Week 8 KNN and Ensemble Methods

Theory and Practice

Ying Lin, Ph.D

March 06, 2020

Outline

- K Nearest Neighbor algorithm
- Ensemble learning strategy
- Bagging: bootstrapping, random forest
- Boosting: Adaptive Boosting (AdaBoost), Gradient Boosting Machine (GBM) and eXtreme Gradient Boosting (XGBoost)
- R/Python demo

Overview of K Nearest Neighbor (KNN)

- Key ideas: data points closer to each other tend to behave similarly
- Lazy learner: no active model-building is done with training dataset
  - Vs. active learner
  - No training time, long prediction time. Too slow for an online system.
- Instance-based and distance-based learning
  - Sensitive towards irrelevant attributes
  - Sensitive towards outliers and noisy data
  - Requires data preprocessing: conversion from categorical to numerical attributes, standardization, normalization
- Non-parametric method: no assumption of data distribution
  - Could detect highly complex nonlinear decision boundary

Variance vs. Bias Tradeoff

- $Error = Variance + Bias$
- Variance: stability of model and predictions
- Bias: accuracy of the model/methodology
- Overfitting: high variance, low bias
- “No Free Lunch” theorem: trade off between variance and bias, no single algorithm will outperform other algorithms on all datasets

Hyper-Parameters of KNN

- Number of neighbors (*K*)
  - Too large: high bias
  - Too small: high variance (overfitting)
- Distance functions
  - Euclidean
  - Manhattan
  - Minkowski

Demo Dataset: Diabetes

```
# install.packages("mlbench")
library(caret)
library(mlbench)
data("PimaIndiansDiabetes")
diabetes <- PimaIndiansDiabetes
str(diabetes)
```

```
## 'data.frame':    768 obs. of  9 variables:
##  $ pregnant: num  6 1 8 1 0 5 3 10 2 8 ...
##  $ glucose : num  148 85 183 89 137 116 78 115 197 125 ...
##  $ pressure: num  72 66 64 66 40 74 50 0 70 96 ...
##  $ triceps : num  35 29 0 23 35 0 32 0 45 0 ...
##  $ insulin : num  0 0 0 94 168 0 88 0 543 0 ...
##  $ mass    : num  33.6 26.6 23.3 28.1 43.1 25.6 31 35.3 30.5 0 ...
##  $ pedigree: num  0.627 0.351 0.672 0.167 2.288 ...
##  $ age     : num  50 31 32 21 33 30 26 29 53 54 ...
##  $ diabetes: Factor w/ 2 levels "neg","pos": 2 1 2 1 2 1 2 1 2 2 ...
```

```
summary(diabetes)
```

```
##     pregnant         glucose         pressure         triceps
##  Min.   : 0.000   Min.   :  0.0   Min.   :  0.00   Min.   : 0.00
```

Train and Test Baseline KNN Model

```
library(caret)

set.seed(188)
train_index <- createDataPartition(diabetes$diabetes, p = 0.7, list = FALSE)

diabetes_train <- diabetes[train_index, ]
diabetes_test <- diabetes[-train_index, ]


model_knn <- train(diabetes ~ ., data = diabetes_train, method = "knn")
predict_knn <- predict(model_knn, newdata = diabetes_test)
```

Measure Model Performance: Hold-Out Method

```
confusionMatrix(predict_knn, diabetes_test$diabetes)
```

```
## Confusion Matrix and Statistics
##
##          Reference
## Prediction neg pos
##        neg 126  36
##        pos  24  44
##
##               Accuracy : 0.7391
##                 95% CI : (0.6773, 0.7946)
##    No Information Rate : 0.6522
##    P-Value [Acc > NIR] : 0.002949
##
##                  Kappa : 0.4041
##  Mcnemar's Test P-Value : 0.155580
##
##            Sensitivity : 0.8400
##            Specificity : 0.5500
##         Pos Pred Value : 0.7778
##         Neg Pred Value : 0.6471
##             Prevalence : 0.6522
##         Detection Rate : 0.5478
```

Measure Model Performance: Bootstrap Method

```
print(model_knn)
```

```
## k-Nearest Neighbors
##
## 538 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 538, 538, 538, 538, 538, 538, ...
## Resampling results across tuning parameters:
##
##   k  Accuracy   Kappa
##   5  0.6901734  0.3082508
##   7  0.6989885  0.3185598
##   9  0.7121062  0.3449934
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 9.
```

Pre-process Data

```
pre_process <- preProcess(diabetes_train, method = c("scale", "center"))
pre_process
```

```
## Created from 538 samples and 9 variables
##
## Pre-processing:
##   - centered (8)
##   - ignored (1)
##   - scaled (8)
```

```
diabetes_train1 <- predict(pre_process, newdata = diabetes_train)
diabetes_test1 <- predict(pre_process, newdata = diabetes_test)
summary(diabetes_train1)
```

```
##     pregnant          glucose           pressure           triceps
##  Min.   :-1.1089   Min.   :-3.7915   Min.   :-3.61879   Min.   :-1.2893
##  1st Qu.:-0.8243   1st Qu.:-0.6594   1st Qu.:-0.36502   1st Qu.:-1.2893
##  Median :-0.2550   Median :-0.1270   Median : 0.05482   Median : 0.1813
##  Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.00000   Mean   : 0.0000
##  3rd Qu.: 0.5989   3rd Qu.: 0.6169   3rd Qu.: 0.47466   3rd Qu.: 0.7328
##  Max.   : 3.7299   Max.   : 2.4414   Max.   : 2.78379   Max.   : 4.7771
##     insulin             mass             pedigree            age
```

New Model Using Standardized Dataset

```
model_knn1 <- train(diabetes ~ ., data = diabetes_train1, method = "knn")
predict_knn1 <- predict(model_knn1, newdata = diabetes_test1)

confusionMatrix(predict_knn1, diabetes_test1$diabetes, positive = "pos")
```

```
## Confusion Matrix and Statistics
##
##          Reference
## Prediction neg pos
##        neg 128  40
##        pos  22  40
##
##               Accuracy : 0.7304
##                 95% CI : (0.6682, 0.7866)
##    No Information Rate : 0.6522
##    P-Value [Acc > NIR] : 0.006878
##
##                  Kappa : 0.3729
##  Mcnemar's Test P-Value : 0.030850
##
##            Sensitivity : 0.5000
##            Specificity : 0.8533
##         Pos Pred Value : 0.6452
```

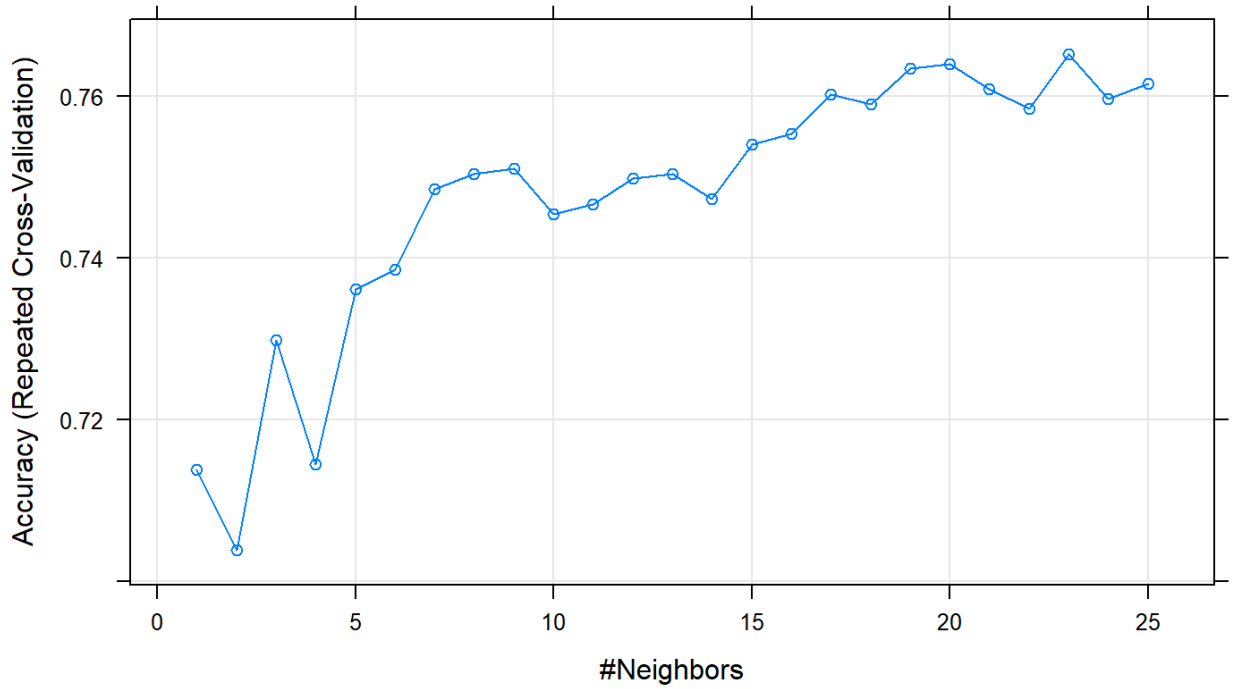
Tune the KNN Model

```
model_knn2 <- train(diabetes ~ ., data = diabetes_train1, method = "knn",
                    tuneGrid = data.frame(k = seq(1, 25)),
                    trControl = trainControl(method = "repeatedcv",
                                             number = 5, repeats = 3))
print(model_knn2)
```

```
## k-Nearest Neighbors
##
## 538 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 3 times)
## Summary of sample sizes: 430, 430, 431, 431, 430, 431, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    1  0.7137764  0.3664780
##    2  0.7038191  0.3375583
##    3  0.7298719  0.3901417
##    4  0.7144283  0.3457516
##    5  0.7360909  0.3933307
```

Sensitivity Analysis of KNN

```
plot(model_knn2)
```



Ensemble Learning Motivations

- Additive model: combine multiple classifiers together and take consensus
- Different mechanisms to produce multiple classifiers from the same datasets
  - Bagging (Bootstrap AGGregating): create multiple samples (with replacement) from the original dataset and train multiple classifiers in parallel; reduce variance
  - Boosting: an iterative algorithm that induces base (weak) learners sequentially (e.g. assigns different weights to the data points to focus on those tougher cases): reduce bias
  - Random Forest: train large number of decision stumps (smaller decision trees with less nodes) and use committee majority voting to decide the final classification

Illustratration of Benefit of Ensemble Learning

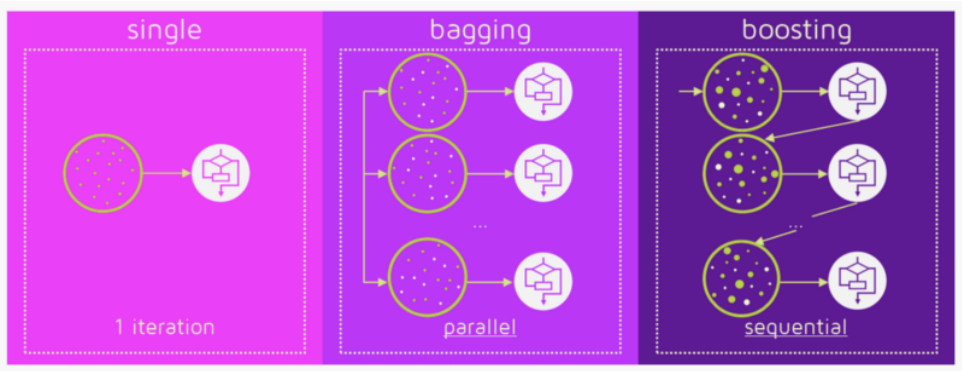
- Wisdom of the mass
- Case study
  - 25 weak learners with 30% error rate
  - What is the error rate of the ensemble classifier

$$= \sum_{i=13}^{25} C_i^{25} P^i (1-P)^{25-i}$$

$$= \sum_{i=13}^{25} C_i^{25} 0.7^i 0.3^{25-i}$$

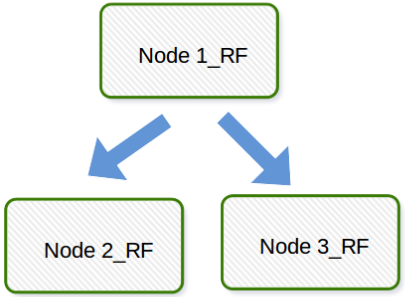
$$= 0.80$$

Bagging vs. Boosting



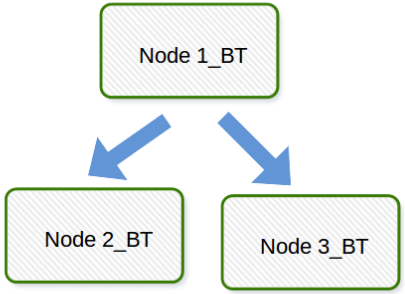
Random forests--

Only m<M features considered for each node for split



Bagging Trees--

All of M features considered for each node for a split



m can be selected via out-of-bag error, but ***m*** = sqrt(***M***) is a good value to start with

Bagging: Bootstrap Resampling Strategy

- Sampling with replacement
- 0.632 Rule
  - Roughly $1/3$ data won't be included in the bootstrap sample, therefore used for validation purpose
  - $lim_{n\to\infty}(1-\frac{1}{n})^n = e^{-1} \approx 0.368$
  - Simulation

```
mean(sapply(createResample(1:100, times = 10000), function(x) length(unique(x))))
```

```
## [1] 63.3761
```

Algorithms: Random Forest (1)

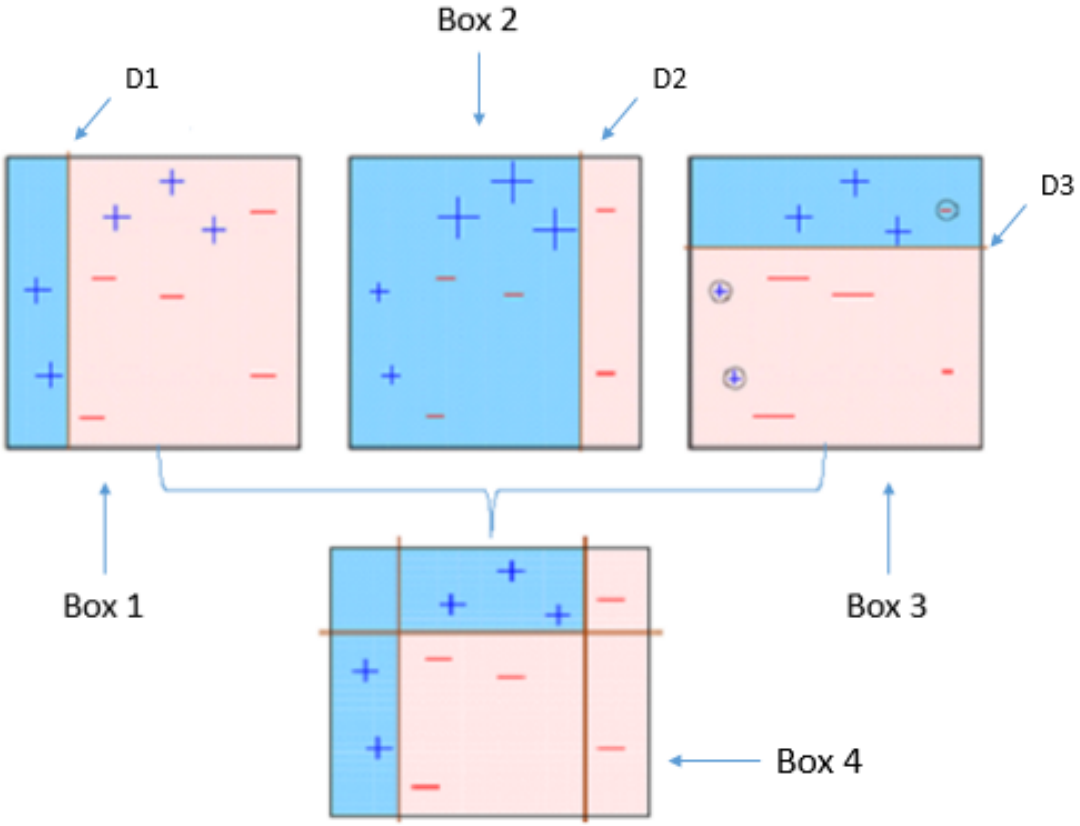
- Produce a distribution of simple ML models on subsets of the original data
- Combine the distribution into one “aggregated” model
- Train

  1. Pick k features from the dataset with m features ($k \ll m$, e.g. $\sqrt{m}$)
  2. Among k features, pick the one that maximizes information gain (or gain ratio) and use it as the node to further split the data
  3. Repeat step 2 to derive individual decision tree
  4. Repeat step 1 - 3 to generate n number of trees

Algorithms: Random Forest (2)

- Predict
  1. Apply each individual tree from the n random trees to the test data point and derive the outcomes and store the n predicted outcome
  2. Calculate the weight for each predicted outcome
  3. Consider the outcome with the highest weight as the final classication/prediction for the test data point

Adaptive Boosting (AdaBoosting) Illustration



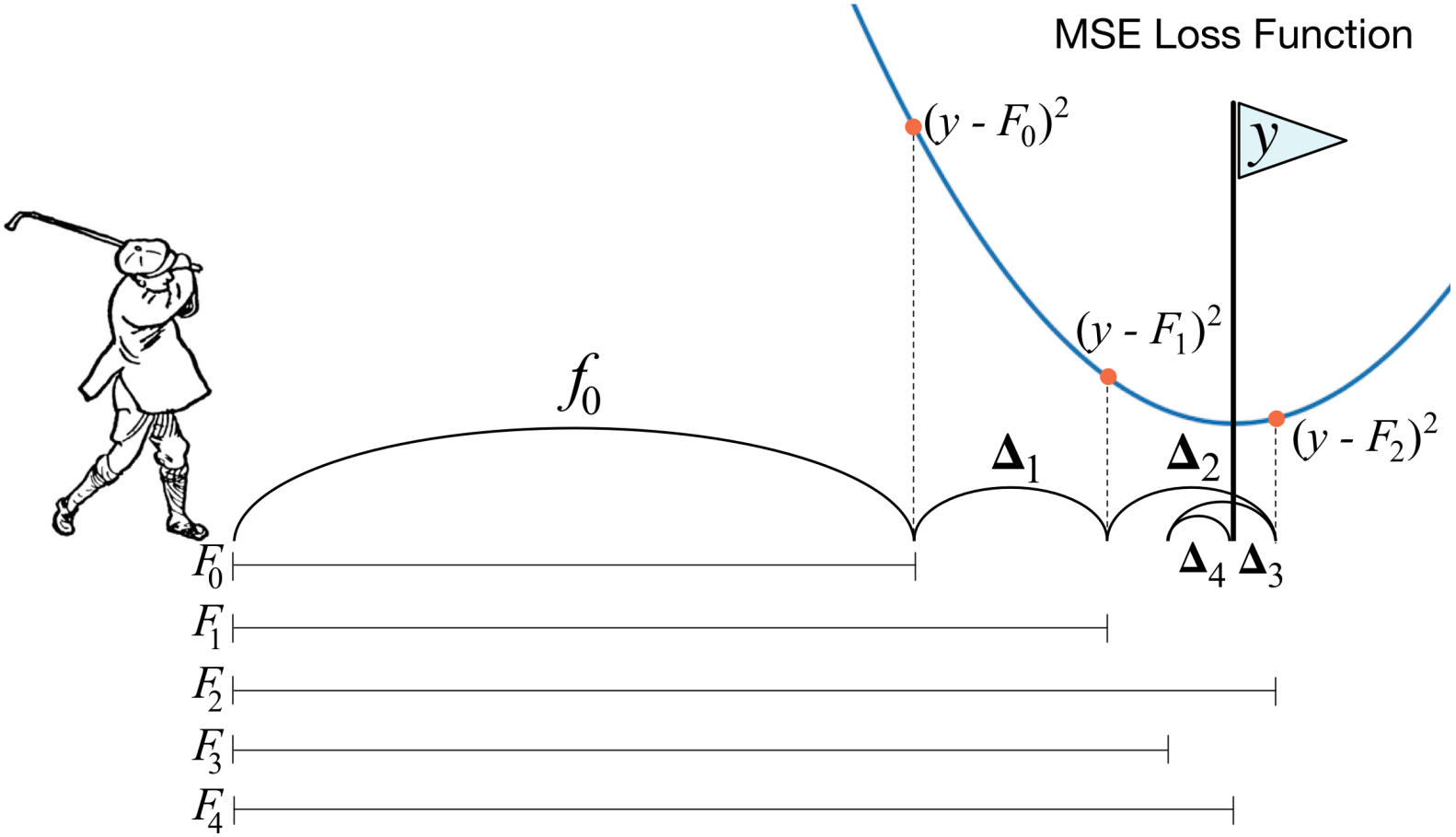
Gradient Boosting Regressor Algorithm

Algorithm: *l2boost*(X,$\mathbf{y}$,M,η) **returns** model F_M
Let $F_0(X) = \frac{1}{N}\sum_{i=1}^{N} y_i$, mean of target $\mathbf{y}$ across all observations
for $m = 1$ **to** M **do**
  Let $\mathbf{r}_{m-1} = \mathbf{y} - F_{m-1}(X)$ be the residual direction vector
  Train regression tree Δ_m on $\mathbf{r}_{m-1}$, minimizing squared error
  $F_m(X) = F_{m-1}(X) + \eta\Delta_m(X)$
end
return F_M

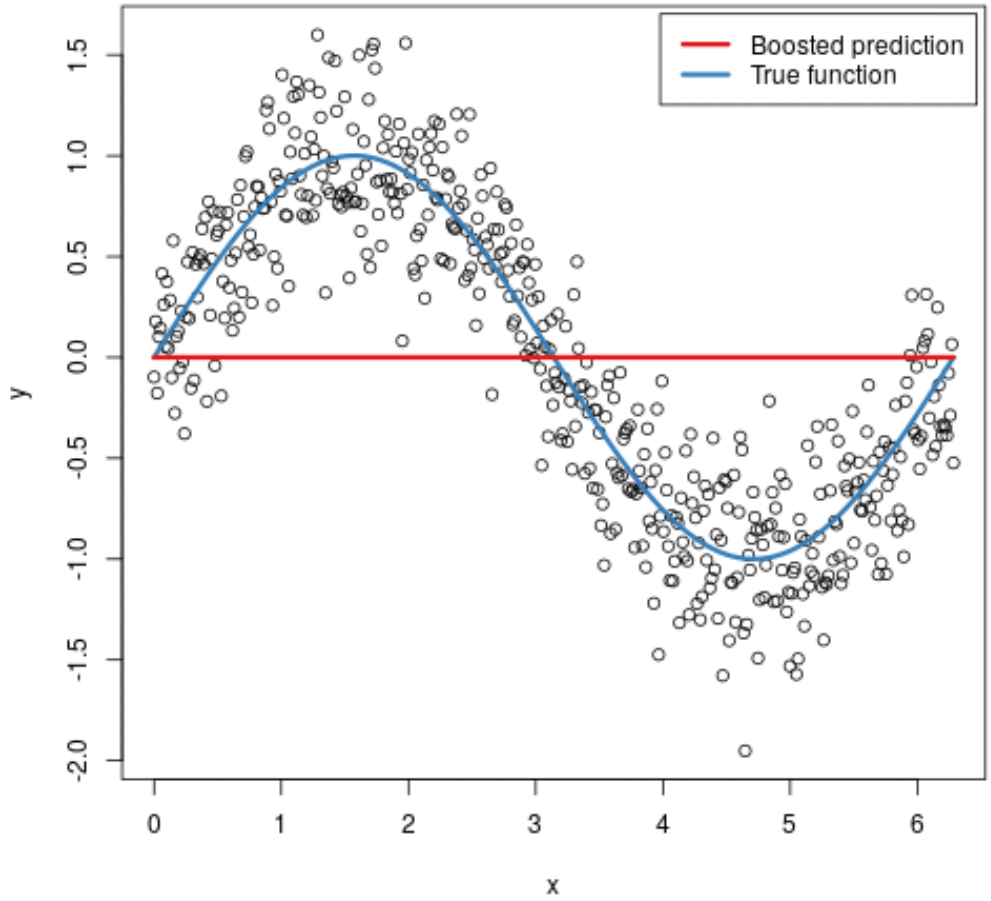
GBM/XGB

- Gradient Boosting Machine (GBM)
  - Additive model
  - Boost performance of weak learners to strong learners additively and sequentially
  - Use gradient descent to construct new base learners that minimize loss function
  - Gradient descent is used in the function space, instead of in the parameter space (e.g. backpropagation in neural network)
- Extreme Gradient Boosting (XGB) improves upon GBM
  - Faster using parallel computing
  - Regularization (hyperparameters such as α, λ), pruning (γ), cross validation enabled, etc.

Analogy of GBM



Visualize GBM



Ensemble Model Hyper-Parameters

- tree-related parameters: maxdepth/max_depth, minsplit/min_samples_split, minbucket/min_samples_leaf, etc.
- mtry/max_features: optional integer for number of features to randomly select at each split, such as $\sqrt{n}$ and n is the number of features, used for random forest or GBM.
- ntree/n_estimators: number of classifiers
- shrinkage/learning_rate (η): GBM/XGB, gradient descent algorithm
- objective function/loss: classification vs. regression
- subsample, etc.

Property of the Algorithms: Pros and Cons

- Pros
  - Perform well on a wide variety of tasks
  - Some ensemble methods such as random forest are robust to overfitting
- Cons
  - More hyper-parameters to tune
  - Interpretability issue
  - Might be sensitive to outliers/noises and overfitting: boosting
  - Higher computational cost

Bagging: Training

```
model_bag <- train(diabetes ~ ., data = diabetes_train1, method = "treebag")
model_bag
```

```
## Bagged CART
##
## 538 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 538, 538, 538, 538, 538, 538, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.7369069  0.4135043
```

Bagging: Prediction and Performance Evaluation

```
predict_bag <- predict(model_bag, newdata = diabetes_test1)
confusionMatrix(predict_bag, diabetes_test1$diabetes)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg 128  30
##        pos  22  50
##
##                Accuracy : 0.7739
##                  95% CI : (0.7143, 0.8263)
##     No Information Rate : 0.6522
##     P-Value [Acc > NIR] : 4.208e-05
##
##                   Kappa : 0.4898
##  Mcnemar's Test P-Value : 0.3317
##
##             Sensitivity : 0.8533
##             Specificity : 0.6250
##          Pos Pred Value : 0.8101
##          Neg Pred Value : 0.6944
##              Prevalence : 0.6522
```

Random Forest

```
model_rf <- train(diabetes ~ ., data = diabetes_train1, method = "rf")
model_rf
```

```
## Random Forest
##
## 538 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 538, 538, 538, 538, 538, 538, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##   2     0.7624434  0.4640791
##   5     0.7529308  0.4486494
##   8     0.7505543  0.4436233
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 2.
```

Gradient Boosting Machine

```
model_gbm <- train(diabetes ~ ., data = diabetes_train1, method = "gbm")
```

```
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1        1.1909             nan     0.1000    0.0159
##      2        1.1682             nan     0.1000    0.0116
##      3        1.1445             nan     0.1000    0.0102
##      4        1.1208             nan     0.1000    0.0095
##      5        1.1006             nan     0.1000    0.0085
##      6        1.0832             nan     0.1000    0.0069
##      7        1.0691             nan     0.1000    0.0065
##      8        1.0581             nan     0.1000    0.0044
##      9        1.0433             nan     0.1000    0.0059
##     10        1.0319             nan     0.1000    0.0049
##     20        0.9391             nan     0.1000    0.0007
##     40        0.8469             nan     0.1000    0.0010
##     60        0.8044             nan     0.1000    0.0000
##     80        0.7735             nan     0.1000    0.0002
##    100        0.7475             nan     0.1000   -0.0010
##    120        0.7305             nan     0.1000   -0.0017
##    140        0.7150             nan     0.1000   -0.0007
##    150        0.7078             nan     0.1000   -0.0008
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
```

Gradient Boosting Machine

```
model_xgb <- train(diabetes ~ ., data = diabetes_train1, method = "xgbTree")
model_xgb
```

```
## eXtreme Gradient Boosting
##
## 538 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 538, 538, 538, 538, 538, 538, ...
## Resampling results across tuning parameters:
##
##   eta  max_depth  colsample_bytree  subsample  nrounds  Accuracy
##   0.3  1          0.6               0.50        50      0.7543059
##   0.3  1          0.6               0.50       100      0.7514498
##   0.3  1          0.6               0.50       150      0.7401059
##   0.3  1          0.6               0.75        50      0.7629041
##   0.3  1          0.6               0.75       100      0.7596995
##   0.3  1          0.6               0.75       150      0.7528011
##   0.3  1          0.6               1.00        50      0.7609537
##   0.3  1          0.6               1.00       100      0.7582121
```

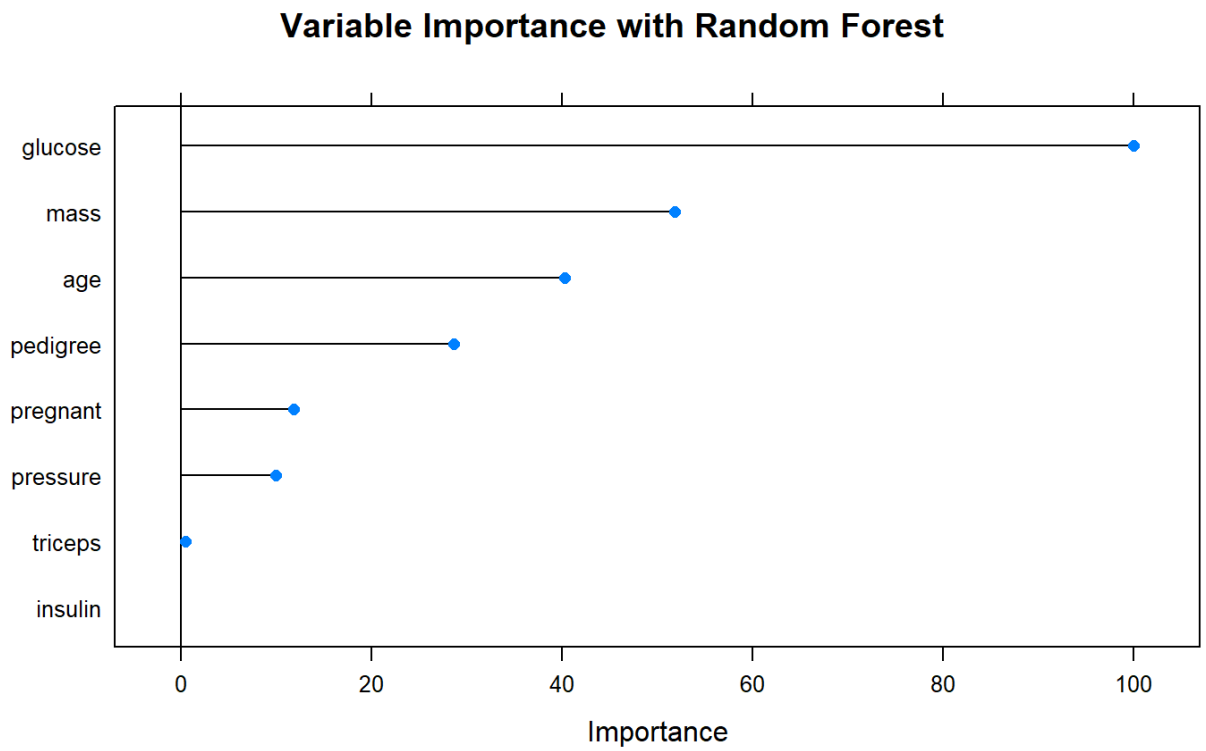
Check the Importance of Attributes

```
varimp_rf <- varImp(model_rf)
varimp_rf
```

```
## rf variable importance
##
##           Overall
## glucose  100.0000
## mass      51.8062
## age       40.3086
## pedigree  28.6711
## pregnant  11.7756
## pressure   9.9492
## triceps    0.4362
## insulin    0.0000
```

Visualize the Attributes' Importance

```
plot(varimp_rf, main = "Variable Importance with Random Forest")
```



Compare the Performance of Multiple Algorithms

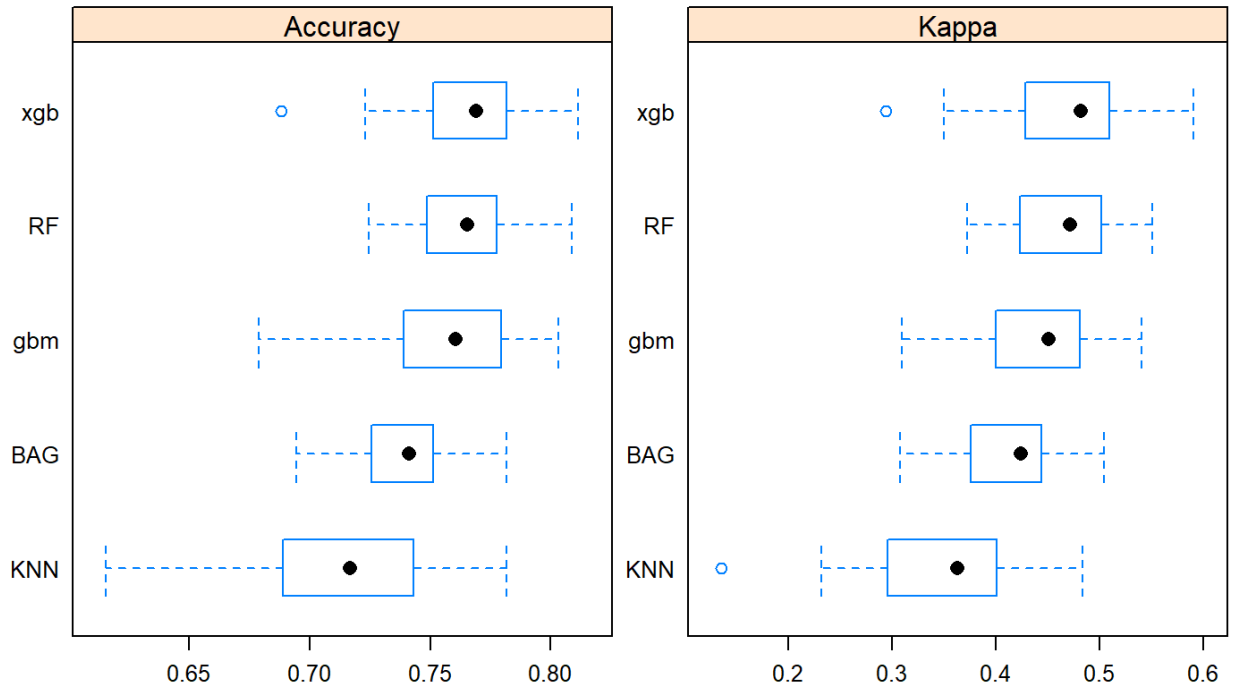
```
model_comparison <- resamples(list(RF = model_rf, BAG = model_bag, KNN = model_knn,
                                   gbm = model_gbm, xgb = model_xgb))
summary(model_comparison)
```

```
##
## Call:
## summary.resamples(object = model_comparison)
##
## Models: RF, BAG, KNN, gbm, xgb
## Number of resamples: 25
##
## Accuracy
##          Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## RF  0.7247191 0.7487179 0.7653061 0.7624434 0.7777778 0.8088235    0
## BAG 0.6945813 0.7258065 0.7411168 0.7369069 0.7512195 0.7817259    0
## KNN 0.6153846 0.6887755 0.7168950 0.7121062 0.7433155 0.7817259    0
## gbm 0.6787565 0.7389163 0.7606383 0.7581561 0.7794872 0.8031088    0
## xgb 0.6881188 0.7512438 0.7692308 0.7638681 0.7817259 0.8115942    0
##
## Kappa
##          Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## RF  0.3724277 0.4234770 0.4718310 0.4640791 0.5022061 0.5508130    0
## BAG 0.3077769 0.3760277 0.4247151 0.4135043 0.4444444 0.5044170    0
```

Graphically Compare Performance

```
scales <- list(x = list(relation = "free"),
               y = list(relation = "free"))

bwplot(model_comparison, scales = scales)
```



Load in Necessary Python packages

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, ShuffleSplit, cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
from scipy.spatial.distance import cdist
from sklearn.pipeline import make_pipeline
import matplotlib.pyplot as plt
```

Data Preparation and KNN modeling

```
diabetes = pd.read_csv("https://datahub.io/machine-learning/diabetes/r/diabetes.csv")
X = diabetes.drop('class', axis=1)
y = diabetes['class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=16)
classifier = KNeighborsClassifier()
classifier.fit(X_train, y_train)
```

```
## KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
##                      metric_params=None, n_jobs=None, n_neighbors=5, p=2,
##                      weights='uniform')
```

```
y_pred = classifier.predict(X_test)
print(f"Accuracy: {round(metrics.accuracy_score(y_test, y_pred)*100, 2)}%")
```

```
## Accuracy: 72.29%
```

```
df_confusion = pd.crosstab(y_test, y_pred)
df_confusion
```

```
## col_0           tested_negative  tested_positive
## class
```

Preprocess Data: Standardization/Normalization

```
import warnings
warnings.filterwarnings("ignore")

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)


## StandardScaler(copy=True, with_mean=True, with_std=True)


X_train_std = scaler.transform(X_train)
X_test_std = scaler.transform(X_test)
pd.DataFrame(X_train_std).mean()


## 0   -2.439596e-17
## 1    1.868979e-16
## 2    7.153392e-17
## 3    5.933594e-17
## 4    1.281822e-17
## 5   -2.456136e-16
## 6    8.042398e-17
## 7    4.507051e-17
## dtype: float64
```

KNN Performance Evaluation

```
classifier_std = KNeighborsClassifier()
classifier_std.fit(X_train_std, y_train)
```

```
## KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
##                      metric_params=None, n_jobs=None, n_neighbors=5, p=2,
##                      weights='uniform')
```

```
y_pred_std = classifier_std.predict(X_test_std)
pd.Series(y_pred_std).value_counts()
```

```
## tested_negative    165
## tested_positive     66
## dtype: int64
```

```
value, count = np.unique(y_pred_std, return_counts=True)
pd.DataFrame({"value": value, "count": count})
```

```
##              value  count
## 0  tested_negative    165
## 1  tested_positive     66
```

Use Pipeline to Streamline the Analysis

```
knn_pipe = make_pipeline(StandardScaler(), KNeighborsClassifier())
knn_pipe.fit(X_train, y_train)
```

```
## Pipeline(memory=None,
##          steps=[('standardscaler',
##                  StandardScaler(copy=True, with_mean=True, with_std=True)),
##                 ('kneighborsclassifier',
##                  KNeighborsClassifier(algorithm='auto', leaf_size=30,
##                                       metric='minkowski', metric_params=None,
##                                       n_jobs=None, n_neighbors=5, p=2,
##                                       weights='uniform'))],
##          verbose=False)
```

```
pipe_pred = knn_pipe.predict(X_test)
pd.Series(pipe_pred).value_counts()
```

```
## tested_negative    165
## tested_positive     66
## dtype: int64
```

Get Repeated Hold Out Accurary of Model

```
cv = ShuffleSplit(n_splits=100, test_size=0.3, random_state=16)
from sklearn.model_selection import KFold
cv = KFold(n_splits=10, shuffle=True, random_state=16)
cross_val_score(knn_pipe, X_train, y_train, cv=cv).mean()
```

```
## 0.7022361984626136
```

Sensitivity Analysis

```
from matplotlib.legend_handler import HandlerLine2D
neighbors = list(range(1, 30))
train_results = []
test_results = []
for n in neighbors:
  model = KNeighborsClassifier(n_neighbors=n)
  model.fit(X_train_std, y_train)
  train_pred = model.predict(X_train_std)
  acc = cross_val_score(model, X_train_std, y_train, cv=cv).mean()*100
  train_results.append(acc)
  y_pred = model.predict(X_test_std)
  acc_test = round(metrics.accuracy_score(y_test, y_pred)*100, 2)
  test_results.append(acc_test)
```

```
## KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
##                      metric_params=None, n_jobs=None, n_neighbors=1, p=2,
##                      weights='uniform')
## KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
##                      metric_params=None, n_jobs=None, n_neighbors=2, p=2,
##                      weights='uniform')
## KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
##                      metric_params=None, n_jobs=None, n_neighbors=3, p=2,
##                      weights='uniform')
```

Ensemble Learning - Bagging

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

cart = DecisionTreeClassifier()
num_trees = 100
model = BaggingClassifier(base_estimator=cart, n_estimators=num_trees, random_state=16)
results = cross_val_score(model, X_train_std, y_train, cv=cv)
print(f"Accuracy: {round(results.mean()*100, 2)}%")


## Accuracy: 71.73%
```

Random Forest

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=num_trees, max_features=5, random_state=16)
results = cross_val_score(model, X_train_std, y_train, cv=cv)
print(f"Accuracy: {round(results.mean()*100, 2)}%")
```

```
## Accuracy: 73.59%
```

```
model
```

```
## RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
##                        max_depth=None, max_features=5, max_leaf_nodes=None,
##                        min_impurity_decrease=0.0, min_impurity_split=None,
##                        min_samples_leaf=1, min_samples_split=2,
##                        min_weight_fraction_leaf=0.0, n_estimators=100,
##                        n_jobs=None, oob_score=False, random_state=16, verbose=0,
##                        warm_start=False)
```

Feature Importance

```
model.fit(X_train_std, y_train)
```

```
## RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
##                        max_depth=None, max_features=5, max_leaf_nodes=None,
##                        min_impurity_decrease=0.0, min_impurity_split=None,
##                        min_samples_leaf=1, min_samples_split=2,
##                        min_weight_fraction_leaf=0.0, n_estimators=100,
##                        n_jobs=None, oob_score=False, random_state=16, verbose=0,
##                        warm_start=False)
```

```
feature_imp = pd.DataFrame(model.feature_importances_, index=X_train.columns,
columns=['importance']).sort_values('importance', ascending=False)
feature_imp
```

```
##       importance
## plas    0.299555
## mass    0.192756
## pedi    0.133711
## age     0.105525
## pres    0.083997
## preg    0.067676
```

AdaBoosting

```
from sklearn.ensemble import AdaBoostClassifier
model = AdaBoostClassifier(n_estimators=num_trees, random_state=16)
results = cross_val_score(model, X_train_std, y_train, cv=cv)
print(f"Accuracy: {round(results.mean()*100, 2)}%")
```

```
## Accuracy: 72.08%
```

```
model
```

```
## AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None, learning_rate=1.0,
##                    n_estimators=100, random_state=16)
```

GBM

```
from sklearn.ensemble import GradientBoostingClassifier as gbm
model = gbm(n_estimators=num_trees, random_state=16)
results = cross_val_score(model, X_train_std, y_train, cv=cv)
print(f"Accuracy for GBM: {round(results.mean()*100, 2)}%")
```

```
## Accuracy for GBM: 72.47%
```

```
model
```

```
## GradientBoostingClassifier(criterion='friedman_mse', init=None,
##                            learning_rate=0.1, loss='deviance', max_depth=3,
##                            max_features=None, max_leaf_nodes=None,
##                            min_impurity_decrease=0.0, min_impurity_split=None,
##                            min_samples_leaf=1, min_samples_split=2,
##                            min_weight_fraction_leaf=0.0, n_estimators=100,
##                            n_iter_no_change=None, presort='auto',
##                            random_state=16, subsample=1.0, tol=0.0001,
##                            validation_fraction=0.1, verbose=0,
##                            warm_start=False)
```

Model Tuning

```
from sklearn.model_selection import GridSearchCV
param_grid = {'learning_rate': np.arange(0.02, 0.1, 0.02),
              'n_estimators': range(60, 160, 50),
              'max_depth': range(2, 5)}
clf = GridSearchCV(gbm(), param_grid)
clf.fit(X_train_std, y_train)
```

```
## GridSearchCV(cv='warn', error_score='raise-deprecating',
##              estimator=GradientBoostingClassifier(criterion='friedman_mse',
##                                                   init=None, learning_rate=0.1,
##                                                   loss='deviance', max_depth=3,
##                                                   max_features=None,
##                                                   max_leaf_nodes=None,
##                                                   min_impurity_decrease=0.0,
##                                                   min_impurity_split=None,
##                                                   min_samples_leaf=1,
##                                                   min_samples_split=2,
##                                                   min_weight_fraction_leaf=0.0,
##                                                   n_estimators=100,
##                                                   n_iter_no_change=None,
##                                                   presort='auto',
##                                                   random_state=None,
##                                                   subsample=1.0, tol=0.0001,
```

Model Tuning (2)

```
print(f"Accuracy for best GBM: {round(clf.best_score_*100, 2)}%")
```

```
## Accuracy for best GBM: 75.23%
```

```
for key, val in clf.best_params_.items():
  print(f"Best hyperparameter is {key}: {val}")
```

```
## Best hyperparameter is learning_rate: 0.02
## Best hyperparameter is max_depth: 2
## Best hyperparameter is n_estimators: 110
```