

Lab 10 - Trees, Bagging, RF, Boosting, XGBoost

Learning goals

- Perform classification and regression with tree-based methods in R
- Recognize that tree-based methods are capable of capturing non-linearities by splitting multiple times on the same variables
- Compare the performance of classification trees, bagging, random forests, and boosting for predicting heart disease based on the `heart` data.

Lab description

For this lab we will be working with the `heart` dataset that you can download from [here](#)

Deliverables

Questions 1-5 answered, pdf or html output uploaded to Quercus

Setup packages

You should install and load `rpart` (trees), `randomForest` (random forest), `gbm` (gradient boosting) and `xgboost` (extreme gradient boosting).

```
install.packages(c("rpart", "rpart.plot", "randomForest", "gbm", "xgboost"))
```

Load packages and data

```
library(tidyverse)
library(rpart)
library(rpart.plot)
library(randomForest)
library(gbm)
library(xgboost)
library(caret)

heart <- read_csv("https://raw.githubusercontent.com/JSC370/JSC370-2025/refs/heads/main/data/heart/heart.csv")

heart <- heart |>
  mutate(
    AHD = 1 * (AHD == "Yes"),
    ChestPain = factor(ChestPain),
```

```

    Thal = factor(Thal)
  )

head(heart)

```

```

## # A tibble: 6 x 14
##   Age Sex ChestPain RestBP Chol Fbs RestECG MaxHR ExAng Oldpeak Slope
##   <dbl> <dbl> <fct>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1   63   1 typical      145   233   1     2   150     0    2.3     3
## 2   67   1 asymptomatic 160   286   0     2   108     1    1.5     2
## 3   67   1 asymptomatic 120   229   0     2   129     1    2.6     2
## 4   37   1 nonanginal  130   250   0     0   187     0    3.5     3
## 5   41   0 nontypical  130   204   0     2   172     0    1.4     1
## 6   56   1 nontypical  120   236   0     0   178     0    0.8     1
## # i 3 more variables: Ca <dbl>, Thal <fct>, AHD <dbl>

```

Question 1: Trees

- Split the heart data into training and testing (70-30%)

```

set.seed(123)
train_index <- createDataPartition(heart$AHD, p = 0.7, list = FALSE)
heart_train <- heart[train_index, ]
heart_test  <- heart[-train_index, ]

```

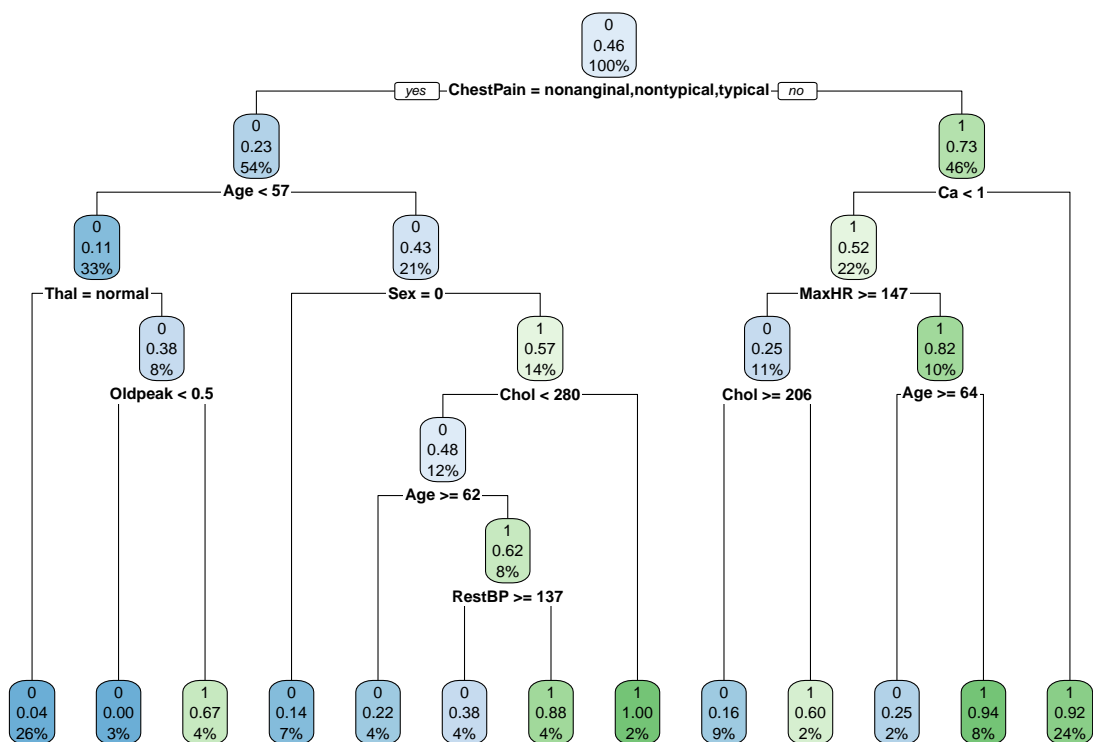
- Fit a classification tree using rpart, plot the full tree. We are trying to predict AHD. Set minsplit = 10, minbucket = 3, and do 10 cross validations.

```

heart_tree <- rpart(AHD ~ .,
  data = heart_train,
  method = "class",
  control = rpart.control(minsplit = 10,
    minbucket = 3,
    cp = 0.01,
    xval = 10))

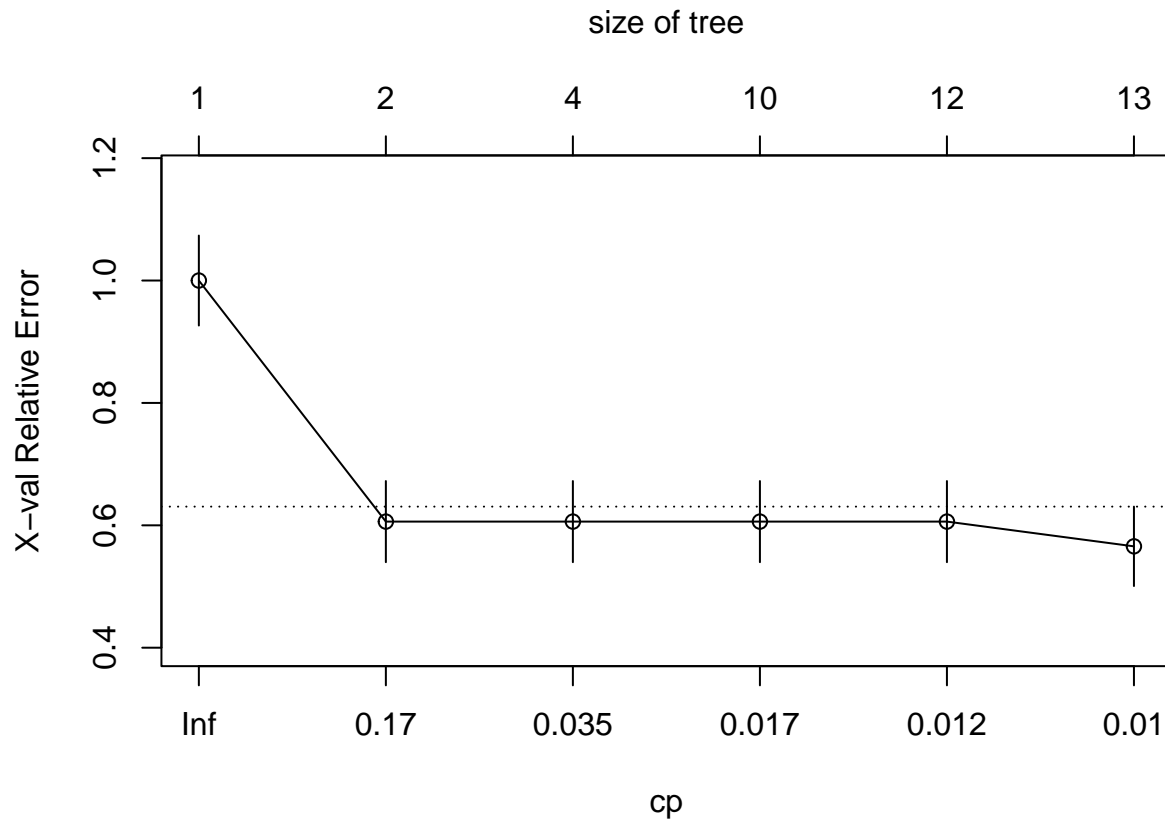
rpart.plot(heart_tree)

```



- Plot the complexity parameter table for an rpart fit and find the optimal cp

```
plotcp(heart_tree)
```



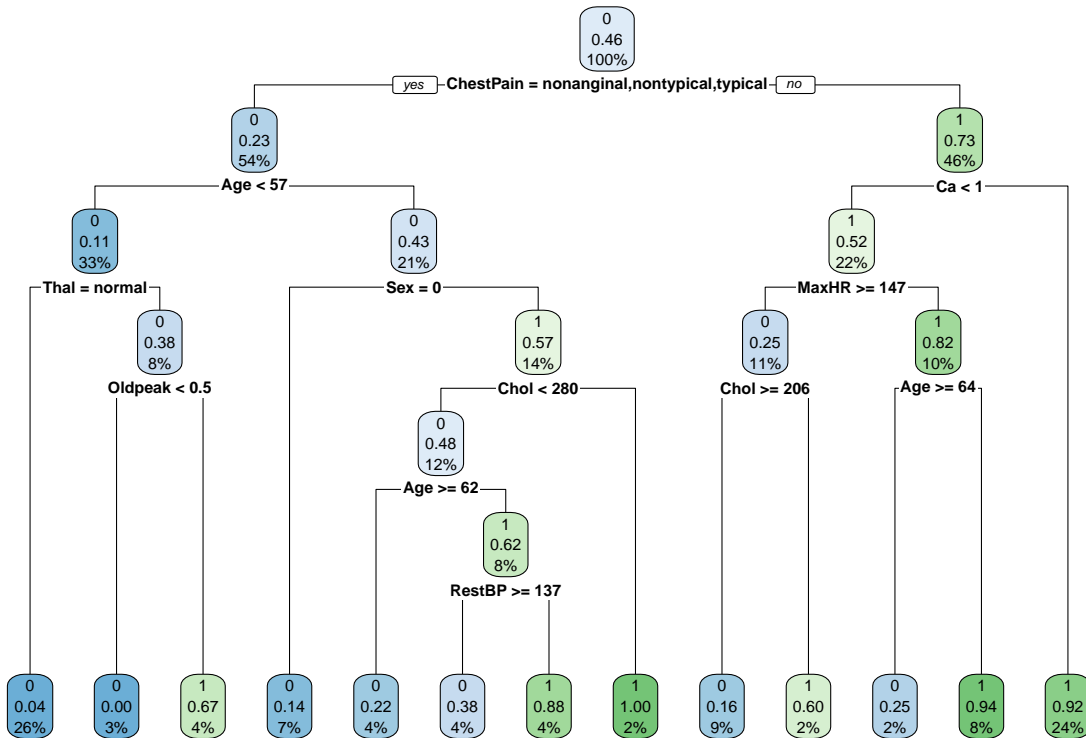
```
printcp(heart_tree)
```

```
##
## Classification tree:
## rpart(formula = AHD ~ ., data = heart_train, method = "class",
##       control = rpart.control(minsplit = 10, minbucket = 3, cp = 0.01,
##       xval = 10))
##
## Variables actually used in tree construction:
## [1] Age      Ca      ChestPain Chol      MaxHR      Oldpeak  RestBP
## [8] Sex      Thal
##
## Root node error: 99/213 = 0.46479
##
## n= 213
##
##      CP nsplit rel error  xerror    xstd
## 1 0.464646      0  1.00000 1.00000 0.073527
## 2 0.060606      1  0.53535 0.60606 0.066313
## 3 0.020202      3  0.41414 0.60606 0.066313
## 4 0.015152      9  0.28283 0.60606 0.066313
## 5 0.010101     11  0.25253 0.60606 0.066313
## 6 0.010000     12  0.24242 0.56566 0.064896
```

```
optimalcp = heart_tree$cptable[which.min(heart_tree$cptable[, "xerror"]), "CP"]
```

- Prune the tree

```
heart_tree_prune <- prune(heart_tree, cp = optimalcp)
rpart.plot(heart_tree_prune)
```



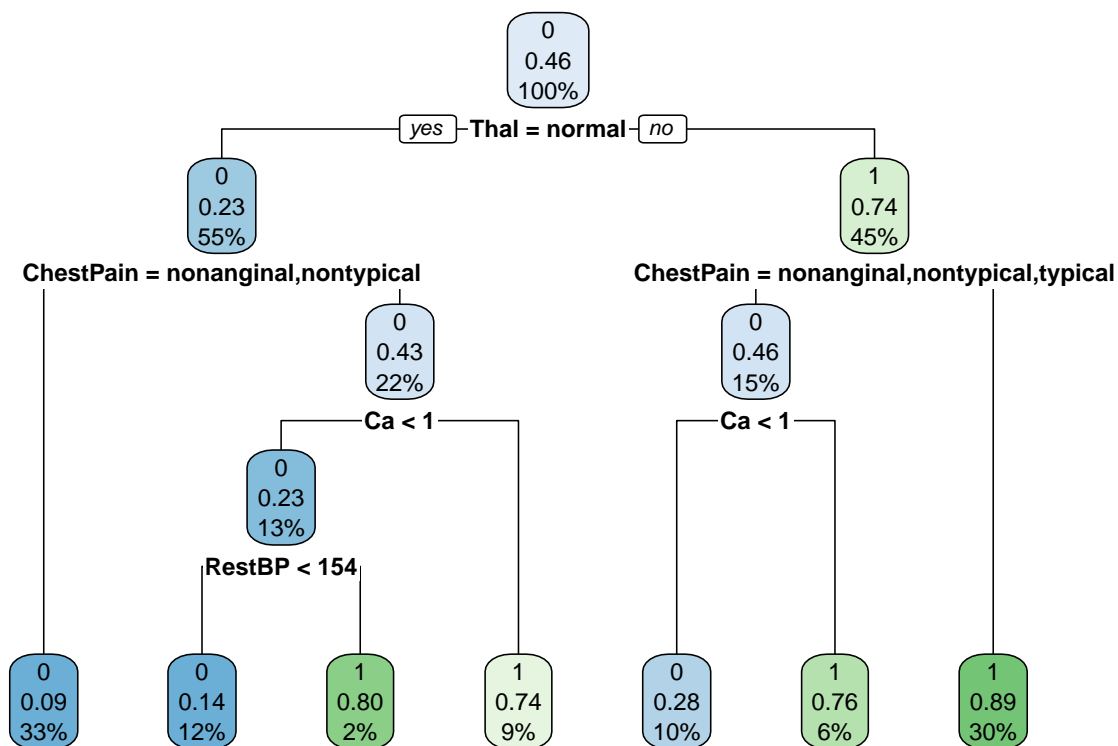
- Compute the test misclassification error

```
heart_pred <- predict(heart_tree_prune, heart_test)
misclass_error <- mean(heart_pred != heart_test$AHD)
misclass_error
```

```
## [1] 0.9833333
```

- Fit the tree with the optimal complexity parameter to the full data (training + testing)

```
heart_tree <- rpart(AHD ~ ., data = heart, method = "class",
                    control = rpart.control(minsplit = 10, minbucket = 3, cp = optimalcp, xval = 1),
                    rpart.plot(heart_tree))
```



- Find the Out of Bag (OOB) error for tree

```
oob_error <- mean(heart_pred != heart_test$AHD)
oob_error
```

```
## [1] 0.9833333
```

Question 2: Bagging, Random Forest

- Compare the performance of classification trees (above), bagging, random forests for predicting heart disease based on the `heart` data.
- Use the training and testing sets from above. Train each of the models on the training data and extract the cross-validation (or out-of-bag error for bagging and Random forest).
- For bagging use `randomForest` with `mtry` equal to the number of features (all other parameters at their default values). Generate the variable importance plot using `varImpPlot` and extract variable importance from the `randomForest` fitted object using the `importance` function.

```
heart_bag <- randomForest(AHD ~ .,
  data = heart_train,
  mtry = ncol(heart_train) - 1,
```

```

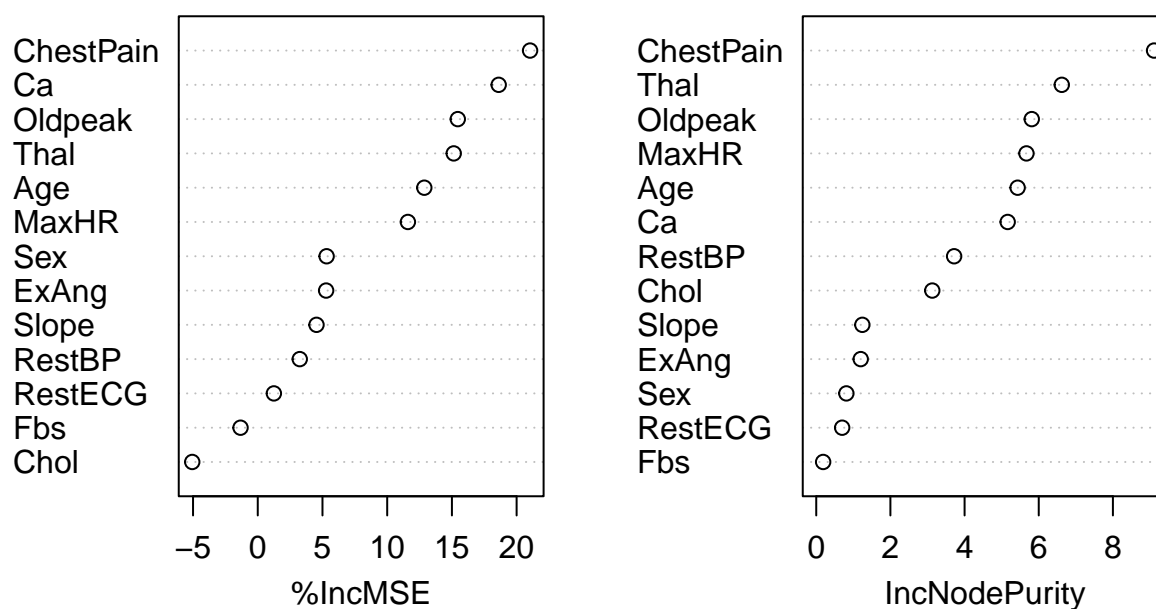
importance = TRUE,
na.action = na.omit)
# oob error rate
heart_bag_oob <- heart_bag$mse[length(heart_bag$mse)]
heart_bag_oob

```

```
## [1] 0.1523859
```

```
varImpPlot(heart_bag)
```

heart_bag



```
importance(heart_bag)
```

```
##           %IncMSE IncNodePurity
## Age      12.867404    5.4307675
## Sex       5.319511    0.8120348
## ChestPain 21.039907    9.1106667
## RestBP    3.244514    3.7183816
## Chol     -5.060963    3.1289056
## Fbs      -1.331278    0.1857550
## RestECG   1.244870    0.6986699
## MaxHR     11.599316    5.6685198
## ExAng      5.283321    1.1992190
## Oldpeak   15.453784    5.8115107
## Slope     4.538594    1.2431167

```

```
## Ca      18.613902    5.1624968
## Thal    15.141788    6.6214136
```

- For random forests use `randomForest` with the default parameters. Generate the variable importance plot using `varImpPlot` and extract variable importance from the `randomForest` fitted object using the `importance` function.

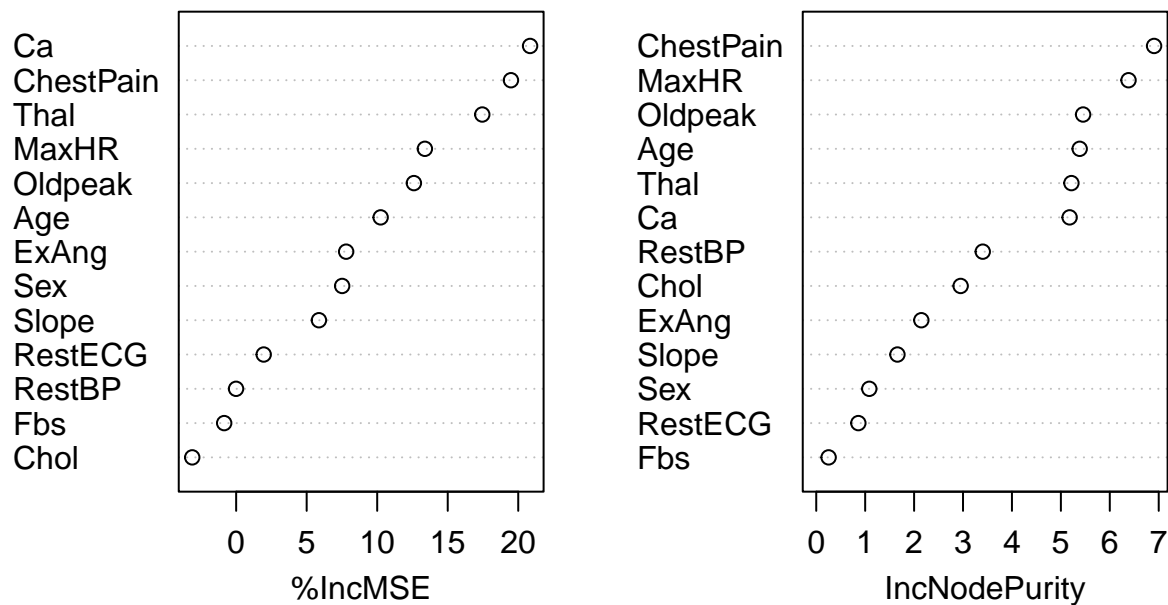
```
heart_rf <- randomForest(AHD ~ .,
  data = heart_train,
  importance = TRUE,
  na.action = na.omit)

#oob error rate
heart_rf_oob <- heart_rf$mse[length(heart_rf$mse)]
heart_rf_oob
```

```
## [1] 0.1437152
```

```
varImpPlot(heart_rf)
```

heart_rf



```
importance(heart_rf)
```

```
##           %IncMSE IncNodePurity
## Age      10.244014583    5.3845988
## Sex       7.514733740    1.0839375
```

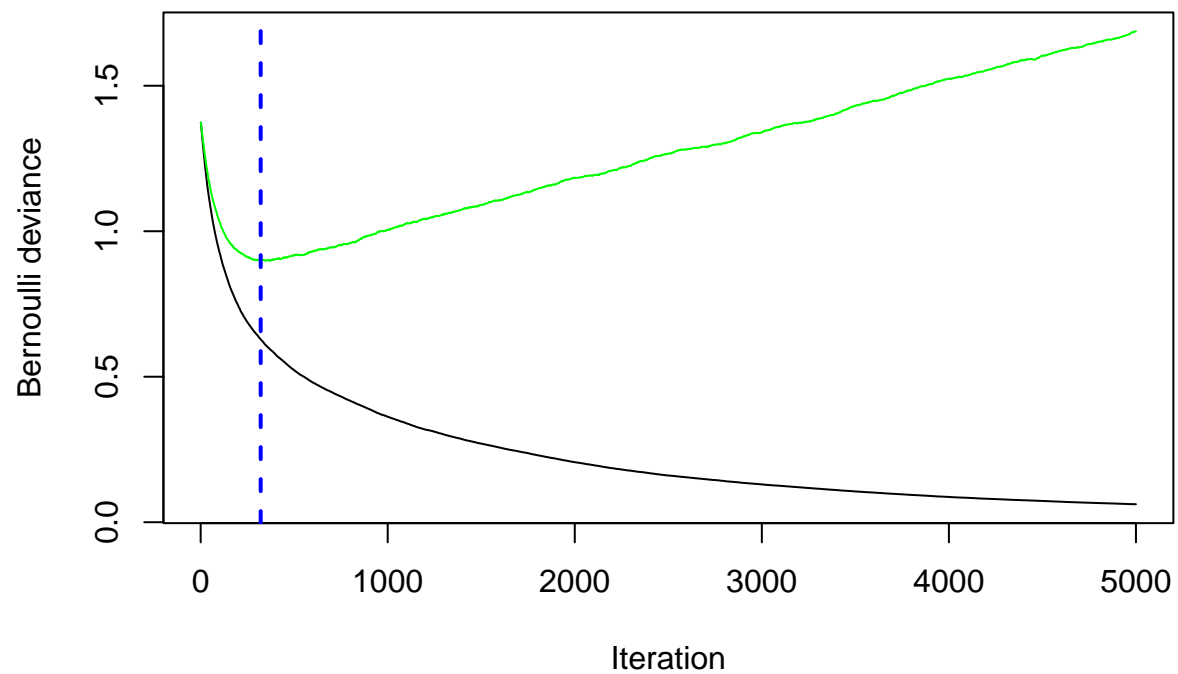

## ChestPain	19.483034641	6.9052084
## RestBP	-0.006897078	3.4032516
## Chol	-3.112866570	2.9511046
## Fbs	-0.849095705	0.2528145
## RestECG	1.952920256	0.8605913
## MaxHR	13.380304038	6.3829931
## ExAng	7.798714969	2.1479200
## Oldpeak	12.601595424	5.4543553
## Slope	5.869284824	1.6594194
## Ca	20.839280433	5.1788306
## Thal	17.449576408	5.2155581

Question 3: Boosting

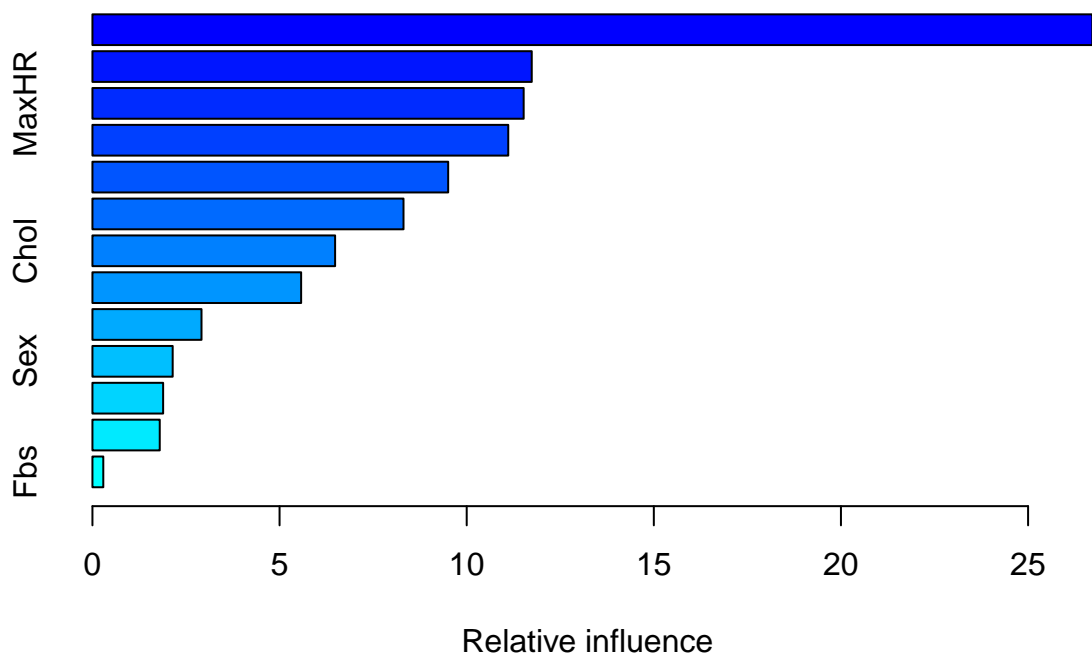
- For boosting use `gbm` with `cv.folds=5` to perform 5-fold cross-validation, and set `class.stratify.cv` to `AHD` (heart disease outcome) so that cross-validation is performed stratifying by `AHD`. Plot the cross-validation error as a function of the boosting iteration/trees (the `$cv.error` component of the object returned by `gbm`) and determine whether additional boosting iterations are warranted. If so, run additional iterations with `gbm.more` (use the R help to check its syntax). Choose the optimal number of iterations. Use the `summary.gbm` function to generate the variable importance plot and extract variable importance/influence (`summary.gbm` does both). Generate 1D and 2D marginal plots with `gbm.plot` to assess the effect of the top three variables and their 2-way interactions.

```
heart_boost = gbm(AHD ~ .,
  data = heart_train,
  distribution = "bernoulli",
  n.trees = 5000,
  interaction.depth = 3,
  shrinkage = 0.01,
  cv.folds = 5,
  class.stratify.cv = TRUE,
  n.minobsinnode = 10,
  verbose = FALSE)

best_iter <- gbm.perf(heart_boost, method = "cv")
```

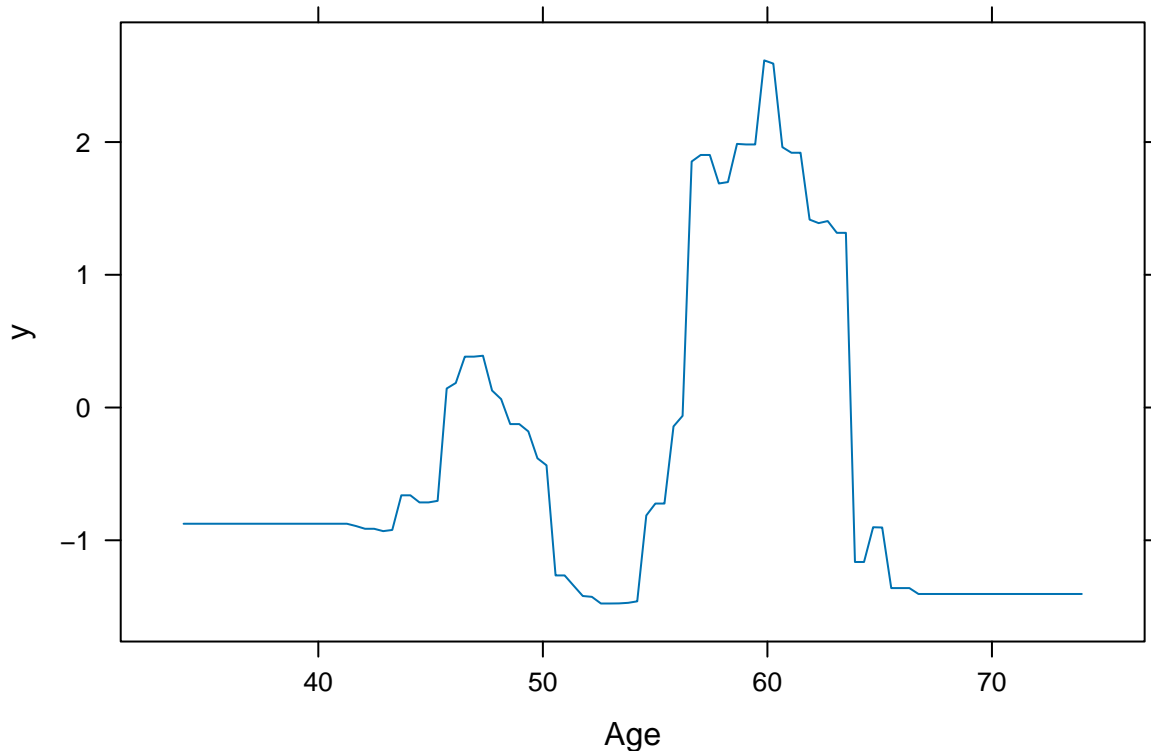


```
heart_boost <- gbm.more(heart_boost, 1000)
summary.gbm(heart_boost)
```



```
##           var      rel.inf
## Thal      Thal 26.7174065
## Age       Age 11.7379682
## MaxHR     MaxHR 11.5220897
## ChestPain ChestPain 11.1103110
## Oldpeak   Oldpeak 9.5051924
## Ca        Ca 8.3115801
## Chol      Chol 6.4842892
## RestBP    RestBP 5.5764347
## ExAng     ExAng 2.9140691
## Sex       Sex 2.1436754
## Slope     Slope 1.8890104
## RestECG   RestECG 1.7982158
## Fbs       Fbs 0.2897575
```

```
plot.gbm(heart_boost, n.plots = 3)
```



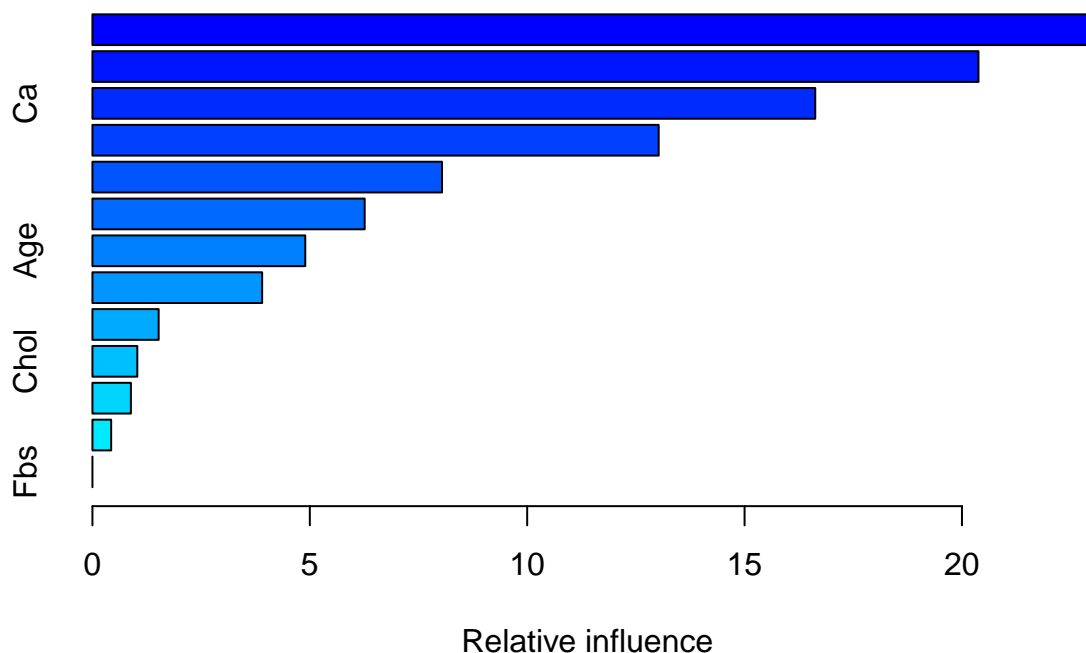
Question 4: Gradient Boosting

Evaluate the effect of critical boosting parameters (number of boosting iterations, shrinkage/learning rate, and tree depth/interaction). In `gbm` the number of iterations is controlled by `n.trees` (default is 100), the shrinkage/learning rate is controlled by `shrinkage` (default is 0.001), and interaction depth by `interaction.depth` (default is 1).

Note, boosting can overfit if the number of trees is too large. The shrinkage parameter controls the rate at which the boosting learns. Very small λ can require using a very large number of trees to achieve good performance. Finally, interaction depth controls the interaction order of the boosted model. A value of 1 implies an additive model, a value of 2 implies a model with up to 2-way interactions, etc. the default is 1.

- Set the seed and train a boosting classification with `gbm` using 10-fold cross-validation (`cv.folds=10`) on the training data with `n.trees = 5000`, `shrinkage = 0.001`, and `interaction.depth = 1`. Plot the cross-validation errors as a function of the boosting iteration and calculate the test MSE.

```
set.seed(301)
heart_boost = gbm(AHD ~ ., data = heart_train, distribution = "bernoulli", n.trees = 5000, interaction.depth = 1)
summary(heart_boost)
```



```
##           var      rel.inf
## ChestPain ChestPain 22.9994834
## Thal      Thal    20.3797007
## Ca        Ca      16.6251597
## Oldpeak   Oldpeak 13.0243176
## MaxHR     MaxHR   8.0406943
## ExAng     ExAng   6.2620816
## Age       Age     4.8953239
## Slope     Slope   3.9035203
## Sex       Sex     1.5228045
## Chol      Chol    1.0318319
## RestBP    RestBP  0.8846967
## RestECG   RestECG 0.4303856
## Fbs       Fbs     0.0000000
```

```
yhat_boost <- predict(heart_boost, heart_test, n.trees = 5000, type = "response")
yhat_boost_class <- ifelse(yhat_boost > 0.5, 1, 0)

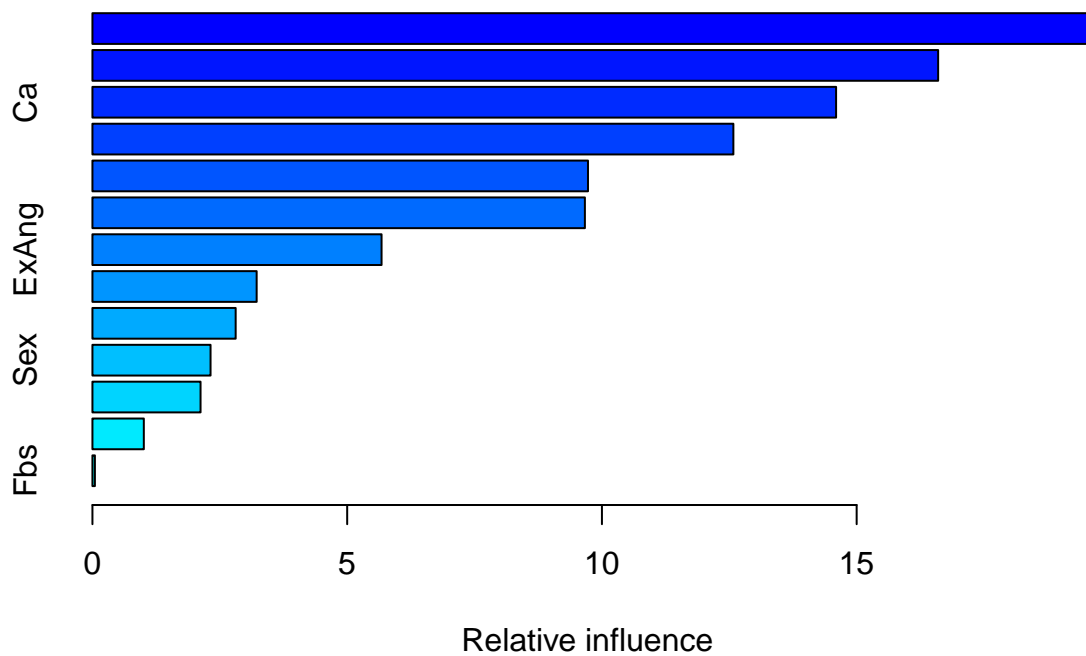
#calculate MSE
mse_boost <- mean((yhat_boost - heart_train$AHD)^2)
mse_boost
```

```
## [1] 0.3703812
```

- Repeat the above using the same seed and `n.trees=5000` with the following 3 additional combination of parameters: a) `shrinkage = 0.001`, `interaction.depth = 2`; b) `shrinkage = 0.01`,

```
interaction.depth = 1; c) shrinkage = 0.01, interaction.depth = 2.
```

```
set.seed(301)
heart_boost = gbm(AHD ~ ., data = heart_train, distribution = "bernoulli", n.trees = 5000, interaction.
summary(heart_boost)
```



```
##          var      rel.inf
## ChestPain ChestPain 19.62625930
## Thal      Thal    16.59930628
## Ca        Ca      14.59642665
## Oldpeak   Oldpeak 12.57994421
## MaxHR     MaxHR   9.72626292
## Age       Age     9.66565392
## ExAng     ExAng   5.67497993
## Slope     Slope   3.22320438
## Chol      Chol    2.81108266
## Sex       Sex     2.31811472
## RestBP    RestBP  2.12202932
## RestECG   RestECG 1.00882566
## Fbs       Fbs     0.04791006
```

```
yhat_boost <- predict(heart_boost, heart_test, n.trees = 5000, type = "response")
yhat_boost_class <- ifelse(yhat_boost > 0.5, 1, 0)
```

```
#calculate MSE
```

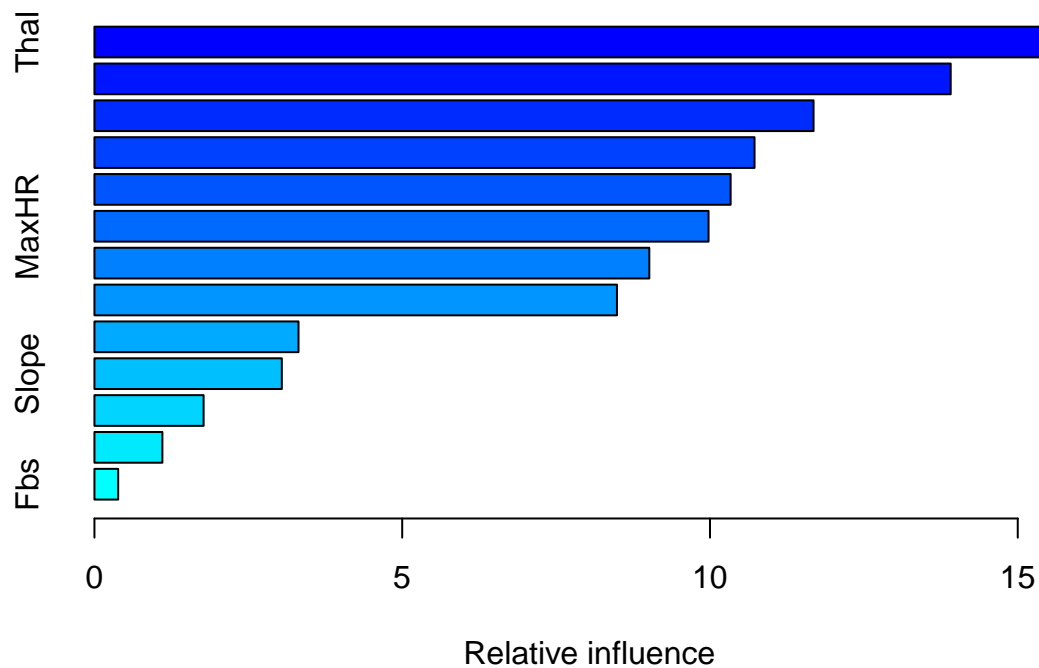
```
mse_boost <- mean((yhat_boost - heart_train$AHD)^2)
```

```
mse_boost
```

```
## [1] 0.3904245
```

```
heart_boost = gbm(AHD ~ ., data = heart_train, distribution = "bernoulli", n.trees = 5000, interaction.p
```

```
summary(heart_boost)
```



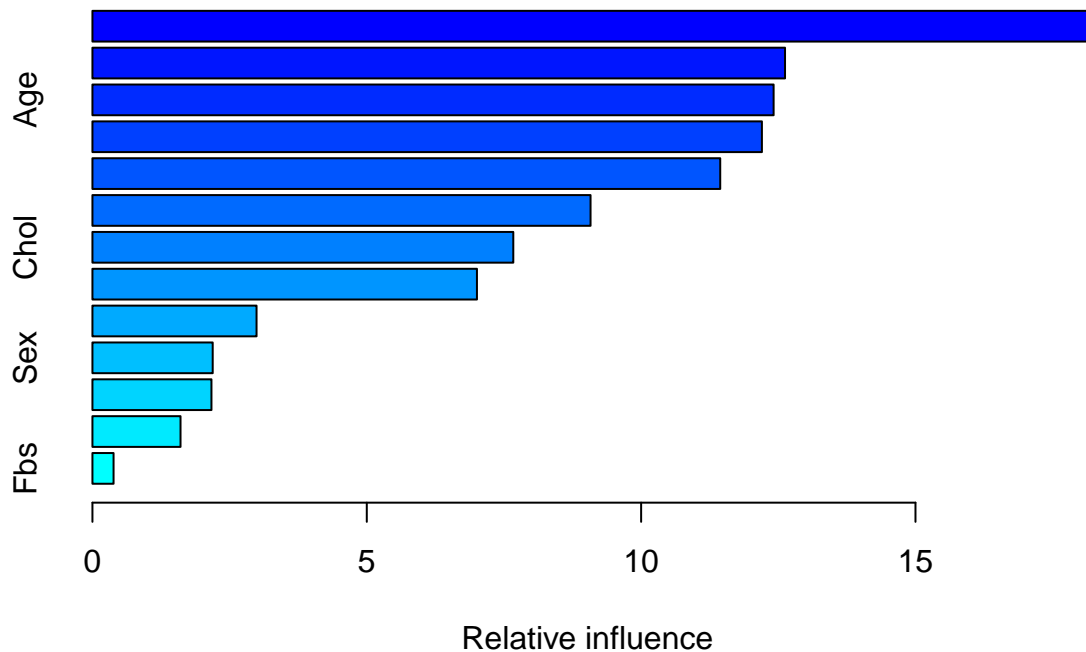
```
##           var    rel.inf
## Thal           Thal 16.2445905
## ChestPain ChestPain 13.9110022
## Oldpeak      Oldpeak 11.6825414
## Age           Age 10.7240062
## Ca            Ca 10.3353034
## MaxHR         MaxHR  9.9766796
## RestBP        RestBP  9.0146959
## Chol          Chol  8.4886526
## ExAng         ExAng  3.3155378
## Slope         Slope  3.0440852
## Sex           Sex  1.7734397
## RestECG       RestECG 1.1039968
## Fbs           Fbs  0.3854686
```

```
yhat_boost <- predict(heart_boost, heart_test, n.trees = 5000, type = "response")
yhat_boost_class <- ifelse(yhat_boost > 0.5, 1, 0)
```

```
#calculate MSE
mse_boost <- mean((yhat_boost - heart_train$AHD)^2)
mse_boost
```

```
## [1] 0.4500244
```

```
heart_boost = gbm(AHD ~ ., data = heart_train, distribution = "bernoulli", n.trees = 5000, interaction.p = 0.01)
summary(heart_boost)
```



```
##           var    rel.inf
## Thal      Thal 18.223537
## MaxHR     MaxHR 12.6221884
## Age       Age 12.4142574
## ChestPain ChestPain 12.2008678
## Oldpeak   Oldpeak 11.4415661
## Ca        Ca 9.0780575
## Chol      Chol 7.6699653
## RestBP    RestBP 7.0074782
## ExAng     ExAng 2.9907216
## Sex       Sex 2.1919262
```



```
## Slope          Slope  2.1692456
## RestECG        RestECG 1.6052804
## Fbs            Fbs   0.3848919
```

```
yhat_boost <- predict(heart_boost, heart_test, n.trees = 5000, type = "response")
yhat_boost_class <- ifelse(yhat_boost > 0.5, 1, 0)

#calculate MSE
mse_boost <- mean((yhat_boost - heart_train$AHD)^2)
mse_boost
```

```
## [1] 0.464779
```

Question 5: Extreme Gradient Boosting

Train a XGBoost model with `xgboost` and perform a grid search for tuning the number of trees and the maximum depth of the tree. Also perform 10-fold cross-validation and determine the variable importance. Finally, compute the test MSE.

Tuning parameters - `max_depth`: tree depth, larger makes model more complex and potentially overfit - `nrounds`: number of boosting iterations - `eta`: learning rate (shrinkage) - `gamma`: minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative the algorithm will be (simpler model) - `min_child_weight`: controls the minimum number of samples in a leaf node before further splitting - `colsample_bytree`: controls the fraction of features (variables) used to build each tree. Default is 1 which is all features

```
train_control = trainControl(method = "cv", number = 10, search = "grid")

tune_grid<- expand.grid(max_depth = c(1, 3, 5, 7),
                      nrounds = (1:10)*50,
                      eta = c(0.01,0.1,0.3),
                      gamma = 0,
                      subsample = 1,
                      min_child_weight = 1,
                      colsample_bytree = 0.6
                      )

heart_xgb<-caret::train(AHD ~ .,
                       data = heart_train,
                       method = "xgbTree",
                       trControl = train_control,
                       tuneGrid = tune_grid,
                       na.action = na.omit)
```

```
## [19:00:30] WARNING: src/c_api/c_api.cc:935: 'ntree_limit' is deprecated, use 'iteration_range' instead
## [19:00:30] WARNING: src/c_api/c_api.cc:935: 'ntree_limit' is deprecated, use 'iteration_range' instead
## [19:00:30] WARNING: src/c_api/c_api.cc:935: 'ntree_limit' is deprecated, use 'iteration_range' instead
## [19:00:30] WARNING: src/c_api/c_api.cc:935: 'ntree_limit' is deprecated, use 'iteration_range' instead
## [19:00:30] WARNING: src/c_api/c_api.cc:935: 'ntree_limit' is deprecated, use 'iteration_range' instead
## [19:00:30] WARNING: src/c_api/c_api.cc:935: 'ntree_limit' is deprecated, use 'iteration_range' instead
## [19:00:30] WARNING: src/c_api/c_api.cc:935: 'ntree_limit' is deprecated, use 'iteration_range' instead
## [19:00:30] WARNING: src/c_api/c_api.cc:935: 'ntree_limit' is deprecated, use 'iteration_range' instead
```

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

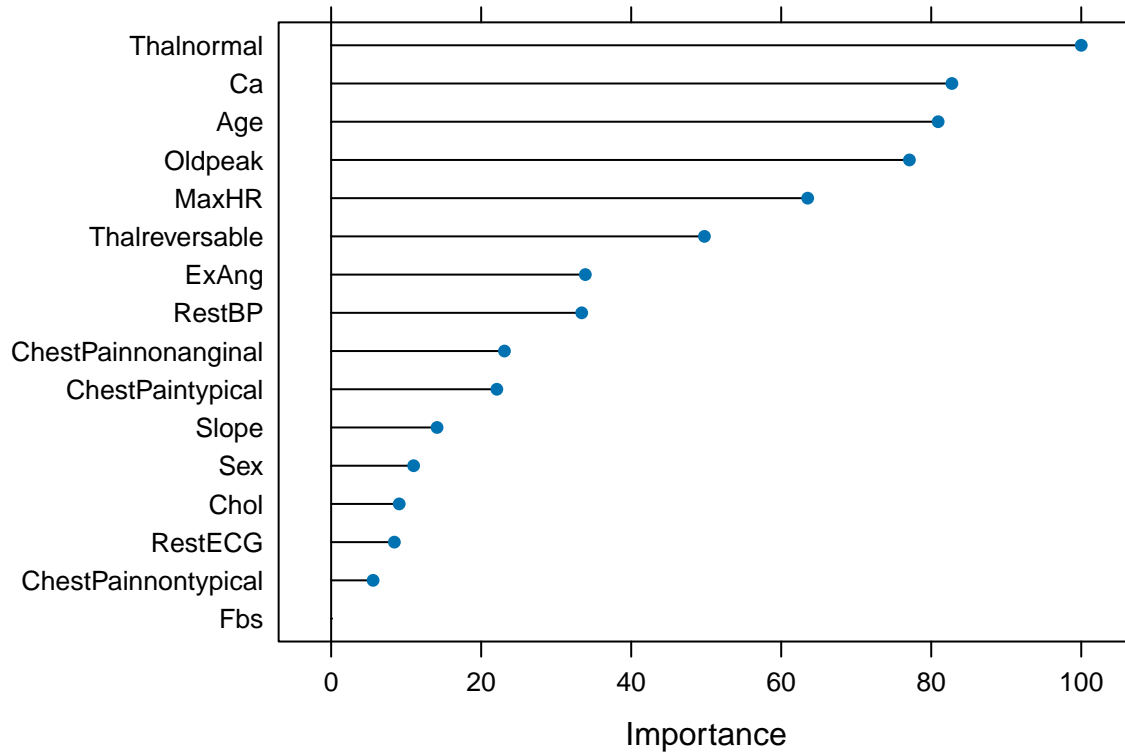
[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



```
xgb_pred <- predict(heart_xgb, heart_test)
xgb_mse <- mean((xgb_pred - heart_test$AHD)^2)
xgb_mse
```

```
## [1] 0.2502881
```

- Compare the the performance of the different models and summarize

```
library(knitr)

model_performance <- data.frame(
  Model = c(
    "Classification Tree",
    "Pruned Tree",
    "Bagging",
    "Random Forest",
    "Boosting (GBM)",
    "XGBoost"
  ),
  Error_MSE = c(
    misclass_error, # From Question 1
    oob_error,      # From Question 1
    heart_bag_oob,  # From Question 2
    heart_rf_oob,   # From Question 2
    mse_boost,      # From Question 4
  )
)
```

```

    xgb_mse      # From Question 5
  )
)

kable(model_performance, col.names = c("Model", "Error / MSE"), caption = "Model Performance Comparison")

```

Table 1: Model Performance Comparison

Model	Error / MSE
Classification Tree	0.9833333
Pruned Tree	0.9833333
Bagging	0.1523859
Random Forest	0.1437152
Boosting (GBM)	0.4647790
XGBoost	0.2502881

The Classification Tree and Pruned Tree performed poorly, both having an error rate of 0.9833, indicating severe overfitting. Bagging (0.1524 error rate) and Random Forest (0.1437 error rate) significantly improved performance, with Random Forest being the best classification model. For boosting models, XGBoost (MSE = 0.2523) outperformed traditional GBM (MSE = 0.3704), making it the best choice for regression-based predictions. Overall, Random Forest is the best for classification, while XGBoost is the most effective boosting model for this dataset.