Classification Modeling

Choonghyun Ryu

2020-01-23

Preface

Once the data set is ready for model development, the model is fitted, predicted and evaluated in the following ways:

- Cleansing the dataset
- Split the data into a train set and a test set
- **Modeling and Evaluate, Predict**
  - **Modeling**
    - **Binary classification modeling**
  - **Evaluate the model**
    - **Predict test set using fitted model**
    - **Calculate the performance metric**
    - **Plot the ROC curve**
    - **Tunning the cut-off**
  - **Predict**
    - **Predict**
    - **Predict with cut-off**

The alookr package makes these steps fast and easy:

Data: Wisconsin Breast Cancer Data

`BreastCancer` of `mlbench package` is a breast cancer data. The objective is to identify each of a number of benign or malignant classes.

A data frame with 699 observations on 11 variables, one being a character variable, 9 being ordered or nominal, and 1 target class.:

- `Id` : character. Sample code number
- `Cl.thickness` : ordered factor. Clump Thickness
- `Cell.size` : ordered factor. Uniformity of Cell Size
- `Cell.shape` : ordered factor. Uniformity of Cell Shape
- `Marg.adhesion` : ordered factor. Marginal Adhesion
- `Epith.c.size` : ordered factor. Single Epithelial Cell Size
- `Bare.nuclei` : factor. Bare Nuclei
- `Bl.cromatin` : factor. Bland Chromatin
- `Normal.nucleoli` : factor. Normal Nucleoli
- `Mitoses` : factor. Mitoses
- `Class` : factor. Class. level is `benign` and `malignant`.

```r
library(mlbench)
data(BreastCancer)

# class of each variables
sapply(BreastCancer, function(x) class(x)[1])
             Id    Cl.thickness       Cell.size      Cell.shape   Marg.adhesion
    "character"       "ordered"       "ordered"       "ordered"       "ordered"
   Epith.c.size     Bare.nuclei     Bl.cromatin Normal.nucleoli         Mitoses
```

```
      "ordered"        "factor"        "factor"        "factor"        "factor" 
          Class 
       "factor" 
```

Preperation the data

Perform data preprocessing as follows.:

- Find and imputate variables that contain missing values.
- Split the data into a train set and a test set.
- To solve the imbalanced class, perform sampling in the train set of raw data.
- Cleansing the dataset for classification modeling.

Fix the missing value with `dlookr::imputate_na()`

find the variables that include missing value. and imputate the missing value using imputate_na() in dlookr package.

```
library(dlookr)
library(dplyr)

# variable that have a missing value
diagnose(BreastCancer) %>%
  filter(missing_count > 0)
# A tibble: 1 x 6
  variables   types  missing_count missing_percent unique_count unique_rate
  <chr>       <chr>          <int>           <dbl>        <int>       <dbl>
1 Bare.nuclei factor            16            2.29           11      0.0157

# imputation of missing value
breastCancer <- BreastCancer %>%
  mutate(Bare.nuclei = imputate_na(BreastCancer, Bare.nuclei, Class,
                         method = "mice", no_attrs = TRUE, print_flag = FALSE))
```

Split data set

Splits the dataset into a train set and a test set with `split_by()`

`split_by()` in the alookr package splits the dataset into a train set and a test set.

The ratio argument of the `split_by()` function specifies the ratio of the train set.

`split_by()` creates a class object named split_df.

```
library(alookr)

# split the data into a train set and a test set by default arguments
sb <- breastCancer %>%
  split_by(target = Class)

# show the class name
class(sb)
[1] "split_df"   "grouped_df" "tbl_df"     "tbl"        "data.frame"

# split the data into a train set and a test set by ratio = 0.6
tmp <- breastCancer %>%
  split_by(Class, ratio = 0.6)
```

The `summary()` function displays the following useful information about the split_df object:

- random seed : The random seed is the random seed used internally to separate the data
- split data : Information of splited data
  - train set count : number of train set
  - test set count : number of test set
- target variable : Target variable name
  - minority class : name and ratio(In parentheses) of minority class
  - majority class : name and ratio(In parentheses) of majority class

```
# summary() display the some information
summary(sb)
** Split train/test set information **
 + random seed        :  43694
 + split data
    - train set count :  489
    - test set count  :  210
 + target variable    :  Class
    - minority class  :  malignant (0.344778)
    - majority class  :  benign (0.655222)

# summary() display the some information
summary(tmp)
** Split train/test set information **
 + random seed        :  17177
 + split data
    - train set count :  419
    - test set count  :  280
 + target variable    :  Class
    - minority class  :  malignant (0.344778)
    - majority class  :  benign (0.655222)
```

Check missing levels in the train set

In the case of categorical variables, when a train set and a test set are separated, a specific level may be missing from the train set.

In this case, there is no problem when fitting the model, but an error occurs when predicting with the model you created. Therefore, preprocessing is performed to avoid missing data preprocessing.

In the following example, fortunately, there is no categorical variable that contains the missing levels in the train set.

```
# list of categorical variables in the train set that contain missing levels
nolevel_in_train <- sb %>%
  compare_category() %>%
  filter(train == 0) %>%
  select(variable) %>%
  unique() %>%
  pull

nolevel_in_train
character(0)

# if any of the categorical variables in the train set contain a missing level,
# split them again.
```

```
while (length(nolevel_in_train) > 0) {
  sb <- breastCancer %>%
    split_by(Class)

  nolevel_in_train <- sb %>%
    compare_category() %>%
    filter(train == 0) %>%
    select(variable) %>%
    unique() %>%
    pull
}
```

Handling the imbalanced classes data with `sampling_target()`

Issue of imbalanced classes data

Imbalanced classes(levels) data means that the number of one level of the frequency of the target variable is relatively small. In general, the proportion of positive classes is relatively small. For example, in the model of predicting spam, the class of interest spam is less than non-spam.

Imbalanced classes data is a common problem in machine learning classification.

`table()` and `prop.table()` are traditionally useful functions for diagnosing imbalanced classes data. However, alookr's `summary()` is simpler and provides more information.

```
# train set frequency table - imbalanced classes data
table(sb$Class)

   benign malignant 
      458       241 

# train set relative frequency table - imbalanced classes data
prop.table(table(sb$Class))

   benign malignant 
0.6552217 0.3447783 

# using summary function - imbalanced classes data
summary(sb)
** Split train/test set information **
 + random seed        :  43694
 + split data
    - train set count :  489
    - test set count  :  210
 + target variable    :  Class
    - minority class  :  malignant (0.344778)
    - majority class  :  benign (0.655222)
```

Handling the imbalanced classes data

Most machine learning algorithms work best when the number of samples in each class are about equal. And most algorithms are designed to maximize accuracy and reduce error. So, we requre handling an imbalanced class problem.

sampling_target() performs sampling to solve an imbalanced classes data problem.

Resampling - oversample minority class

Oversampling can be defined as adding more copies of the minority class.

Oversampling is performed by specifying "ubOver" in the method argument of the `sampling_target()` function.

```
# to balanced by over sampling
train_over <- sb %>%
  sampling_target(method = "ubOver")

# frequency table
table(train_over$Class)

   benign malignant
      319       319
```

Resampling - undersample majority class

Undersampling can be defined as removing some observations of the majority class.

Undersampling is performed by specifying "ubUnder" in the method argument of the `sampling_target()` function.

```
# to balanced by under sampling
train_under <- sb %>%
  sampling_target(method = "ubUnder")

# frequency table
table(train_under$Class)

   benign malignant
      170       170
```

Generate synthetic samples - SMOTE

SMOTE(Synthetic Minority Oversampling Technique) uses a nearest neighbors algorithm to generate new and synthetic data.

SMOTE is performed by specifying "ubSMOTE" in the method argument of the `sampling_target()` function.

```
# to balanced by SMOTE
train_smote <- sb %>%
  sampling_target(seed = 1234L, method = "ubSMOTE")

# frequency table
table(train_smote$Class)

   benign malignant
      680       510
```

Cleansing the dataset for classification modeling with `cleanse()`

The `cleanse()` cleanse the dataset for classification modeling.

This function is useful when fit the classification model. This function does the following.:

- Remove the variable with only one value.

- And remove variables that have a unique number of values relative to the number of observations for a character or categorical variable.
  - In this case, it is a variable that corresponds to an identifier or an identifier.
- And converts the character to factor.

In this example, The `cleanse()` function removed a variable ID with a high unique rate.

```
# clean the training set
train <- train_smote %>%
  cleanse
── Checking unique value ─────────────────────────── unique value is one ──
No variables that unique value is one.

── Checking unique rate ─────────────────────────────── high unique rate ──
remove variables with high unique rate
● Id = 435(0.365546218487395)

── Checking character variables ───────────────────────── categorical data ──
No character variables.
```

Extract test set for evaluation of the model with `extract_set()`

```
# extract test set
test <- sb %>%
  extract_set(set = "test")
```

Binary classification modeling with `run_models()`

`run_models()` performs some representative binary classification modeling using `split_df` object created by `split_by()`.

Currently supported algorithms are as follows.:

- logistic : logistic regression using using `stats` package
- rpart : Recursive Partitioning Trees using `rpart` package
- ctree : Conditional Inference Trees using `party` package
- randomForest :Classification with Random Forest using `randomForest` package
- ranger : A Fast Implementation of Random Forests using `ranger` package

`run_models()` returns a `model_df` class object.

The `model_df` class object contains the following variables.:

- step : character. The current stage in the classification modeling process.
  - For objects created with `run_models()`, the value of the variable is "1.Fitted".
- model_id : model identifiers
- target : name of target variable
- positive : positive class in target variable
- fitted_model : list. Fitted model object by model_id's algorithms

```
result <- train %>%
  run_models(target = "Class", positive = "malignant")
result
# A tibble: 5 x 5
  step     model_id     target positive  fitted_model
  <chr>    <chr>        <chr>  <chr>     <list>
1 1.Fitted logistic     Class  malignant <glm>
```

```
2 1.Fitted rpart        Class  malignant <rpart>
3 1.Fitted ctree        Class  malignant <BinaryTr>
4 1.Fitted randomForest Class  malignant <rndmFrs.>
5 1.Fitted ranger       Class  malignant <ranger>
```

Evaluate the model

Evaluate the predictive performance of fitted models.

Predict test set using fitted model with `run_predict()`

`run_predict()` predict the test set using `model_df` class fitted by `run_models()`.

The `model_df` class object contains the following variables.:

- step : character. The current stage in the classification modeling process.
  - For objects created with `run_predict()`, the value of the variable is "2.Predicted".
- model_id : character. Type of fit model.
- target : character. Name of target variable.
- positive : character. Level of positive class of binary classification.
- fitted_model : list. Fitted model object by model_id's algorithms.
- predicted : result of predcit by each models

```
pred <- result %>%
  run_predict(test)
pred
# A tibble: 5 x 6
  step        model_id     target positive  fitted_model predicted
  <chr>       <chr>        <chr>  <chr>     <list>       <list>
1 2.Predicted logistic     Class  malignant <glm>        <fct [210]>
2 2.Predicted rpart        Class  malignant <rpart>      <fct [210]>
3 2.Predicted ctree        Class  malignant <BinaryTr>   <fct [210]>
4 2.Predicted randomForest Class  malignant <rndmFrs.>   <fct [210]>
5 2.Predicted ranger       Class  malignant <ranger>     <fct [210]>
```

Calculate the performance metric with `run_performance()`

`run_performance()` calculate the performance metric of `model_df` class predicted by `run_predict()`.

The `model_df` class object contains the following variables.:

- step : character. The current stage in the classification modeling process.
  - For objects created with `run_performance()`, the value of the variable is "3.Performanced".
- model_id : character. Type of fit model.
- target : character. Name of target variable.
- positive : character. Level of positive class of binary classification.
- fitted_model : list. Fitted model object by model_id's algorithms
- predicted : list. Predicted value by individual model. Each value has a predict_class class object.
- performance : list. Calculate metrics by individual model. Each value has a numeric vector.

```
# Calculate performace metrics.
perf <- run_performance(pred)
perf
# A tibble: 5 x 7
  step          model_id     target positive fitted_model predicted  performance
  <chr>         <chr>        <chr>  <chr>    <list>       <list>     <list>
1 3.Performanc... logistic     Class  maligna... <glm>        <fct [210... <dbl [15]>
```

```
2 3.Performanc... rpart        Class  maligna... <rpart>      <fct [210... <dbl [15]>
3 3.Performanc... ctree        Class  maligna... <BinaryTr>   <fct [210... <dbl [15]>
4 3.Performanc... randomForest Class  maligna... <rndmFrs.>   <fct [210... <dbl [15]>
5 3.Performanc... ranger       Class  maligna... <ranger>     <fct [210... <dbl [15]>
```

The performance variable contains a list object, which contains 15 performance metrics:

- ZeroOneLoss : Normalized Zero-One Loss(Classification Error Loss).
- Accuracy : Accuracy.
- Precision : Precision.
- Recall : Recall.
- Sensitivity : Sensitivity.
- Specificity : Specificity.
- F1_Score : F1 Score.
- Fbeta_Score : F-Beta Score.
- LogLoss : Log loss / Cross-Entropy Loss.
- AUC : Area Under the Receiver Operating Characteristic Curve (ROC AUC).
- Gini : Gini Coefficient.
- PRAUC : Area Under the Precision-Recall Curve (PR AUC).
- LiftAUC : Area Under the Lift Chart.
- GainAUC : Area Under the Gain Chart.
- KS_Stat : Kolmogorov-Smirnov Statistic.

```
# Performance by analytics models
performance <- perf$performance
names(performance) <- perf$model_id
performance
$logistic
ZeroOneLoss    Accuracy   Precision      Recall Sensitivity Specificity
 0.08095238  0.91904762  0.83750000  0.94366197  0.94366197  0.90647482
   F1_Score Fbeta_Score     LogLoss         AUC        Gini       PRAUC
 0.88741722  0.88741722  2.59993981  0.93165468  0.89968589  0.03282411
    LiftAUC     GainAUC     KS_Stat
 1.09419244  0.78571429 86.45252812

$rpart
ZeroOneLoss    Accuracy   Precision      Recall Sensitivity Specificity
 0.06190476  0.93809524  0.86250000  0.97183099  0.97183099  0.92086331
   F1_Score Fbeta_Score     LogLoss         AUC        Gini       PRAUC
 0.91390728  0.91390728  0.21706627  0.94553653  0.92055933  0.01580544
    LiftAUC     GainAUC     KS_Stat
 1.05689745  0.79490275 89.26942953

$ctree
ZeroOneLoss    Accuracy   Precision      Recall Sensitivity Specificity
 0.04285714  0.95714286  0.94285714  0.92957746  0.92957746  0.97122302
   F1_Score Fbeta_Score     LogLoss         AUC        Gini       PRAUC
 0.93617021  0.93617021  0.58588256  0.97948120  0.95338940  0.25765179
    LiftAUC     GainAUC     KS_Stat
 1.36137711  0.81737089 90.08004864

$randomForest
ZeroOneLoss    Accuracy   Precision      Recall Sensitivity Specificity
 0.03809524  0.96190476  0.89873418  1.00000000  1.00000000  0.94244604
   F1_Score Fbeta_Score     LogLoss         AUC        Gini       PRAUC
 0.94666667  0.94666667  0.12480154  0.99376837  0.98703009  0.64673174 
    LiftAUC     GainAUC     KS_Stat 
 1.76056298  0.82682763 95.02482521 

$ranger
ZeroOneLoss    Accuracy   Precision      Recall Sensitivity Specificity 
 0.04285714  0.95714286  0.89743590  0.98591549  0.98591549  0.94244604 
   F1_Score Fbeta_Score     LogLoss         AUC        Gini       PRAUC 
 0.93959732  0.93959732  0.11479786  0.99432567  0.98865133  0.85988479 
    LiftAUC     GainAUC     KS_Stat 
 1.96553511  0.82719651 94.99442699 
```

If you change the list object to tidy format, you'll see the following at a glance:

```
# Convert to matrix for compare performace.
sapply(performance, "c")
               logistic       rpart       ctree randomForest      ranger
ZeroOneLoss  0.08095238  0.06190476  0.04285714   0.03809524  0.04285714
Accuracy     0.91904762  0.93809524  0.95714286   0.96190476  0.95714286
Precision    0.83750000  0.86250000  0.94285714   0.89873418  0.89743590
Recall       0.94366197  0.97183099  0.92957746   1.00000000  0.98591549
Sensitivity  0.94366197  0.97183099  0.92957746   1.00000000  0.98591549
Specificity  0.90647482  0.92086331  0.97122302   0.94244604  0.94244604
F1_Score     0.88741722  0.91390728  0.93617021   0.94666667  0.93959732
Fbeta_Score  0.88741722  0.91390728  0.93617021   0.94666667  0.93959732
LogLoss      2.59993981  0.21706627  0.58588256   0.12480154  0.11479786
AUC          0.93165468  0.94553653  0.97948120   0.99376837  0.99432567
Gini         0.89968589  0.92055933  0.95338940   0.98703009  0.98865133
PRAUC        0.03282411  0.01580544  0.25765179   0.64673174  0.85988479
LiftAUC      1.09419244  1.05689745  1.36137711   1.76056298  1.96553511
GainAUC      0.78571429  0.79490275  0.81737089   0.82682763  0.82719651
KS_Stat     86.45252812 89.26942953 90.08004864  95.02482521 94.99442699
```

`compare_performance()` return a list object(results of compared model performance). and list has the following components:

- recommend_model : character. The name of the model that is recommended as the best among the various models.
- top_count : numeric. The number of best performing performance metrics by model.
- mean_rank : numeric. Average of ranking individual performance metrics by model.
- top_metric : list. The name of the performance metric with the best performance on individual performance metrics by model.

In this example, `compare_performance()` recommend the **"ranger"** model.

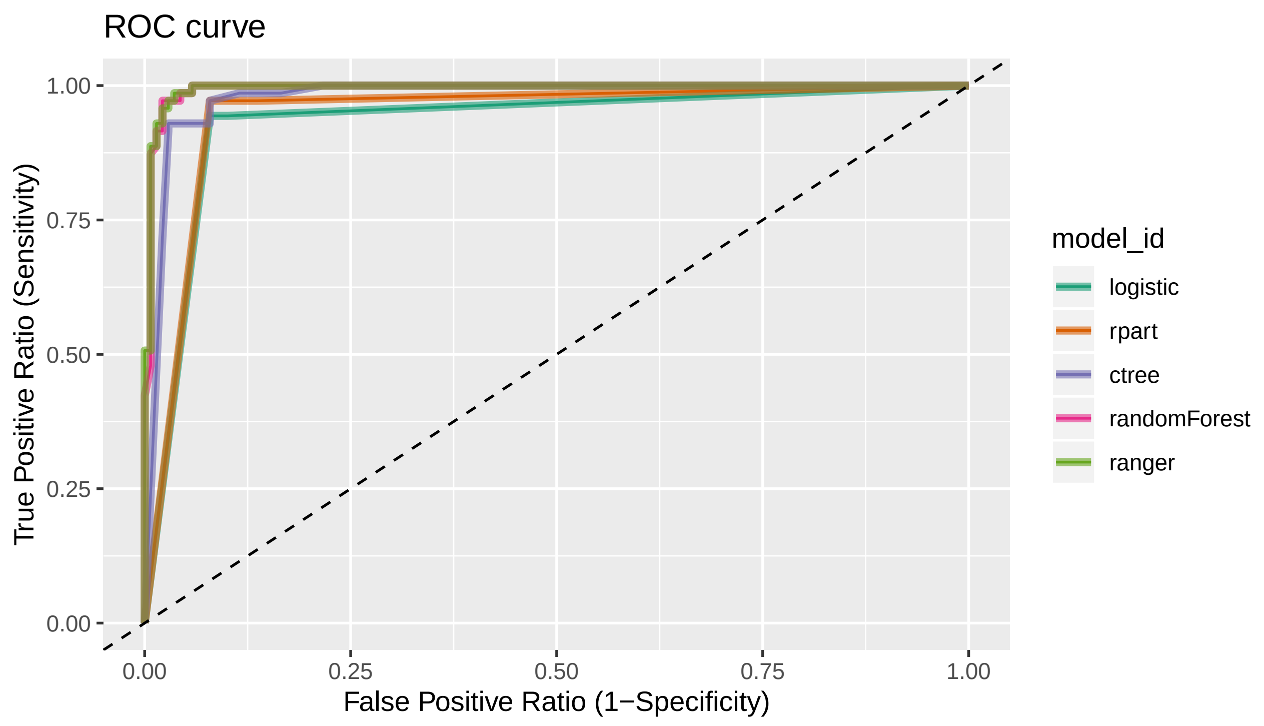
```
# Compaire the Performance metrics of each model
comp_perf <- compare_performance(pred)
comp_perf
$recommend_model
[1] "ranger"

$top_metric_count
    logistic        rpart        ctree randomForest       ranger 
           0            0            2            5            6 

$mean_rank
```

```
    logistic        rpart        ctree randomForest       ranger 
    4.769231     4.000000     2.846154     1.653846     1.730769 

$top_metric
$top_metric$logistic
NULL

$top_metric$rpart
NULL

$top_metric$ctree
[1] "Precision"   "Specificity"

$top_metric$randomForest
[1] "ZeroOneLoss" "Accuracy"    "Recall"      "F1_Score"    "KS_Stat"    

$top_metric$ranger
[1] "LogLoss" "AUC"     "Gini"    "PRAUC"   "LiftAUC" "GainAUC"
```

Plot the ROC curve with `plot_performance()`

`compare_performance()` plot ROC curve.

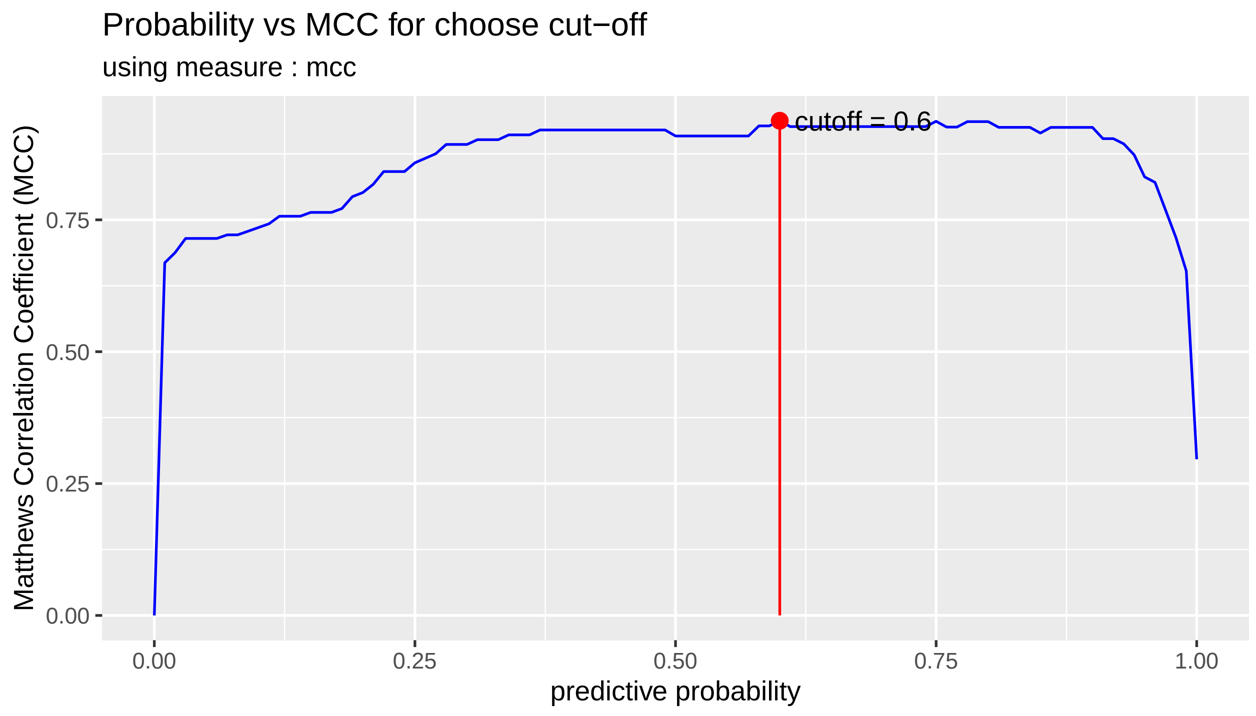
```r
# Plot ROC curve
plot_performance(pred)
```



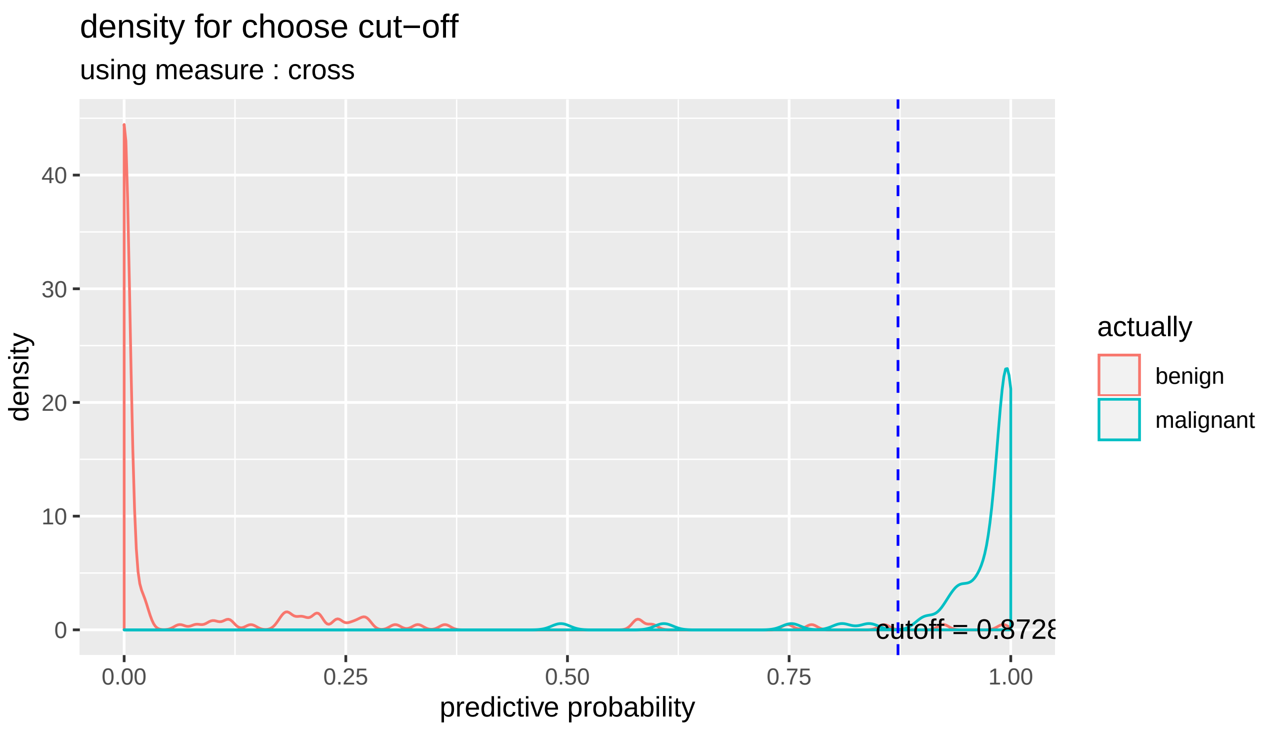
Tunning the cut-off

Compare the statistics of the numerical variables of the train set and test set included in the "split_df" class.

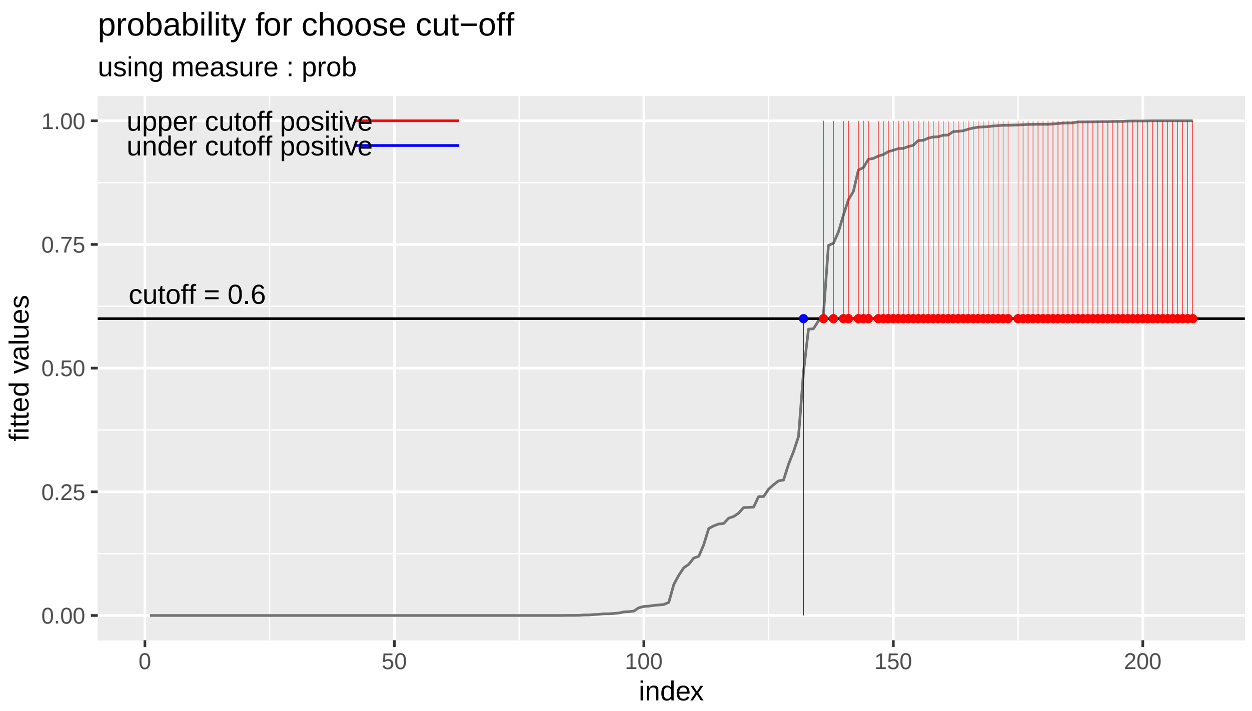
```r
pred_best <- pred %>% 
  filter(model_id == comp_perf$recommend_model) %>% 
  select(predicted) %>%
  pull %>%
  .[[1]] %>%
  attr("pred_prob")

cutoff <- plot_cutoff(pred_best, test$Class, "malignant", type = "mcc")
```



```
cutoff
[1] 0.6

cutoff2 <- plot_cutoff(pred_best, test$Class, "malignant", type = "density")
```



```
cutoff2
[1] 0.8728

cutoff3 <- plot_cutoff(pred_best, test$Class, "malignant", type = "prob")
```



```
cutoff3
[1] 0.6
```

Performance comparison between prediction and tuned cut-off with `performance_metric()`

Compare the performance of the original prediction with that of the tuned cut-off. Compare the cut-off with the non-cut model for the model with the best performance `comp_perf$recommend_model`.

```
comp_perf$recommend_model
[1] "ranger"

# extract predicted probability
idx <- which(pred$model_id == comp_perf$recommend_model)
pred_prob <- attr(pred$predicted[[idx]], "pred_prob")

# or, extract predicted probability using dplyr
pred_prob <- pred %>%
  filter(model_id == comp_perf$recommend_model) %>%
  select(predicted) %>%
  pull %>%
  "[["(1) %>%
  attr("pred_prob")

# predicted probability
pred_prob
  [1] 0.0049896825 0.9954333333 0.1429658730 0.8093626984 0.0024000000
  [6] 0.9986555556 0.9287301587 0.0000000000 0.9958000000 0.0000000000
 [11] 0.0000000000 0.9920293651 0.0000000000 0.9997777778 0.9912206349
 [16] 0.9978444444 0.0000000000 0.9677214286 0.0010666667 0.9781746032
 [21] 0.0000000000 0.1194746032 0.0000000000 1.0000000000 0.0000000000
 [26] 0.0000000000 0.0000000000 0.9936793651 0.6087642857 0.9798293651
 [31] 0.0033738095 0.2553722222 0.9503380952 0.9376650794 0.1859253968
 [36] 0.1811722222 0.5956142857 0.0156634921 0.0212690476 0.8404698413
 [41] 0.0000000000 0.0000000000 0.9908960317 0.0000000000 0.0018666667
 [46] 0.3057912698 0.9221214286 0.0000000000 0.2190896825 0.0000000000
 [51] 0.0000000000 0.0000000000 1.0000000000 0.9876396825 0.0000000000
 [56] 0.0000000000 0.9978571429 0.9982555556 0.9929277778 1.0000000000
 [61] 0.9479690476 0.9996000000 0.9944396825 0.0000000000 1.0000000000
 [66] 0.9996000000 0.0000000000 0.9870484127 0.0000000000 0.9605000000
 [71] 0.0000000000 1.0000000000 0.2722095238 0.9673444444 0.2403992063
 [76] 0.9993777778 0.0000000000 0.9437642857 0.9956920635 0.9601039683
 [81] 0.2643095238 0.9929333333 0.9901880952 0.0000000000 0.9986555556
 [86] 0.9927023810 0.8571626984 0.7481063492 0.7524380952 0.0000000000
 [91] 0.0000000000 0.2738325397 0.0000000000 0.9239761905 0.0221912698
 [96] 0.4924555556 0.0000000000 0.9854055556 0.0204436508 0.9405420635
[101] 0.0000000000 0.9711936508 0.0000000000 0.1033920635 0.0033738095
[106] 0.0000000000 0.9996000000 1.0000000000 0.5796214286 0.0000000000
[111] 0.0000000000 0.0182325397 0.0000000000 0.0000000000 0.9909214286
[116] 0.0000000000 0.9982920635 0.0188722222 0.0000000000 0.0000000000
[121] 0.0626325397 0.0000000000 0.1757960317 0.0000000000 0.9981071429
[126] 0.1161666667 0.0000000000 0.0000000000 0.9915833333 0.0072539683
[131] 0.3312373016 0.0000000000 0.0000000000 0.9985444444 0.0000000000
[136] 0.2182507937 0.0002222222 0.0002222222 0.9979777778 0.0009398693
[141] 0.9929206349 0.9925714286 0.5788105042 0.0000000000 0.0000000000
[146] 0.0000000000 0.0000000000 0.9881833333 0.0000000000 0.9441063492
[151] 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
[156] 0.0000000000 0.0089650794 0.0000000000 0.0000000000 0.0000000000
[161] 0.0000000000 0.0964714286 1.0000000000 0.9006063492 0.9894666667
```

```
[166] 0.0000000000 0.0000000000 0.0000000000 0.9977777778 0.9053000000
[171] 0.9829873016 0.0000000000 0.2068896825 0.9786944444 0.0000000000
[176] 0.9317365079 0.0000000000 0.2403563492 0.9649238095 0.1849722222
[181] 0.0000000000 0.0000000000 0.2000841270 0.3617515873 0.0000000000
[186] 0.0002857143 0.0040952381 0.0000000000 0.0000000000 0.0000000000
[191] 1.0000000000 0.0813015873 0.0000000000 0.0000000000 0.0000000000
[196] 0.7753238095 0.9708555556 0.0001111111 0.0264730159 0.0000000000
[201] 0.0077357143 0.1967507937 0.0000000000 1.0000000000 0.0000000000
[206] 0.0000000000 0.0000000000 0.2184658730 0.0000000000 0.0000000000

# compaire Accuracy
performance_metric(pred_prob, test$Class, "malignant", "Accuracy")
[1] 0.9571429
performance_metric(pred_prob, test$Class, "malignant", "Accuracy",
                   cutoff = cutoff)
[1] 0.9714286

# compaire Confusion Matrix
performance_metric(pred_prob, test$Class, "malignant", "ConfusionMatrix")
           actual
predict     benign malignant
  benign       131         1
  malignant      8        70
performance_metric(pred_prob, test$Class, "malignant", "ConfusionMatrix",
                   cutoff = cutoff)
           actual
predict     benign malignant
  benign       134         1
  malignant      5        70

# compaire F1 Score
performance_metric(pred_prob, test$Class, "malignant", "F1_Score")
[1] 0.9395973
performance_metric(pred_prob, test$Class,  "malignant", "F1_Score",
                   cutoff = cutoff)
[1] 0.9589041
performance_metric(pred_prob, test$Class,  "malignant", "F1_Score",
                   cutoff = cutoff2)
[1] 0.9496403
```

If the performance of the tuned cut-off is good, use it as a cut-off to predict positives.

Predict

If you have selected a good model from several models, then perform the prediction with that model.

Create data set for predict

Create sample data for predicting by extracting 100 samples from the data set used in the previous under sampling example.

```
data_pred <- train_under %>%
  cleanse
── Checking unique value ─────────────────────────── unique value is one ──
No variables that unique value is one.
```

```
── Checking unique rate ───────────────────────────── high unique rate ──
remove variables with high unique rate
● Id = 329(0.967647058823529)

── Checking character variables ─────────────────────── categorical data ──
No character variables.

set.seed(1234L)
data_pred <- data_pred %>%
  nrow %>%
  seq %>%
  sample(size = 50) %>%
  data_pred[., ]
```

Predict with alookr and dplyr

Do a predict using the `dplyr` package. The last `factor()` function eliminates unnecessary information.

```
pred_actual <- pred %>%
  filter(model_id == comp_perf$recommend_model) %>%
  run_predict(data_pred) %>%
  select(predicted) %>%
  pull %>%
  "[["(1) %>%
  factor()

pred_actual
 [1] benign    benign    benign    malignant benign    benign    malignant
 [8] malignant benign    malignant benign    malignant benign    benign
[15] benign    benign    benign    benign    malignant benign    benign
[22] malignant malignant benign    malignant malignant malignant benign
[29] benign    benign    benign    malignant malignant benign    malignant
[36] malignant malignant malignant malignant benign    malignant malignant
[43] malignant benign    malignant malignant benign    benign    malignant
[50] benign
Levels: benign malignant
```

If you want to predict by cut-off, specify the `cutoff` argument in the `run_predict()` function as follows.:

In the example, there is no difference between the results of using cut-off and not.

```
pred_actual2 <- pred %>%
  filter(model_id == comp_perf$recommend_model) %>%
  run_predict(data_pred, cutoff) %>%
  select(predicted) %>%
  pull %>%
  "[["(1) %>%
  factor()

pred_actual2
 [1] benign    benign    benign    malignant benign    benign    malignant
 [8] malignant benign    malignant benign    malignant benign    benign
[15] benign    benign    benign    benign    malignant benign    benign
[22] malignant malignant benign    malignant malignant malignant benign
[29] benign    benign    benign    malignant malignant benign    malignant
```

```
[36] malignant malignant malignant malignant benign    malignant malignant
[43] malignant benign    malignant malignant benign    benign    malignant
[50] benign
Levels: benign malignant

sum(pred_actual != pred_actual2)
[1] 0
```