Lecture 17

Decision Trees, ensembles

Ensembles for decision trees are extremely popular in machine learning. Let's start with an overview of some of the most common algorithms of this sort, and then we'll look more closely at specific algorithms. These methods leverage the idea that while a single decision tree might be prone to overfitting or have high variance, combining multiple trees can produce a more robust and accurate model. The three most commonly used ensemble methods based on decision trees are **Bagging (Bootstrap Aggregating), Random Forests,** and **Boosting** (e.g., AdaBoost, Gradient Boosting).

***1. Bagging (Bootstrap Aggregating)***

*Overview:*

- Process: Bagging involves creating multiple versions of the training dataset by bootstrapping (random sampling with replacement). A decision tree is then trained on each bootstrapped dataset. Finally, the predictions of all trees are averaged (for regression) or voted (for classification) to obtain the final prediction.
- Example: Randomly create 100 different training datasets from the original dataset. Train a decision tree on each of these datasets. For any new input, the output prediction is the average prediction of all 100 trees.

*Strengths:*

- Reduces Variance: By averaging the predictions of multiple trees, bagging reduces the variance, leading to a more stable and accurate model.
- Handles Overfitting: Since each tree is trained on a different subset of the data, bagging helps to mitigate the risk of overfitting that a single decision tree might have.

*Weaknesses:*

- Interpretability: The model becomes less interpretable compared to a single decision tree because it's based on an ensemble of many trees.
- Computational Cost: Training multiple trees can be computationally expensive, especially with large datasets.

***2. Random Forest***

*Overview:*

- Process: A Random Forest is an extension of bagging. In addition to bootstrapping the dataset, Random Forests also introduce a random selection of features at each split in the tree. This adds another layer of randomness, which further reduces correlation among the individual trees and improves generalization.
- Example: Suppose you have a dataset with 10 features. Instead of considering all 10 features at each split, Random Forest might randomly choose only 3 features to consider when determining the best split.

*Strengths:*

- Higher Accuracy: Random Forests generally achieve better accuracy than individual decision trees due to the combination of multiple decorrelated trees.
- Feature Importance: Random Forests provide an estimate of feature importance, helping to understand which variables contribute the most to the model.

- Random Forest can also be more robust to working with missing values, so that they need not be imputed.

*Weaknesses:*

- Complexity: Random Forest models are even more complex and harder to interpret than bagged trees.
- Training Time: Training a large number of trees with random subsets of features can be time-consuming.

***3. Boosting***

*Overview:*

- Process: Boosting involves training trees sequentially, where each new tree tries to correct the errors made by the previous ones. The model gives more weight to the data points that were misclassified (or had high residuals) by the previous trees, focusing on "hard" examples. Common boosting algorithms include AdaBoost and Gradient Boosting Machines (GBMs).
- Example: Train the first decision tree on the entire dataset. Then, calculate the errors (or residuals) and give more weight to the misclassified points. Train the second tree on this weighted dataset, and repeat the process iteratively.

*Strengths:*

- High Predictive Power: Boosting methods, particularly Gradient Boosting, often achieve very high predictive accuracy.
- Flexibility: Boosting can adapt to different types of loss functions, making it applicable to both regression and classification problems.

*Weaknesses:*

- Overfitting: Boosting is prone to overfitting if not properly regularized (e.g., by controlling tree depth, learning rate, or number of trees).
- Training Time: Boosting can be slower to train compared to other ensemble methods because of its sequential nature.

***4. Stacking***

*Overview:*

- Process: Stacking involves training several different models (which could be decision trees or other types of models) and then combining their predictions using another model (called a meta-learner). The idea is to let the meta-learner identify the best way to combine the base models' predictions.
- Example: Train a Random Forest, Gradient Boosting, and a Support Vector Machine (SVM) on the same dataset. Use their predictions as input features to a logistic regression model, which acts as the meta-learner to produce the final prediction.

*Strengths:*

- Leverages Multiple Models: By combining different types of models, stacking can capture diverse patterns in the data, leading to improved performance.
- Flexibility: Any type of model can be used as a base learner or meta-learner.

*Weaknesses:*

- *Complexity:* Stacking adds an additional layer of complexity and requires careful tuning to prevent overfitting.
- Training Time: Since multiple models are trained, and their predictions are used as features for the meta-learner, stacking can be computationally intensive.

Let's look more closely at two specific algorithms and how they work (big picture), then we'll dig into the specifics of each.

**Random Forest Algorithm**
***Overview:*** Random Forest is an ensemble learning method that combines the predictions of multiple decision trees to improve accuracy and reduce overfitting.

***Steps:***

1. *Bootstrap Sampling:* Randomly select a subset of the training data with replacement (bootstrap sampling) to create a new training dataset for each tree. This means some data points may be repeated while others may be left out (out-of-bag data).
2. *Train Decision Trees:* For each tree, randomly select a subset of features at each split. This step introduces diversity among the trees by decorrelating them. Grow each decision tree to its maximum depth without pruning.
3. *Prediction:* For classification, each tree in the forest casts a "vote" for the predicted class. The final prediction is made by majority voting (the class with the most votes). For regression, the predictions from all trees are averaged to produce the final prediction.
4. *Out-of-Bag Error (Optional):* The data points that were not included in the bootstrap sample (out-of-bag data) can be used to estimate the model's accuracy. This is an internal validation method that does not require a separate validation set.

***Key Points:***

- *Feature Randomness:* Randomly selecting features at each split helps to reduce the correlation between trees and thus lowers variance.
- *Number of Trees:* The more trees, the better the model generalization, up to a certain point. However, after a certain number of trees, the model may achieve diminishing returns in terms of performance.
- *Hyperparameters:* Important hyperparameters include the number of trees, the number of features to consider at each split, and the maximum depth of each tree.

**AdaBoost Algorithm**
***Overview:*** AdaBoost (Adaptive Boosting) is a boosting algorithm that focuses on improving the performance of weak learners, typically decision trees, by iteratively adjusting the weights of incorrectly classified instances.

***Steps:***

1. *Initialize Weights:* Start by assigning equal weights to all training instances.
2. *Iterative Process:* For a specified number of iterations (or until the error is minimized):
    - *Train Weak Learner:* Train a weak learner (e.g., a decision tree stump with only one split) on the weighted dataset.
    - *Calculate Error:* Compute the error rate of the weak learner on the weighted training data. The error is calculated as the sum of the weights of the misclassified instances.

   - *Compute Learner Weight:* Calculate the weight of the weak learner based on its error rate. The lower the error, the higher the weight of the learner.
   - *Update Instance Weights:* Increase the weights of the misclassified instances and decrease the weights of the correctly classified ones. This allows the next learner to focus more on the harder-to-classify instances.
3. *Final Model:* Combine the weak learners using a weighted sum (for regression) or weighted majority voting (for classification). The weights are determined by the performance of each learner.
4. *Prediction:* For classification, predict the class that receives the highest weighted vote from all weak learners. For regression, predict the weighted sum of the predictions from all learners.

***Key Points:***

- *Weak Learners:* AdaBoost typically uses decision stumps (single-level decision trees) as weak learners. The idea is to combine many weak learners to form a strong learner.
- *Focus on Hard Cases:* AdaBoost iteratively adjusts the weights of misclassified instances, forcing the model to focus on the more difficult cases.
- *Robustness to Overfitting:* AdaBoost is less prone to overfitting compared to other boosting algorithms, especially when using weak learners.
- *Hyperparameters:* Key hyperparameters include the number of weak learners (iterations) and the learning rate, which controls the contribution of each learner.

***Summary of Differences:***

- *Random Forest* builds multiple full decision trees on bootstrapped data, each using a random subset of features. It reduces variance and overfitting by averaging multiple trees.
- *AdaBoost* sequentially builds weak learners, focusing on misclassified instances from previous learners. It reduces both bias and variance by iteratively refining the model, giving more importance to difficult cases.

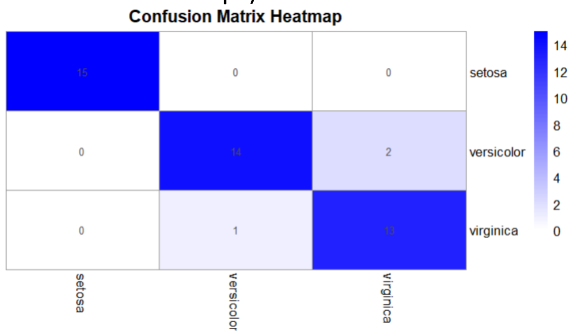
Both algorithms are highly effective ensemble methods but are used differently depending on the nature of the problem and the data at hand.

Let’s look at the code for the random forest algorithm. For this implementation, we’ll use decision tree code from a specific package so that we can focus on the bigger picture of the random forest itself. We’ll include some analysis and visualizations of the result at the end. We’ll implement the algorithm for classification.

```
# Install and load necessary libraries
if (!require(caret)) install.packages("caret", dependencies=TRUE)
library(rpart)
library(caret)
library(dplyr)

# Function to build a single decision tree on bootstrapped data
build_tree <- function(data, formula) {
  # Bootstrap sample
  bootstrapped_data <- data[sample(1:nrow(data), nrow(data), replace = TRUE), ]
  # Train a decision tree using rpart
```

```r
  tree_model <- rpart(formula, data = bootstrapped_data, method = "class", control =
rpart.control(minsplit = 2, cp = 0))
  return(tree_model)
}

# Function to predict using the Random Forest
predict_random_forest <- function(trees, test_data) {
  predictions <- sapply(trees, function(tree) predict(tree, newdata = test_data, type = "class"))
  # Aggregate predictions by majority voting
  final_predictions <- apply(predictions, 1, function(row) names(sort(table(row), decreasing =
TRUE)[1]))
  return(final_predictions)
}

# Random Forest function
random_forest <- function(train_data, test_data, formula, n_trees = 100) {
  trees <- list()
  for (i in 1:n_trees) {
    tree <- build_tree(train_data, formula)
    trees[[i]] <- tree
  }

  # Make predictions on the test set
  predictions <- predict_random_forest(trees, test_data)

  return(list(predictions = predictions))
}

# Split the iris dataset
set.seed(123)
train_index <- createDataPartition(iris$Species, p = 0.7, list = FALSE)
train_data <- iris[train_index, ]
test_data <- iris[-train_index, ]

# Run the Random Forest
set.seed(123)
rf_results <- random_forest(train_data, test_data, Species ~ ., n_trees = 100)

# Predictions
rf_predictions <- rf_results$predictions

# Confusion Matrix and Metrics
conf_matrix <- confusionMatrix(factor(rf_predictions, levels = levels(test_data$Species)),
test_data$Species)
print(conf_matrix)

# Load the necessary package for visualization
if (!require(pheatmap)) install.packages("pheatmap", dependencies=TRUE)
```

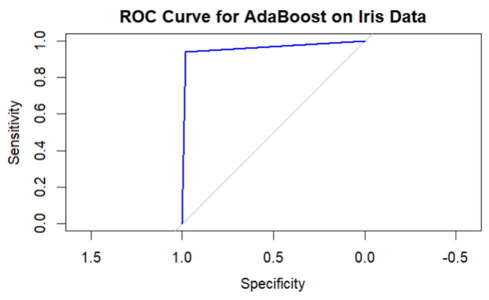
```
library(pheatmap)

# Create a heatmap of the confusion matrix
conf_matrix_df <- as.data.frame.matrix(conf_matrix$table)
pheatmap(conf_matrix_df,
         cluster_rows = FALSE,
         cluster_cols = FALSE,
         display_numbers = TRUE,
         number_format = "%.0f",
         color = colorRampPalette(c("white", "blue"))(100),
         main = "Confusion Matrix Heatmap")
```

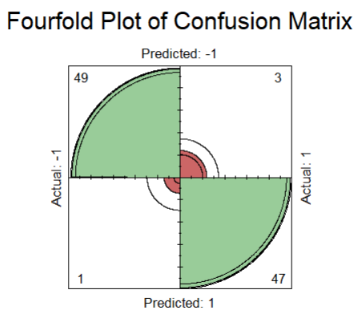


Next, we'll consider AdaBoost.

```
# Load necessary libraries
if (!require(rpart)) install.packages("rpart", dependencies=TRUE)
library(rpart)

# AdaBoost Algorithm
adaboost <- function(X, y, M = 10) {
  n <- nrow(X)
  weights <- rep(1/n, n)  # Initialize weights
  classifiers <- list()  # To store classifiers
  classifier_weights <- numeric(M)  # To store classifier weights

  for (m in 1:M) {
    # Train weak learner
    model <- rpart(as.factor(y) ~ ., data = X, weights = weights, method = "class", control =
rpart.control(maxdepth = 1))

    # Make predictions
    predictions <- predict(model, X, type = "class")

    # Calculate error and classifier weight
    incorrect <- as.numeric(predictions != y)
    error <- sum(weights * incorrect) / sum(weights)

    alpha <- 0.5 * log((1 - error) / max(error, 1e-10))
    classifier_weights[m] <- alpha
    # Update weights
    weights <- weights * exp(-alpha * (2 * as.numeric(predictions == y) - 1))
    weights <- weights / sum(weights)  # Normalize weights

    classifiers[[m]] <- model  # Store the weak learner
  }

  return(list(classifiers = classifiers, classifier_weights = classifier_weights))
}

predict_adaboost <- function(model, X) {
  M <- length(model$classifiers)
  n <- nrow(X)

  # Initialize prediction matrix
  pred_matrix <- matrix(0, nrow = n, ncol = M)

  for (m in 1:M) {
    pred_matrix[, m] <- as.numeric(predict(model$classifiers[[m]], X, type = "class"))
  }

  # Calculate weighted vote (This needs to be adjusted)
  final_pred <- apply(pred_matrix, 1, function(row) {
    weighted_sum <- sum(model$classifier_weights * (2 * row - 3))  # 2*row-3 converts 1/-1 to 1/-
1 scale for the vote.
    return(ifelse(weighted_sum > 0, 1, -1))  # Convert back to class labels
  })

  return(final_pred)
}

# Convert iris data into binary classification problem
iris_binary <- iris[iris$Species != "setosa", ]  # Remove setosa to make it binary
iris_binary$Species <- ifelse(iris_binary$Species == "versicolor", 1, -1)  # 1 for versicolor, -1 for
virginica

# Check the balance of the dataset
table(iris_binary$Species)

# Retrain the model
set.seed(123)
ada_model <- adaboost(iris_binary[, -5], iris_binary$Species, M = 10)

# Predict on the training data
predictions <- predict_adaboost(ada_model, iris_binary[, -5])

# Confusion Matrix
conf_matrix <- table(Predicted = predictions, Actual = iris_binary$Species)
```

```
print(conf_matrix)

# Accuracy
accuracy <- sum(diag(conf_matrix)) / sum(conf_matrix)
print(paste("Accuracy:", round(accuracy, 3)))

# Visualization
roc_obj <- roc(iris_binary$Species, predictions)
plot(roc_obj, col = "blue", main = "ROC Curve for AdaBoost on Iris Data")
auc(roc_obj)

# Calculate confusion matrix
conf_matrix <- table(Predicted = predictions, Actual = iris_binary$Species)

# Ensure confusion matrix is 2x2 (it should be)
print(conf_matrix)

# Plot using fourfoldplot
fourfoldplot(conf_matrix, color = c("#CC6666", "#99CC99"),
        conf.level = 0.95, margin = 1, main = "Fourfold Plot of Confusion Matrix")
```





Note that this fourfoldplot of the confusion matrix only works when the confusion matrix is 2x2 (binary classifier). However, it is a nice alternative to a heatmap.

Resources:

1. https://medium.com/@pankaj_pandey/decision-tree-ensemble-model-f422ba280fa8
2. https://www.almabetter.com/bytes/tutorials/data-science/ensembles-of-decision-tree
3. https://towardsdatascience.com/decision-trees-understanding-the-basis-of-ensemble-methods-e075d5bfa704