI : (0.9239, 0.9891)
##     No Information Rate : 0.3333
##     P-Value [Acc > NIR] : < 2.2e-16
```

LIME: Local Interpretable Model-agnostic Explanations

LIME focuses on explaining an **individual prediction** by building a simple, interpretable model *around* the prediction point.

Intuition: Trusting a local neighborhood.

1. **Select a prediction**: Choose one data point **x** to explain.
2. **Generate neighbors**: Perturb the data point **x** to create many new, slightly modified "neighbor" points.

LIME: Local Interpretable Model-agnostic Explanations

3. **Weight neighbors**: Weight these neighbors based on their proximity to **x**.
4. **Train a simple model*}**: Train a simple, interpretable model (like a linear regression or a shallow decision tree) on these weighted, perturbed data points.
5. **Explain}** The parameters of the simple model locally approximate the behavior of the complex model and provide the feature contribution for that single prediction.

LIME: Local Interpretable Model-agnostic Explanations

Key Feature: LIME explanations are **local**—they only explain *why* a specific data point **x** received its prediction.

LIME Example: GBM on Iris Data

```r
# install.packages("lime")
library(lime)
iris_features <- iris[, -5]

# Create a LIME explainer for the GBM model
gbm_explainer <- lime(
  x = iris_features,
  model = gbm_fit
)
```

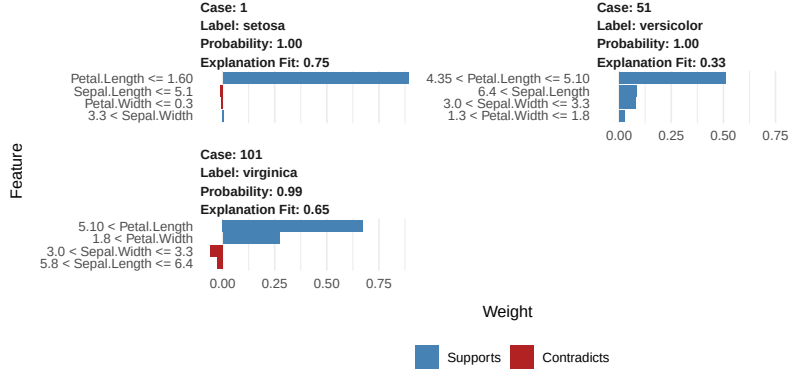
LIME Example: GBM on Iris Data

```r
# first 2 iris samples
samples_to_explain <- iris_features[c(1,51,101), ]

gbm_lime_results <- explain(
  x = samples_to_explain,
  explainer = gbm_explainer,
  n_features = 4,
  n_labels = 1
)
```

LIME Example: GBM on Iris Data

```
plot_features(gbm_lime_results)
```



SHAP: SHapley Additive exPlanations

SHAP connects optimal feature attribution with game theory, providing a single, consistent framework to measure the contribution of each feature.

SHAP values are based on **Shapley values** from cooperative game theory.

SHAP: SHapley Additive exPlanations

- The features of a model are considered the "players" in a "coalition."
- The prediction is the "payout."
- The SHAP value for a feature is its average marginal contribution across all possible feature combinations (coalitions).

SHAP: SHapley Additive exPlanations

Key Properties:

- **Consistency**: A feature that contributes more (or the same) to the prediction should always be assigned a higher (or equal) SHAP value.
- **Model-Agnostic**: Can be used for any model, although faster implementations exist for tree ensembles (Tree SHAP).
- **Global and Local**: SHAP can explain a single prediction (**Local**) and can be aggregated to visualize overall feature importance and relationships (**Global**).

SHAP: SHapley Additive exPlanations

Interpretation: A SHAP value represents the feature's contribution to the prediction, pushing it either higher or lower than the average prediction.

Session Info

```
sessionInfo()
```

```
## R version 4.5.1 (2025-06-13)
## Platform: aarch64-apple-darwin20
## Running under: macOS Sequoia 15.6.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRlapack.dylib;  LAPACK version 3.12.1
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Denver
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] lime_0.5.3      lubridate_1.9.4 forcats_1.0.0   stringr_1.5.1
##  [5] dplyr_1.1.4     purrr_1.1.0     readr_2.1.5     tidyr_1.3.1
##  [9] tibble_3.3.0    tidyverse_2.0.0 caret_7.0-1     lattice_0.22-7
## [13] ggplot2_3.5.2
##
## loaded via a namespace (and not attached):
```