Gradient Boosting & Bagging

Boosting is another famous ensemble learning technique in which we are not concerned with reducing the variance of learners like in **Bagging** where our aim is to reduce the high variance of learners by averaging lots of models fitted on bootstrapped data samples generated with replacement from training data, so as to avoid overfitting.

Another major difference between both the techniques is that in Bagging the various models which are generated are independent of each other and have equal weightage. Whereas Boosting is a sequential process in which each next model which is generated is added so as to improve a bit from the previous model.

Simply saying each of the model that is added to mix is added so as to improve on the performance of the previous collection of models. In Boosting we do weighted averaging. Both the ensemble techniques are common in terms of generating lots of models on training data and using their combined power to increase the accuracy of the final model which is formed by combining them.

But Boosting is more towards **Bias** i.e. simple learners or more specifically **Weak learners**. Now a weak learner is a learner which always learns something i.e. does better than chance and also has error rate less than 50%. The best example of a weak learner is a Decision tree. This is the reason we generally use ensemble technique on decision trees to improve its accuracy and performance.

Implementing Gradient Boosting in R

Let's use **gbm** package in R to fit gradient boosting model.

```
require(gbm)

require (MASS) #package with the boston housing dataset

# Separating training and test data

train=sample (1:506, size=374)
```

We will use the Boston housing data to predict the median value of the houses.

```
Boston.boost <- gbm (medv ~., data = Boston[train,], distribution = "gaussian", n.trees = 10000,

                shrinkage = 0.01, interaction.depth = 4)

Boston.boost
Summary (Boston.boost) # Summary gives a table of Variable Importance and a plot of Variable Importance
```

The above Boosted Model is a Gradient Boosted Model which generates 10000 trees and the shrinkage parameter lambda=0.01 which is also a sort of **learning rate**. Next parameter is the interaction depth d which is the total splits we want to do. So here each tree is a small tree with only 4 splits.

Bagging consists of taking multiple subsets of the training data set, then building multiple independent decision tree models, and then average the models allowing to create a very performant predictive model compared to the classical CART model.

B**oosting**, which is similar to the bagging method, except that the trees are grown sequentially: each successive tree is grown using information from previously grown trees, with the aim to minimize the error of the previous models.

Shrinkage parameter lambda is a small positive value, typically comprised between 0.01 and
There are different variants of boosting, including Adaboost, gradient boosting and stochastic gradient boosting. A weak model is one whose error rate is only slightly better than random guessing.

Stochastic gradient boosting, implemented in the R package **xgboost**, is the most commonly used boosting technique, which involves resampling of observations and columns in each round. It offers the best performance. **xgboost** stands for **extremely gradient boosting**.
Boosting can be used for both classification and regression problems.

Whereas bagging builds an ensemble of deep independent trees, GBMs build an ensemble of shallow and weak successive trees with each tree learning and improving on the previous. When combined, these many weak successive trees produce a powerful "committee" that are often hard to beat with other algorithms.

- **gbm:** Training and tuning with the gbm package
- **xgboost**: Training and tuning with the xgboost package
- **library(gbm)** # basic implementation
- **library(xgboost)** # a faster implementation of gbm

Important note: Tree-based methods tend to perform well on unprocessed data (i.e. without normalizing, centering, scaling features)

<u>Advantages & Disadvantages</u>

Advantages:

- Often provides predictive accuracy that cannot be beat.
- Lots of flexibility - can optimize on different loss functions and provides several hyperparameter tuning options that make the function fit very flexible.
- No data pre-processing required - often works great with categorical and numerical values as is.
- Handles missing data - imputation not required.

Disadvantages:

- GBMs will continue improving to minimize all errors. This can overemphasize outliers and cause overfitting. Must use cross-validation to neutralize.
- Computationally expensive - GBMs often require many trees (>1000) which can be time and memory exhaustive.
- The high flexibility results in many parameters that interact and influence heavily the behaviour of the approach (number of iterations, tree depth, regularization parameters, etc.). This requires a large grid search during tuning.

- Less interpretable although this is easily addressed with various tools (variable importance, partial dependence plots, LIME, etc.).

Training weak models: A weak model is one whose error rate is only slightly better than random guessing. The idea behind boosting is that each sequential model builds a simple weak model to slightly improve the remaining errors. With regards to decision trees, shallow trees represent a weak learner. Commonly, trees with only 1-6 splits are used. Combining many weak models (versus strong ones) has a few benefits:

- **Speed**: Constructing weak models is computationally cheap.
- **Accuracy improvement**: Weak models allow the algorithm to *learn slowly*; making minor adjustments in new areas where it does not perform well. In general, statistical approaches that learn slowly tend to perform well.
- **Avoids overfitting**: Due to making only small incremental improvements with each model in the ensemble, this allows us to stop the learning process as soon as overfitting has been detected (typically by using cross-validation).

Sequential training with respect to errors: Boosted trees are grown sequentially; each tree is grown using information from previously grown trees.

Tuning
Part of the beauty and challenges of GBM is that they offer several tuning parameters. The beauty in this is GBMs are highly flexible. The challenge is that they can be time consuming to tune and find the optimal combination of hyperparameters. The most common hyperparameters that you will find in most GBM implementations include:

- **Number of trees:** The total number of trees to fit. GBMs often require many trees; however, unlike random forests GBMs can overfit so the goal is to find the optimal number of trees that minimize the loss function of interest with cross validation.
- **Depth of trees:** The number *d* of splits in each tree, which controls the complexity of the boosted ensemble. Often d=1d=1 works well, in which case each tree is a *stump* consisting of a single split. More commonly, d is greater than 1 but it is unlikely d>10d>10 will be required.
- **Learning rate:** Controls how quickly the algorithm proceeds down the gradient descent. Smaller values reduce the chance of overfitting but also increases the time to find the optimal fit. This is also called *shrinkage*.
- **Subsampling:** Controls whether or not you use a fraction of the available training observations. Using less than 100% of the training observations means you are implementing stochastic gradient descent. This can help to minimize overfitting and keep from getting stuck in a local minimum or plateau of the loss function gradient.

- **gbm**: The original R implementation of GBMs
- **xgboost**: A fast and efficient gradient boosting framework

gbm

The gbm R package is an implementation of extensions to Freund and Schapire's AdaBoost algorithm and Friedman's gradient boosting machine. This is the original R implementation of GBM. A presentation is available here by Mark Landry.

Features include:

- Stochastic GBM.
- Supports up to 1024 factor levels.
- Supports Classification and regression trees.
- Can incorporate many loss functions.
- Out-of-bag estimator for the optimal number of iterations is provided.
- Easy to overfit since early stopping functionality is not automated in this package.
- If internal cross-validation is used, this can be parallelized to all cores on the machine.
- Currently undergoing a major refactoring & rewrite (and has been for some time).
- GPL-2/3 License.

Basic implementation

The default settings in gbm includes a learning rate (shrinkage) of 0.001. This is a very small learning rate and typically requires a large number of trees to find the minimum MSE. However, gbm uses a default number of trees of 100, which is rarely sufficient. Consequently, I crank it up to 10,000 trees. The default depth of each tree (**interaction.depth**) is 1, which means we are ensembling a bunch of stumps. Lastly, I also include **cv.folds** to perform a 5-fold cross validation. The model took about 90 seconds to run and the results show that our MSE loss function is minimized with 10,000 trees.

```
# For reproducibility
set.seed(123)

# Train GBM model
gbm.fit <- gbm(formula = Sale_Price ~ ., distribution = "gaussian", data = ames_train, n.trees = 10000, interaction.depth = 1, shrinkage = 0.001, cv.folds = 5, n.cores = NULL, # will use all cores by default verbose = FALSE)

# Print results
print(gbm.fit)
## gbm(formula = Sale_Price ~ ., distribution = "gaussian", data = ames_train,
##     n.trees = 10000, interaction.depth = 1, shrinkage = 0.001,
##     cv.folds = 5, verbose = FALSE, n.cores = NULL)
## A gradient boosted model with gaussian loss function.
## 10000 iterations were performed.
## The best cross-validation iteration was 10000.
## There were 80 predictors of which 45 had non-zero influence.
```

xgboost

The xgboost R package provides an R API to "Extreme Gradient Boosting", which is an efficient implementation of gradient boosting framework (apprx 10x faster than gbm).
The xgboost/demo repository provides a wealth of information. You can also find a fairly comprehensive parameter tuning guide here. The xgboost package has been quite popular and successful on Kaggle for data mining competitions.

Features include:

- Provides built-in k-fold cross-validation
- Stochastic GBM with column and row sampling (per split and per tree) for better generalization.
- Includes efficient linear model solver and tree learning algorithms.
- Parallel computation on a single machine.
- Supports various objective functions, including regression, classification and ranking.
- The package is made to be extensible, so that users are also allowed to define their own objectives easily.
- Apache 2.0 License.

Basic implementation

XGBoost only works with matrices that contain all numeric variables; consequently, we need to one hot encode our data. **vtreat** is a robust package for data prep and helps to eliminate problems caused by missing values, novel categorical levels that appear in future data sets that were not in the training data, etc. However, **vtreat** is not very intuitive.

Boosting is one of the ensemble learning techniques in machine learning and widely used in regression and classification problems. The main concept of this method is to improve (boost) the week learners sequentially and increase the model accuracy with a combined model. There are several boosting algorithms such as **Gradient boosting, AdaBoost (Adaptive Boost), XGBoost** and others.

Boosting aims to build a set of weak learners (i.e. predictive models that are only slightly better than random chance) to create one 'strong' learner (i.e. a predictive model that predicts the response variable with a high degree of accuracy). Gradient Boosting is a boosting method which aims to optimise an arbitrary (differentiable) cost function (for example, squared error).

Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow
- `xgboost` for computing boosting algorithm

```
library(tidyverse)
library(caret)
library(xgboost)
```

Classification

Example of data set

Data set: `PimaIndiansDiabetes2` [in `mlbench` package], introduced in Chapter @ref(classification-in-r), for predicting the probability of being diabetes positive based on multiple clinical variables.

Randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data and remove NAs
data("PimaIndiansDiabetes2", package = "mlbench")
PimaIndiansDiabetes2 <- na.omit(PimaIndiansDiabetes2)
# Inspect the data
sample_n(PimaIndiansDiabetes2, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- PimaIndiansDiabetes2$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- PimaIndiansDiabetes2[training.samples, ]
test.data <- PimaIndiansDiabetes2[-training.samples, ]
```

Boosted classification trees

We'll use the `caret` workflow, which invokes the `xgboost` package, to automatically adjust the model parameter values, and fit the final best boosted tree that explains the best our data.

We'll use the following arguments in the function `train()`:

- `trControl`, to set up 10-fold cross validation

```
# Fit the model on the training set
set.seed(123)
model <- train(
  diabetes ~., data = train.data, method = "xgbTree",
  trControl = trainControl("cv", number = 10)
  )
# Best tuning parameter
model$bestTune
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 18     150         1 0.3     0              0.8                1         1
```

```
# Make predictions on the test data
predicted.classes <- model %>% predict(test.data)
head(predicted.classes)
```

```
## [1] neg pos neg neg pos neg
## Levels: neg pos
```

```
# Compute model prediction accuracy rate
mean(predicted.classes == test.data$diabetes)
```

```
## [1] 0.744
```

The prediction accuracy on new test data is 74%, which is good.

Variable importance

The function `varImp()` [in caret] displays the importance of variables in percentage:

```
varImp(model)
```

```
## xgbTree variable importance
##
##          Overall
## glucose   100.00
## mass       20.23
## pregnant   15.83
## insulin    13.15
## pressure    9.51
## triceps     8.18
## pedigree    0.00
## age         0.00
```

Regression

Similarly, you can build a random forest model to perform regression, that is to predict a continuous variable.

Example of data set

We'll use the `Boston` data set [in `MASS` package], introduced in Chapter @ref(regression-analysis), for predicting the median house value (`mdev`), in Boston Suburbs, using different predictor variables.

Randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model).

```
# Load the data
data("Boston", package = "MASS")
# Inspect the data
sample_n(Boston, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- Boston$medv %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- Boston[training.samples, ]
test.data <- Boston[-training.samples, ]
```

Boosted regression trees

Here the prediction error is measured by the RMSE, which corresponds to the average difference between the observed known values of the outcome and the predicted value by the model.

```
# Fit the model on the training set
set.seed(123)
model <- train(
  medv ~., data = train.data, method = "xgbTree",
  trControl = trainControl("cv", number = 10)
  )
# Best tuning parameter mtry
model$bestTune
# Make predictions on the test data
predictions <- model %>% predict(test.data)
head(predictions)
# Compute the average prediction error RMSE
RMSE(predictions, test.data$medv)
```

Here is all you need to do, to build a GBM model.

```
fitControl <- trainControl(method = "repeatedcv", number = 4, repeats = 4)
```

```
trainData$outcome1 <- ifelse(trainData$Disbursed == 1, "Yes","No")

set.seed(33)

gbmFit1 <- train(as.factor(outcome1) ~ ., data = trainData[,-26], method = "gbm", trControl = fitCont

rol,verbose = FALSE)
```

xgboost provides different training functions (i.e. **xgb.train** which is just a wrapper for **xgboost**). However, to train an XGBoost we typically want to use **xgb.cv**, which incorporates cross-validation.

Boosting is a class of ensemble learning techniques for regression and classification problems. Boosting aims to build a set of weak learners (i.e. predictive models that are only slightly better than random chance) to create one 'strong' learner (i.e. a predictive model that predicts the response variable with a high degree of accuracy). Gradient Boosting is a boosting method which aims to optimise an arbitrary (differentiable) cost function (for example, squared error).