Heart Attack Analysis & Prediction

Şeyma Günönü

Packages

Before editing the data set, we need to install some packages.

```
#install.packages("caret")
#install.packages("ROCR")
#install.packages("readr")
library(caret)
library(ROCR)
library(readr)
```

Data set

We are using a data set about heart attack prediction from Kaggle. The data set is a csv file, so we can use readr package to import the data set. After that, we can assign the data set as heart_attack.

```
heart_attack <- read_csv("heart.csv")
```

After adding our data set, we can use `str()` function to see details about our data set.

```
str(heart_attack)
```

```
spc_tbl_ [303 x 14] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ age     : num [1:303] 63 37 41 56 57 57 56 44 52 57 ...
 $ sex     : num [1:303] 1 1 0 1 0 1 0 1 1 1 ...
 $ cp      : num [1:303] 3 2 1 1 0 0 1 1 2 2 ...
```

```
 $ trtbps  : num [1:303] 145 130 130 120 120 140 140 120 172 150 ...
 $ chol    : num [1:303] 233 250 204 236 354 192 294 263 199 168 ...
 $ fbs     : num [1:303] 1 0 0 0 0 0 0 0 1 0 ...
 $ restecg : num [1:303] 0 1 0 1 1 1 0 1 1 1 ...
 $ thalachh: num [1:303] 150 187 172 178 163 148 153 173 162 174 ...
 $ exng    : num [1:303] 0 0 0 0 1 0 0 0 0 0 ...
 $ oldpeak : num [1:303] 2.3 3.5 1.4 0.8 0.6 0.4 1.3 0 0.5 1.6 ...
 $ slp     : num [1:303] 0 0 2 2 2 1 1 2 2 2 ...
 $ caa     : num [1:303] 0 0 0 0 0 0 0 0 0 0 ...
 $ thall   : num [1:303] 1 2 2 2 2 1 2 3 3 2 ...
 $ output  : num [1:303] 1 1 1 1 1 1 1 1 1 1 ...
 - attr(*, "spec")=
  .. cols(
  ..   age = col_double(),
  ..   sex = col_double(),
  ..   cp = col_double(),
  ..   trtbps = col_double(),
  ..   chol = col_double(),
  ..   fbs = col_double(),
  ..   restecg = col_double(),
  ..   thalachh = col_double(),
  ..   exng = col_double(),
  ..   oldpeak = col_double(),
  ..   slp = col_double(),
  ..   caa = col_double(),
  ..   thall = col_double(),
  ..   output = col_double()
  .. )
 - attr(*, "problems")=<externalptr>
```

Our variables are;

1. Age : Age of the patient
2. Sex : Sex of the patient
3. exang: exercise induced angina (1 = yes; 0 = no)
4. ca: number of major vessels (0-3)
5. cp : Chest Pain type

Value 1: typical angina Value 2: atypical angina Value 3: non-anginal pain Value 4: asymptomatic

6. trtbps : resting blood pressure (in mm Hg)
7. chol : cholestoral in mg/dl fetched via BMI sensor
8. fbs : (fasting blood sugar > 120 mg/dl) (1 = true; 0 = false)
9. rest_ecg : resting electrocardiographic results

Value 0: normal Value 1: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV) Value 2: showing probable or definite left ventricular hypertrophy by Estes' criteria

10. thalach : maximum heart rate achieved
11. output/target : 0= less chance of heart attack 1= more chance of heart attack

When we look at our data set, we have a whole data set with numeric variables, but some of them will be specified as character (0 and 1 will be yes or no). In this data set, our target variable is output, because we want to predict whether people are at risk of heart attack or not.

```
heart_attack <- na.exclude(heart_attack)
```

First, we have to exclude all the NA's on the data set. Even if we do that, it does change nothing. After that we can check the dimension of the data set by using `dim()` function.

```
dim(heart_attack)
```

```
[1] 303  14
```

We have 303 observation and 14 variables. As we can see, after excluding the NA's observations and variables are remained same values.

Training Logistic Regression Model

We are checking whether people are at risk of heart attack or not, so we have a categorical variables which is our target variable. We have two values in the output variable which are 0 and 1 and they are assigned like that. We should change their class to factor with two levels. It will be needed for training the model.

```
heart_attack$output <- as.factor(heart_attack$output)
class(heart_attack$output)
```

```
[1] "factor"
```

1 - Splitting the Data set

After editing our data set, we have to split the data set to compute target variable. We have to split the data set as a two subset by using `sample()` function. But first, we have to set a seed to get same observations. We create a sample that assigned as index observation. This sample starts with a sequence from 1 to number of row of heart_attack, and we should split 80% of the data set as a train set and the remain part will be the test set.

```
set.seed(123)
index <- sample(1 : nrow(heart_attack), round(nrow(heart_attack) * 0.80))
train <- heart_attack[index, ]
test <- heart_attack[-index, ]
```

2 - Training a logistic regression model

In this part, we want to train our model by using `glm()` function. Firstly, we should write the model formula which is target variable (output) and the features as a '.'. Secondly, we use the train model that we set the previous part. Finally, adding the family distribution of the target variable. We are interesting the binary logistic regression that has a only two outcomes, so it will be set as a 'binomial'.

```
lr_model <- glm(output ~., data = train, family = "binomial")
lr_model
```

```
Call:  glm(formula = output ~ ., family = "binomial", data = train)

Coefficients:
(Intercept)          age          sex           cp       trtbps         chol
   4.555347    -0.013283    -1.540661     0.769378    -0.017916    -0.003182
        fbs      restecg     thalachh         exng      oldpeak          slp
   0.132748     0.658516     0.019761    -1.077694    -0.678520     0.263729
        caa        thall
  -0.646435    -1.015510

Degrees of Freedom: 241 Total (i.e. Null);  228 Residual
Null Deviance:      333.5
Residual Deviance: 173.1    AIC: 201.1
```

When we look at the output, we can saw that model formula, trained data, and coefficients which is estimated values of model parameters. The coefficients show us the log odds ratios of the features. So, interpretting them easier, we can take odds ratios of these features for taking exponential of them by using `exp()` function.

```
exp(lr_model$coefficients)
```

```
(Intercept)         age         sex          cp      trtbps        chol
 95.1397450   0.9868048   0.2142394   2.1584226   0.9822439   0.9968233
        fbs     restecg    thalachh        exng     oldpeak         slp
  1.1419621   1.9319239   1.0199577   0.3403796   0.5073671   1.3017754
        caa       thall
  0.5239100   0.3622178
```

Taking exponential of them causes that if we increase the value of age by 1 unit, it explains the target variable as approximately 0.98.

We use `summary()` function to see more details about the dataset, and the it shows us the model formula, residuals, and coefficients. We can see the results and make inferences using these statistics for each parameter. If any class of a categorical variable is significant that means the others are significant too.

```
summary(lr_model)
```

```
Call:
glm(formula = output ~ ., family = "binomial", data = train)

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  4.555347   2.794683   1.630  0.10310
age         -0.013283   0.025289  -0.525  0.59941
sex         -1.540661   0.509861  -3.022  0.00251 **
cp           0.769378   0.195309   3.939 8.17e-05 ***
trtbps      -0.017916   0.011003  -1.628  0.10348
chol        -0.003182   0.004177  -0.762  0.44617
fbs          0.132748   0.566248   0.234  0.81465
restecg      0.658516   0.389009   1.693  0.09049 .
thalachh     0.019761   0.011439   1.727  0.08408 .
exng        -1.077694   0.448609  -2.402  0.01629 *
oldpeak     -0.678520   0.240961  -2.816  0.00486 **
```

```
slp          0.263729   0.411859   0.640  0.52195
caa         -0.646435   0.201346  -3.211  0.00132 **
thall       -1.015510   0.330194  -3.075  0.00210 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 333.48  on 241  degrees of freedom
Residual deviance: 173.07  on 228  degrees of freedom
AIC: 201.07

Number of Fisher Scoring iterations: 6
```

3 - Measuring model performance

We can check the model performance of the model on test set. For model performance, we have to compute the predicted value of target variable on test set. We should not forget to exclude the target variable on test set. Predicting the values, we can check the first six values using `head()` function.

```r
predicted_probs <- predict(lr_model, test[ ,-14], type = "response")
head(predicted_probs)
```

```
        1         2         3         4         5         6
0.7079596 0.9397678 0.9823383 0.9748455 0.6193038 0.9260086
```

This vector has a probability of people who have a higher chance of having a heart attack. After that we should transform these probabilities to classes by using `ifelse()` function. We set the condition like that. If there is greater than 0.5, it means "1". If it is smaller than 0.05, it assigned "0".

```r
predicted_classes <- ifelse(predicted_probs > 0.5, 1, 0)
head(predicted_classes)
```

```
1 2 3 4 5 6
1 1 1 1 1 1
```

We can create confusion matrix using the metrics. We assign the positive and negative classes as a 1 and 0.

```
TP <- sum(predicted_classes[which(test$output == "1")] == 1)
FP <- sum(predicted_classes[which(test$output == "1")] == 0)
TN <- sum(predicted_classes[which(test$output == "0")] == 0)
FN <- sum(predicted_classes[which(test$output == "0")] == 1)

recall      <- TP / (TP + FN)
specificity <- TN / (TN + FP)
precision   <- TP / (TP + FP)
accuracy    <- (TN + TP) / (TP + FP + TN + FN)

recall
```

```
[1] 0.7948718
```

```
specificity
```

```
[1] 0.9090909
```

```
precision
```

```
[1] 0.9393939
```

```
accuracy
```

```
[1] 0.8360656
```

When we look at the calculated model performance metrics, the precision is the highest one, and it contains %93. Accuracy of the model is classified %83.

```
confusionMatrix(table(ifelse(test$output == "1", "1", "0"),
                      predicted_classes),
                positive = "1")
```

```
Confusion Matrix and Statistics

   predicted_classes
     0  1
  0 20  8
  1  2 31

               Accuracy : 0.8361
                 95% CI : (0.7191, 0.9185)
    No Information Rate : 0.6393
    P-Value [Acc > NIR] : 0.000614

                  Kappa : 0.6645

 Mcnemar's Test P-Value : 0.113846

            Sensitivity : 0.7949
            Specificity : 0.9091
         Pos Pred Value : 0.9394
         Neg Pred Value : 0.7143
             Prevalence : 0.6393
         Detection Rate : 0.5082
   Detection Prevalence : 0.5410
      Balanced Accuracy : 0.8520

       'Positive' Class : 1
```

The confusion matrix shows us the how many observation values the data set has. We have observations of predicted classes, accuracy, precision. The accuracy shows us the performance of the model, and it is 83% as we computed above. Positive prediction value means that precision, and it is 93%.

4 - ROC Curve and AUC

We can construct the metrics using ROC curve.

```
#install.packages("DALEX")
#install.packages("dplyr")
library(dplyr)
```

```
Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

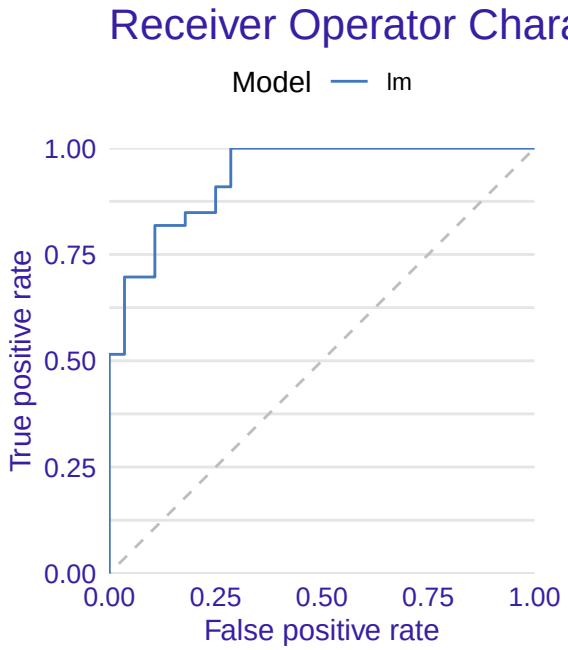
```r
library(DALEX)
```

```
Welcome to DALEX (version: 2.4.3).
Find examples and detailed introduction at: http://ema.drwhy.ai/
Additional features will be available after installation of: ggpubr.
Use 'install_dependencies()' to get all suggested dependencies


Attaching package: 'DALEX'

The following object is masked from 'package:dplyr':

    explain
```

```r
explain_lr <- explain(model = lr_model,
                      data = test[ , -14],
                      y = test$output == "1",
                      type = "classification",
                      verbose = FALSE)
```

After that, we can draw ROC curve.

```r
performance_lr <- model_performance(explain_lr)
plot(performance_lr, geom = "roc")
```



When we look at the graph of ROC curve should be above the dashed line. If the ROC curve is close to this line that means the model is a random model and it does not perform well. When we look at the area under curve value, we can say that this value is 93%. Likewise accuracy is 83%.

```
performance_lr
```

```
Measures for:  classification
recall     : 0.9393939
precision  : 0.7948718
f1         : 0.8611111
accuracy   : 0.8360656
auc        : 0.9339827

Residuals:
         0%         10%         20%         30%         40%         50%
-0.92500426 -0.66440247 -0.23803316 -0.05420990 -0.00710291  0.01057154
        60%         70%         80%         90%        100%
 0.02908870  0.05916045  0.19665257  0.30022165  0.59491469
```

Making Decision Tree

The purpose of decision trees is to split the trained data into homogeneous subgroups. Also, decision trees are often used to model non-linear relationships. Thus, we do not have to check the assumptions as it is non-linear. They work in detail by partitioning features into small pieces.

In the decision tree structure, it starts with the root at the beginning and is divided into branches in order. Here we can also see the features in the dataset and which feature gets how many percentiles. We can see that the root node is named cp at the top. In short, we can say that the root node is the starting point of the decision tree. The next node is called the internal/sub node. Branches help connect these nodes together. Finally, the last nodes are called leaf/terminal nodes and they are the endpoints of the decision tree.

Before we build the decision tree, we need to install the required packages.

```
#install.packages("tidymodels") # it helps us training models.
#install.packages("rpart.plot") # it helps us visualizing the decision tree.
#install.packages("rsample") # helps us splitting the dataset.
#install.packages("recipes")
#install.packages("parsnip") # for model fitting
#install.packages("tune") # for model tuning
#install.packages("yardstick") # for model evaluation
library(recipes)
library(parsnip)
library(tune)
library(yardstick)
library(rsample)
library(tidymodels)
library(rpart.plot)
```

We have separated the dataset at the above so we don't need to separate it again, but here is how to separate the dataset in different ways.

This time, we split the dataset with probability 0.80 using the `initial_split()` function. It is simpler to do this than to get from 1 to the number of rows in the dataset as we did in the first part. Then, we obtain this data set, which we have separated, by using the `training()` and `testing()` functions as train and test set. The results will be the same as the values which we obtained above. In order not to be confused with the above, we assign it to different names.

```
set.seed(123)
heart_attack_split <- initial_split(data = heart_attack, prop = 0.80)
```

```
heart_attack_train <- heart_attack_split |> training()
heart_attack_test  <- heart_attack_split |> testing()
```

1- Defining the model specification

To show what kind of task we are dealing with, we specify the classification task with the `set_mode()` function and assign our model to form a decision tree.

```
heart_attack_model <- decision_tree() |>
  set_engine("rpart") |>
  set_mode("classification") # helps us if it is a regression or classification problem.
```

2 - Training a model for decision tree

In fact, there is no difference with what we did above. It is just that the `fit()` function was used instead of the `glm()` function, and we didn't specify that it should be binomial in addition.

```
ha <- heart_attack_model |>
  fit(output ~., data = heart_attack_train)

ha
```

```
parsnip model object

n= 242

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 242 110 1 (0.45454545 0.54545455)
   2) cp< 0.5 118  34 0 (0.71186441 0.28813559)
     4) caa>=0.5 66   5 0 (0.92424242 0.07575758) *
     5) caa< 0.5 52  23 1 (0.44230769 0.55769231)
      10) thall>=2.5 20   4 0 (0.80000000 0.20000000) *
      11) thall< 2.5 32   7 1 (0.21875000 0.78125000) *
   3) cp>=0.5 124  26 1 (0.20967742 0.79032258)
     6) oldpeak>=2.45 9   2 0 (0.77777778 0.22222222) *
     7) oldpeak< 2.45 115  19 1 (0.16521739 0.83478261)
      14) age>=56.5 44  14 1 (0.31818182 0.68181818)
        28) sex>=0.5 26  12 1 (0.46153846 0.53846154)
```

```
      56) chol>=253 9   1 0 (0.88888889 0.11111111) *
      57) chol< 253 17   4 1 (0.23529412 0.76470588) *
    29) sex< 0.5 18   2 1 (0.11111111 0.88888889) *
  15) age< 56.5 71   5 1 (0.07042254 0.92957746) *
```

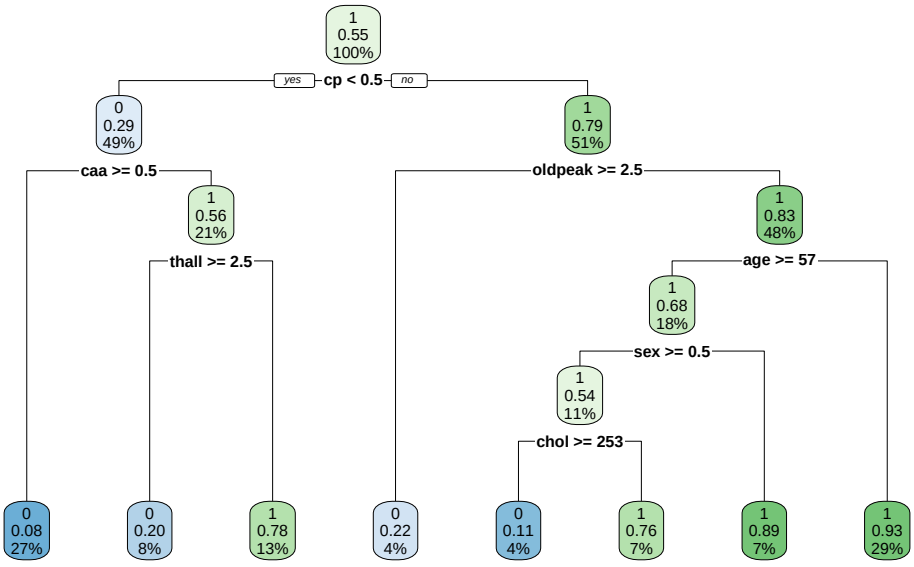
After training the model, the output lists which variables the tree will take from the root to the branches with which values. Since the output is in a complex form, it becomes difficult to interpret, so we need to visualize it.

3 - Visualizing the decision tree

We can convert the values we have obtained into a decision tree using the `{rpart.plot}` package, and visualize it.

```
rpart.plot(ha$fit)
```

```
Warning: Cannot retrieve the data used to build the model (so cannot determine roundint and
To silence this warning:
    Call rpart.plot with roundint=FALSE,
    or rebuild the rpart model with model=TRUE.
```



It is seen that those who have a higher chance of having a heart attack which written in the first line of the root part of our decision tree are in the majority. So 1 is the majority class in this data set. The second row contains the percentage of the intended class. In this data

set, the class we are interested in has a low chance of having a heart attack, that is, 0, so the chance of having a heart attack is given to us as 55%. The last line shows the percentage of observations in these nodes. The reason why it's 100%, it contains the entire dataset. The variable specified at the root is the cp and is specified as smaller than 0.5. If it is smaller than 0.5, we follow the yes part, if not, the no part is followed. All nodes will be separated in this way. We make our choices according to the variable we want to deal with. When we look at the second node, there is a difference in colors because the higher chance of having a heart attack was determined as green, while the lower one was determined as blue. More in fact, the colors represent the majority classes. Whichever class is more, there is a color change in that node. In the second node, in the blue part on the left, the majority class is less likely to have a heart attack. Since our Majority class has a cp is smaller than 0.5, the class we are interested in in the second row will have less percentile. The 49% in the last row shows all the observations in this node. Finally, when we look at the node on the right, 51% of the observations we have went to the part where the cp is greater than 0.5. We can comment in the same way on other nodes.

We can compute the predicted values of the target variables in the test set using `predict()` function. The function we use is no different from the logistic regression model. Since the target variable is determined as 0 and 1 as the output, it will be determined in the same way.

```
ha_predictions <- ha |>
  predict(new_data = heart_attack_test)

ha_predictions
```

```
# A tibble: 61 x 1
   .pred_class
   <fct>
 1 0
 2 1
 3 1
 4 1
 5 1
 6 1
 7 1
 8 1
 9 1
10 1
# i 51 more rows
```

If we want to see the probabilities of the predicted values, it will be sufficient to specify only the "type" as "prob" on the same function.

```
ha |>
  predict(new_data = heart_attack_test,
          type     = "prob")
```

```
# A tibble: 61 x 2
   .pred_0 .pred_1
     <dbl>   <dbl>
 1  0.778    0.222
 2  0.0704   0.930
 3  0.0704   0.930
 4  0.111    0.889
 5  0.219    0.781
 6  0.0704   0.930
 7  0.0704   0.930
 8  0.219    0.781
 9  0.0704   0.930
10  0.0704   0.930
# i 51 more rows
```

As a result, it returns us the probabilities of 0 and 1.

4 - Evaluating model performance

We created a tibble to compare the model prediction and the actual values of our target variable in the test data.

```
ha_results <- tibble(predicted = ha_predictions$.pred_class,
                             actual    = heart_attack_test$output)
ha_results
```

```
# A tibble: 61 x 2
   predicted actual
   <fct>     <fct>
 1 0         1
 2 1         1
 3 1         1
 4 1         1
 5 1         1
```

```
 6 1         1
 7 1         1
 8 1         1
 9 1         1
10 1         1
# i 51 more rows
```

By doing this we can get the confusion matrix.

```
ha_results |> conf_mat(truth    = actual,
                        estimate = predicted)
```

```
          Truth
Prediction  0  1
         0 20  2
         1  8 31
```

If we want to calculate the accuracy, we just use the accuracy() function. After calculating accuracy we have a result %75.

```
ha_results |> accuracy(truth = actual, estimate = predicted)
```

```
# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.836
```

We can calculate sensitivity metric like the way we use accuracy, we can use sens() function. After that, we obtain sensitivity approximately %60.

```
ha_results |> sens(truth = actual, estimate = predicted)
```

```
# A tibble: 1 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 sens    binary         0.714
```

The result of specificity is %87.

```r
ha_results |> spec(truth = actual, estimate = predicted)
```

```
# A tibble: 1 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 spec    binary         0.939
```

We can compute the model fitting using last_fit() function.

```r
ha_last_fit <- heart_attack_model |>
  last_fit(output ~., split = heart_attack_split)
ha_last_fit
```

```
# Resampling results
# Manual resampling
# A tibble: 1 x 6
  splits           id               .metrics .notes   .predictions .workflow
  <list>           <chr>            <list>   <list>   <list>       <list>
1 <split [242/61]> train/test split <tibble> <tibble> <tibble>     <workflow>
```

After the model fitting, we can obtain the metrics and predictions using `collect_metrics()` and `collect_predictions()` functions.

```r
ha_last_fit |> collect_metrics()
```

```
# A tibble: 2 x 4
  .metric  .estimator .estimate .config
  <chr>    <chr>          <dbl> <chr>
1 accuracy binary         0.836 Preprocessor1_Model1
2 roc_auc  binary         0.893 Preprocessor1_Model1
```

```r
ha_last_fit |> collect_predictions()
```

```
# A tibble: 61 x 7
   id               .pred_0 .pred_1  .row .pred_class output .config
   <chr>              <dbl>   <dbl> <int> <fct>       <fct>  <chr>
```

```
 1 train/test split  0.778    0.222     2 0           1      Preprocessor1_Mode~
 2 train/test split  0.0704   0.930     3 1           1      Preprocessor1_Mode~
 3 train/test split  0.0704   0.930    12 1           1      Preprocessor1_Mode~
 4 train/test split  0.111    0.889    15 1           1      Preprocessor1_Mode~
 5 train/test split  0.219    0.781    19 1           1      Preprocessor1_Mode~
 6 train/test split  0.0704   0.930    28 1           1      Preprocessor1_Mode~
 7 train/test split  0.0704   0.930    38 1           1      Preprocessor1_Mode~
 8 train/test split  0.219    0.781    44 1           1      Preprocessor1_Mode~
 9 train/test split  0.0704   0.930    47 1           1      Preprocessor1_Mode~
10 train/test split  0.0704   0.930    49 1           1      Preprocessor1_Mode~
# i 51 more rows
```

Randomforest

Randomforest is often used the subset of features to train each tree. The main hyperparameters of randomforest are;

1- the number of trees 2- the number of features to consider at any given split: (mtry) 3- the complexity of each tree

Before starting we should install `{ranger}` package.

```
#install.packages("ranger")
library(ranger)
```

1 - Training random forests

After installing the package, using the ranger() function, we determined the number of features, namely mtry, as 8 according to the target variable in our train data and assigned it to trained_ha.

```
set.seed(123)
trained_ha <- ranger(output ~ .,
                     data = heart_attack_train,
                     mtry = 8)
trained_ha
```

```
Ranger result

Call:
 ranger(output ~ ., data = heart_attack_train, mtry = 8)
```

```
Type:                             Classification
Number of trees:                  500
Sample size:                      242
Number of independent variables:  13
Mtry:                             8
Target node size:                 1
Variable importance mode:         none
Splitrule:                        gini
OOB prediction error:             21.49 %
```

trained_ha includes the hyperparameters mentioned above. Since it would be better to obtain optimal value of the number of features at 10 times, the value that we have is 500.

If we run the same code this time without specifying mtry, it will automatically specify mtry as 3.

```
set.seed(123)
trained_rf_ha <- ranger(output ~ .,
                        data = heart_attack_train)
trained_rf_ha
```

```
Ranger result

Call:
 ranger(output ~ ., data = heart_attack_train)

Type:                             Classification
Number of trees:                  500
Sample size:                      242
Number of independent variables:  13
Mtry:                             3
Target node size:                 1
Variable importance mode:         none
Splitrule:                        gini
OOB prediction error:             18.60 %
```

2 - Compare the test performance of the models for randomforests

While we measure the performance of the model where we set mtry as 8 in the first code, we measure the performance of the model by using the automatically determined value in the second code. Accuracy increased slightly from 80% with mtry = 8 to 83% at 3. Likewise, their

balanced accuracy also led to a similar slight increase. In this case, mtry is a hyperparameter that affects the performance of the model.

```
preds_bt_ha <- predict(trained_ha, heart_attack_test)

confusionMatrix(preds_bt_ha$predictions,
                heart_attack_test$output)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 19  3
         1  9 30

               Accuracy : 0.8033
                 95% CI : (0.6816, 0.894)
    No Information Rate : 0.541
    P-Value [Acc > NIR] : 1.767e-05

                  Kappa : 0.5974

 Mcnemar's Test P-Value : 0.1489

            Sensitivity : 0.6786
            Specificity : 0.9091
         Pos Pred Value : 0.8636
         Neg Pred Value : 0.7692
             Prevalence : 0.4590
         Detection Rate : 0.3115
   Detection Prevalence : 0.3607
      Balanced Accuracy : 0.7938

       'Positive' Class : 0
```

```
preds_rf_ha <- predict(trained_rf_ha, heart_attack_test)

confusionMatrix(preds_rf_ha$predictions,
                heart_attack_test$output)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 19  1
         1  9 32

               Accuracy : 0.8361
                 95% CI : (0.7191, 0.9185)
    No Information Rate : 0.541
    P-Value [Acc > NIR] : 1.184e-06

                  Kappa : 0.6626

 Mcnemar's Test P-Value : 0.02686

            Sensitivity : 0.6786
            Specificity : 0.9697
         Pos Pred Value : 0.9500
         Neg Pred Value : 0.7805
             Prevalence : 0.4590
         Detection Rate : 0.3115
   Detection Prevalence : 0.3279
      Balanced Accuracy : 0.8241

       'Positive' Class : 0
```

When we look at all three techniques, the accuracy of the model was 83% in the logistic regression, while it was 0.75 in the decision tree and 80% in the randomforest technique. But when the mtry hyperparameter in randomforest changes, or rather, when it gets smaller, this situation has slightly increased.