IS6481-Fall2018-Assignment-4: Binary Classifier Bake-Off

Bhuvananjali Challagalla

Estimating and Comparing Binary Classifiers

In today's assignment, we are going to study the performance of some binary classifiers algorithms. Classification algorithms are trained - on the basis of a *training set* of data - in order to be able to identify to which of a set of categories a new observation belongs. Binary classifiers are the simple case in which there are just two set of categories.

There are a myriad of algorithms that are used to solve these classification problems. Today we are going to study `Binary Logistic Regression`, `Naive Bayes Classifier` and a `Tree Model`. And we'll compare their advantages and cons

Our main concern is going to be *overfitting*. One of the most common pitfalls when training Machine Learning algorithms is to overfit your model. Overfitting means that your algorithm is trained in a way that is extremely proficient fitting the model to the observations present in the *training set*, but it shows poor results to otheh observations. Hence the model lacks generalization capabilities. This happens when the model reduces its *bias* drastically, whereas the *variance* balloones.

There are many techniques used in order to prevent overfitting, such as Regularization or Cross-Validation. Today we will focus on assessing the generalization of our algorithms, and so we'll mainly do *Cross-Validation*.

Load Libraries

```
library(tidyverse) # Beautiful Data Munging and visualizations
library(data.table) # Brlazingly fast Data Munging
library(harrypotter) # Pretty Colour Palette
library(mice) # Predictive Imputation of Missing Values

library(rpart)
library(e1071)
library(ROCR)
library(pROC)
```

Retrieve Data: The RMS Titanic Tragedy

We are going to check our predictive skills making use of the very popular dataset of the RMS Titanic Passangers.

The sinking of the RMS Titanic is one of the most infamous shipwrecks in history. On April 15, 1912, during her maiden voyage, the Titanic sank after colliding with an iceberg, killing 1502 out of 2224 passengers and crew. This sensational tragedy shocked the international community and led to better safety regulations for ships.

One of the reasons that the shipwreck led to such loss of life was that there were not enough lifeboats for the passengers and crew. Although there was some element of luck involved in surviving the sinking, some groups of people were more likely to survive than others, such as women, children, and the upper-class.

With this data set we have the opportunity to test which characteristics of the passengers made them more prone to surviving the shipwreck. And it is a great opportunity to test simple binary classification models, in which we'll need to predict whether a passengers survives.

```
load(url("http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic.sav"))
titanic %>% glimpse()
```

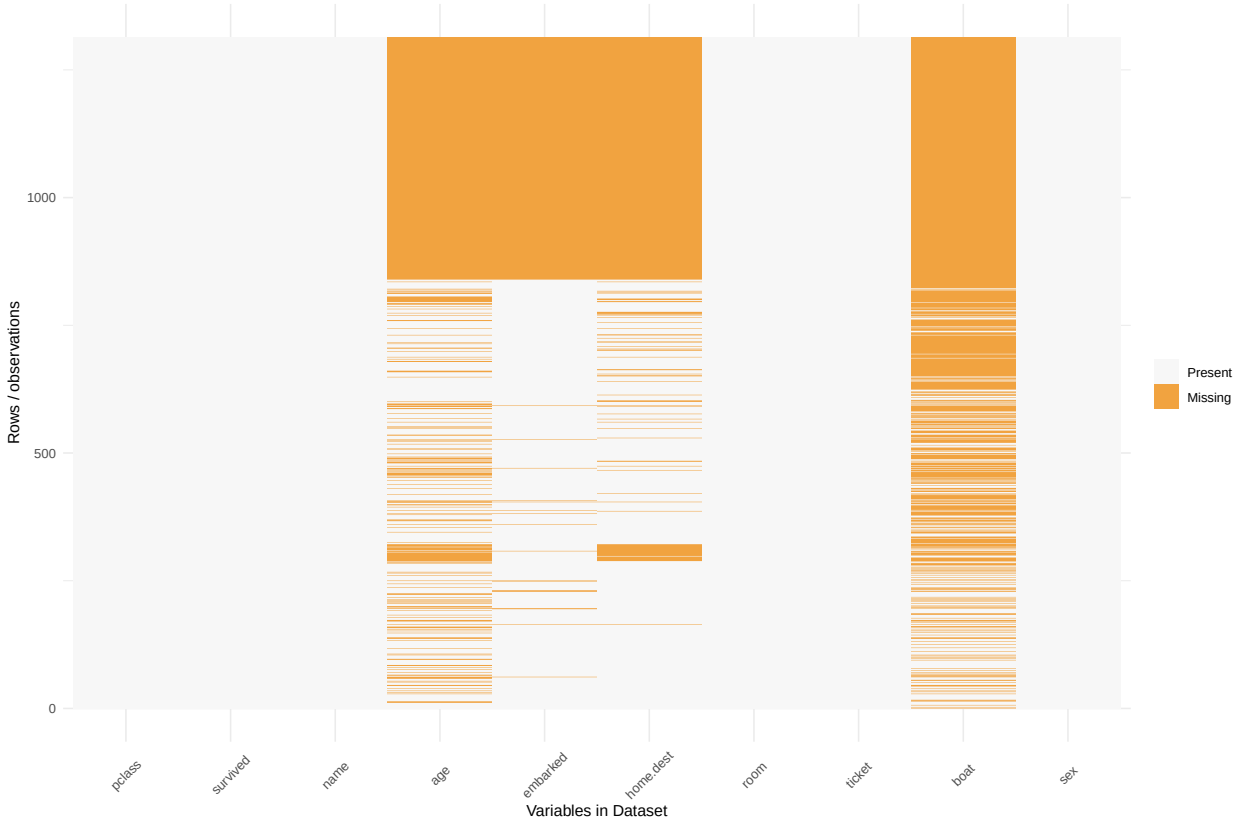
```
## Observations: 1,313
## Variables: 10
## $ pclass    <fct> 1st, 1st, 1st, 1st, 1st, 1st, 1st, 1st, 1st, 1st, 1s...
## $ survived  <int> 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0...
## $ name      <chr> "Allen, Miss Elisabeth Walton", "Allison, Miss Helen...
## $ age       <dbl> 29.0000, 2.0000, 30.0000, 25.0000, 0.9167, 47.0000, ...
## $ embarked  <fct> Southampton, Southampton, Southampton, Southampton, ...
## $ home.dest <fct> St Louis, MO, Montreal, PQ / Chesterville, ON, Montr...
## $ room      <chr> "B-5", "C26", "C26", "C26", "C22", "E-12", "D-7", "A...
## $ ticket    <chr> "24160 L221", "", "", "", "", "", "13502 L77", "", "...
## $ boat      <fct> 2, NA, (135), NA, 11, 3, 10, NA, 2, (22), (124), 4, ...
## $ sex       <fct> female, female, male, female, male, male, female, ma...
```

Missing Data

Dealing with missing values is one of the messiest tasks to address in Data Science. There is a myriad of ways we could face them, and none of them is perfect. The simplest approach would be simply removing all rows containing missing values, but given the small size of the data set, we probably shouldn't do that.

Firstly, we can take a look at where they are:

```
na_map <- function(x){
    x %>%
        is.na() %>%
        melt() %>%
        ggplot(data = .,
                    aes(x = Var2,
                           y = Var1)) +
        geom_raster(aes(fill = value)) +
        theme_minimal() +
        theme(axis.text.x  = element_text(angle=45, vjust=0.5)) +
        labs(x = "Variables in Dataset",
                 y = "Rows / observations") +
        scale_fill_brewer(name = "", labels = c("Present", "Missing"), type = "div", palette = 4, direc
}

titanic %>% na_map()
```



Looking at this map, what we can see is that we have some columns with a huge proportion of missing values.

We could go deep into analyzing the reason for those upper-side rows with such consistent missingness in `age`, `embarked`, `home.dest` and `boat`. But for simplicity's sake, I'd like to make use of the `mice` package and its predictive imputation algorithm. You can read more about its application in `r` here. The problem with this algorithm is that it takes a lot of time to compute. And to keep it quick and simple, I decided to do bootstrap on the missing values.

Firstly, we should note that there are some `NA` values camouflaged as characteres that say `<NA>` or nothing at all `""`. There are very noticeable, but it is far too easy to miss them.

```
hidden_na <- function(x) x %in% c("<NA>", "")

titanic <- titanic %>% mutate_all(funs(ifelse(hidden_na(.), NA, .)))
```

And then we do bootstrap:

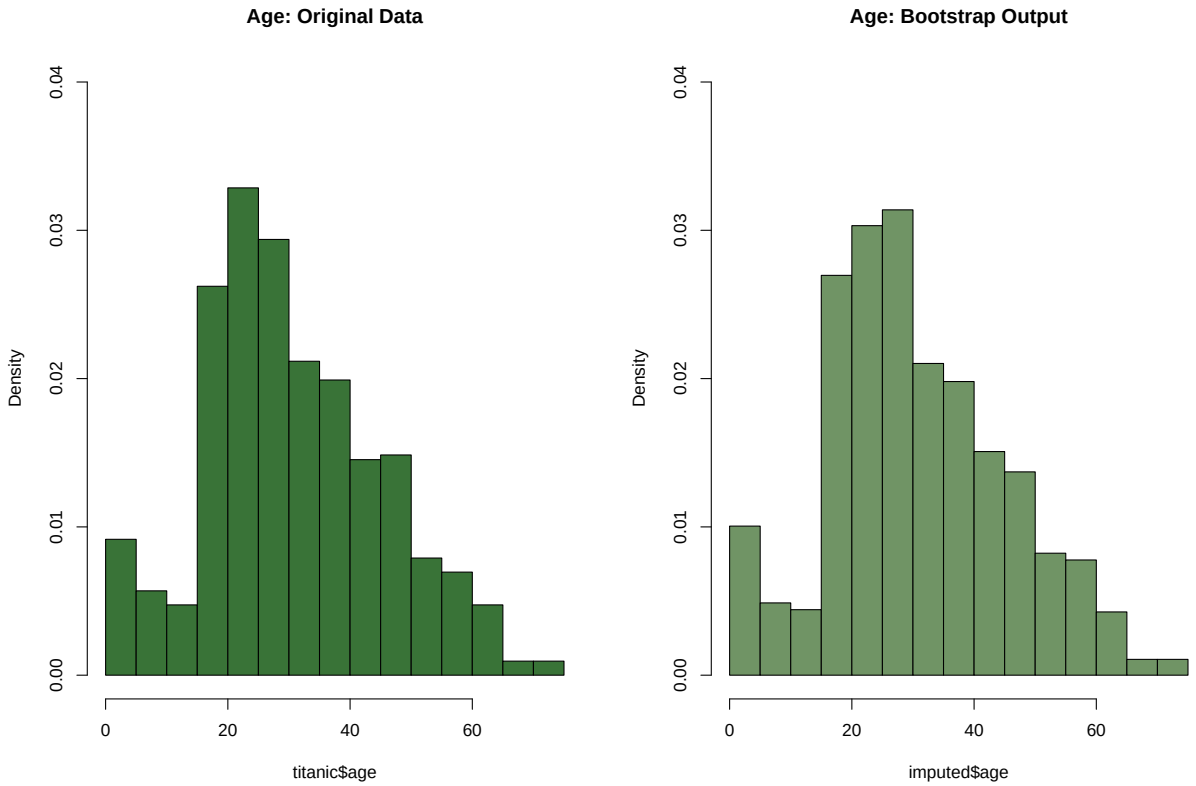
```
na_replace <- function(x){
    if(is.vector(x) & !is.list(x)){
        new_x <- x
        w <- which(is.na(x))
        y <- x[!is.na(x)]
        for(i in w) new_x[i] <- sample(x = y, size = 1, replace = TRUE); cat(paste0("... ", floor(i/leng
        return(new_x)
    }else if(is.data.frame(x)){
        df <- as.data.frame(x)
        ncols <- ncol(df)
        for(i in 1:ncols){
            cat(paste0("... ", floor(i/ncols*100), "% ... \n"))
```

```
            x <- df[i]
            if(sum(is.na(x)) > 0){
                new_x <- x
                w <- which(is.na(x))
                y <- x[!is.na(x)]
                for(k in w) new_x[k,] <- sample(x = y, size = 1, replace = TRUE)
                df[i] <- new_x
            }
        }
        return(df)
    }else if(is.list(x)){
        stop("A list can not be evaluated. Please introduce a vector instead.")
    }else{stop("Unrecognized Format.")}
}

imputed <- na_replace(titanic)
```

```
## ... 10% ...
## ... 20% ...
## ... 30% ...
## ... 40% ...
## ... 50% ...
## ... 60% ...
## ... 70% ...
## ... 80% ...
## ... 90% ...
## ... 100% ...
```

And compare the results obtained with our data. Ideally, they should look similar, since we are just trying to replicate the non-missing data. Let's use `age` as example.

```
par(mfrow=c(1,2))
hist(titanic$age, freq=F, main='Age: Original Data',
  col=hp(10, house = "Slytherin")[[8]], ylim=c(0,0.04))
hist(imputed$age, freq=F, main='Age: Bootstrap Output',
  col=hp(100, house = "Slytherin")[[47]], ylim=c(0,0.04))
```



So, if we are happy, we include this new data

```
titanic <- imputed
```

Feature Engineering

Deck

Looking at the values of the `room` variable, we can see some funny pattern. The first letter of the variable is telling us the `Deck`. So we can make a new variable out of this:

```
titanic$deck <- factor(sapply(titanic$room, function(x) strsplit(x,NULL)[[1]][[1]]))
```

Title

Where the Titanic sinked, society was much more classist than now, and so we can fairly assume that those people with a proper Title in ther name would have had more chances to get help from the security forces of the ship, or maybe access to more robut survival resources. Anyway, we are going to craft a variable trying to stuff this information into a predictive feature.

```
titanic$title <- sub("[[:space:]].*", "", gsub('(.*, )|(\\..*)', '', titanic$name))

titles <- c("Mr", "Rev", "Miss", "Mrs", "Ms", "Dr")

titanic <-
```

```
    titanic %>%
    mutate(title = ifelse(title %in% titles, title, "none"))
```

Dropping useless variables

There are some variables that they are utterly useless. We could do some feature engineering out of them, but see that in their raw form there almost as many unique values as instances.

```
titanic$name %>% unique() %>% length()
```

```
## [1] 1310
```

```
nrow(titanic)
```

```
## [1] 1313
```

```
titanic$name <- NULL
```

Training the Models

We firstly set the variables as factors

```
data <- titanic
features <- colnames(data)

for(f in features) {
    if ((class(data[[f]])=="factor") || (class(data[[f]])=="character")) {
        levels <- unique(data[[f]])
        data[[f]] <- (factor(data[[f]], levels=levels))
    }
}
titanic <- data
rm(data);gc()
```

```
##           used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells 2172924 116.1    3774137 201.6  3774137 201.6
## Vcells 5007659  38.3   10146329  77.5  8388608  64.0
```

```
titanic %>% glimpse()
```

```
## Observations: 1,313
## Variables: 11
## $ pclass    <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
## $ survived  <int> 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0...
## $ age       <dbl> 29.0000, 2.0000, 30.0000, 25.0000, 0.9167, 47.0000, ...
## $ embarked  <int> 3, 3, 3, 3, 3, 3, 3, 3, 3, 1, 1, 1, 1, 3, 3, 1, 1, 1...
## $ home.dest <int> 311, 232, 232, 232, 232, 238, 163, 25, 23, 230, 238,...
## $ room      <fct> B-5, C26, C26, C26, C22, E-12, D-7, A-36, C-101, D-7...
## $ ticket    <fct> 24160 L221, 13502 L77, 111361 L57 19s 7d, 7076, 1113...
## $ boat      <int> 87, 82, 12, 71, 79, 88, 78, 82, 87, 34, 8, 89, 95, 9...
## $ sex       <int> 1, 1, 2, 1, 2, 2, 1, 2, 1, 2, 2, 1, 1, 2, 2, 1, 2, 2...
## $ deck      <fct> B, C, C, C, C, E, D, A, C, D, B, C, B, A, C, B, B, C...
## $ title     <fct> Miss, Miss, Mr, Mrs, none, Mr, Miss, none, Mrs, Mr, ...
```

And then we convert

Split into Training & Test sets

```
tr_id <- sample(1:nrow(titanic), nrow(titanic)*0.7)
train <- titanic[tr_id,]
test  <- titanic[-tr_id,]
```

Binary Logistic Regression

```
blr_model <- glm(survived ~., family=binomial(link='logit'), data = train)

blr_preds_test <- ifelse(predict(blr_model,test, type = "response") > 0.5, 1, 0)
blr_preds_train <- ifelse(predict(blr_model,train, type = "response") > 0.5, 1, 0)

tr_acc <- round(1 - mean(blr_preds_train != train$survived), 4)
te_acc <- round(1 - mean(blr_preds_test  != test$survived), 4)
print(paste('Train Set Accuracy: ', tr_acc))
```

```
## [1] "Train Set Accuracy:  0.852"
```

```
print(paste('Test Set Accuracy: ', te_acc))
```

```
## [1] "Test Set Accuracy:  0.7665"
```

In general, the *Train Set Accuracy* is always to be higher that the test set. We'll say that our model is overfitting if it is *much* higher. How much is that? There is no clear cut answer. But we can measure the *degree of overfitting* that our model is doing, by computing the ratio between both accuracies:

```
oc_blr <- round(100*(tr_acc - te_acc)/tr_acc,6)
print(paste0("Overfitting Coefficient: ", oc_blr))
```

```
## [1] "Overfitting Coefficient: 10.035211"
```

The greater this number, the worst is the overfitting.

Naive Bayes Classifier

```
nbc_model <- naiveBayes(as.factor(survived)~., data = train)

nbc_preds_test <- ifelse(predict(nbc_model,test, type = "raw")[,2] > 0.5, 1, 0)
nbc_preds_train <- ifelse(predict(nbc_model,train, type = "raw")[,2] > 0.5, 1, 0)

tr_acc <- round(1 - mean(nbc_preds_train != train$survived), 4)
te_acc <- round(1 - mean(nbc_preds_test  != test$survived), 4)
print(paste('Train Set Accuracy: ', tr_acc))
```

```
## [1] "Train Set Accuracy:  0.802"
```

```
print(paste('Test Set Accuracy: ', te_acc))
```

```
## [1] "Test Set Accuracy:  0.7919"
```

```
oc_nbc <- round(100*(tr_acc - te_acc)/tr_acc,6)
print(paste0("Overfitting Coefficient: ", oc_nbc))
```

```
## [1] "Overfitting Coefficient: 1.259352"
```

Recursive Partitioning Tree Model

```
rpart_model <- rpart(as.factor(survived)~., data = train)

rpart_preds_test <- ifelse(predict(rpart_model,test)[,2] > 0.5, 1, 0)
rpart_preds_train <- ifelse(predict(rpart_model,train)[,2] > 0.5, 1, 0)

tr_acc <- round(1 - mean(rpart_preds_train != train$survived), 4)
te_acc <- round(1 - mean(rpart_preds_test  != test$survived), 4)
print(paste('Train Set Accuracy: ', tr_acc))
```

```
## [1] "Train Set Accuracy:  0.8607"
```

```
print(paste('Test Set Accuracy: ', te_acc))
```

```
## [1] "Test Set Accuracy:  0.7487"
```

```
oc_rpart <- round(100*(tr_acc - te_acc)/tr_acc,6)
print(paste0("Overfitting Coefficient: ", oc_rpart))
```

```
## [1] "Overfitting Coefficient: 13.012664"
```

Comparing Results

So far we've seen the results of all three models and we can have a rough idea of what model we would prefer. However, I think measuring this once can be mesleading, for the values of the overfitting coefficient may vary considerable just because of randomness. In order to smooth out this uncertainty, I decided to iterate this process 100 times and measure the obtained Overfitting Coefficient and Test Accuracy at each iteration. This way we can assess the performance of every model properly.

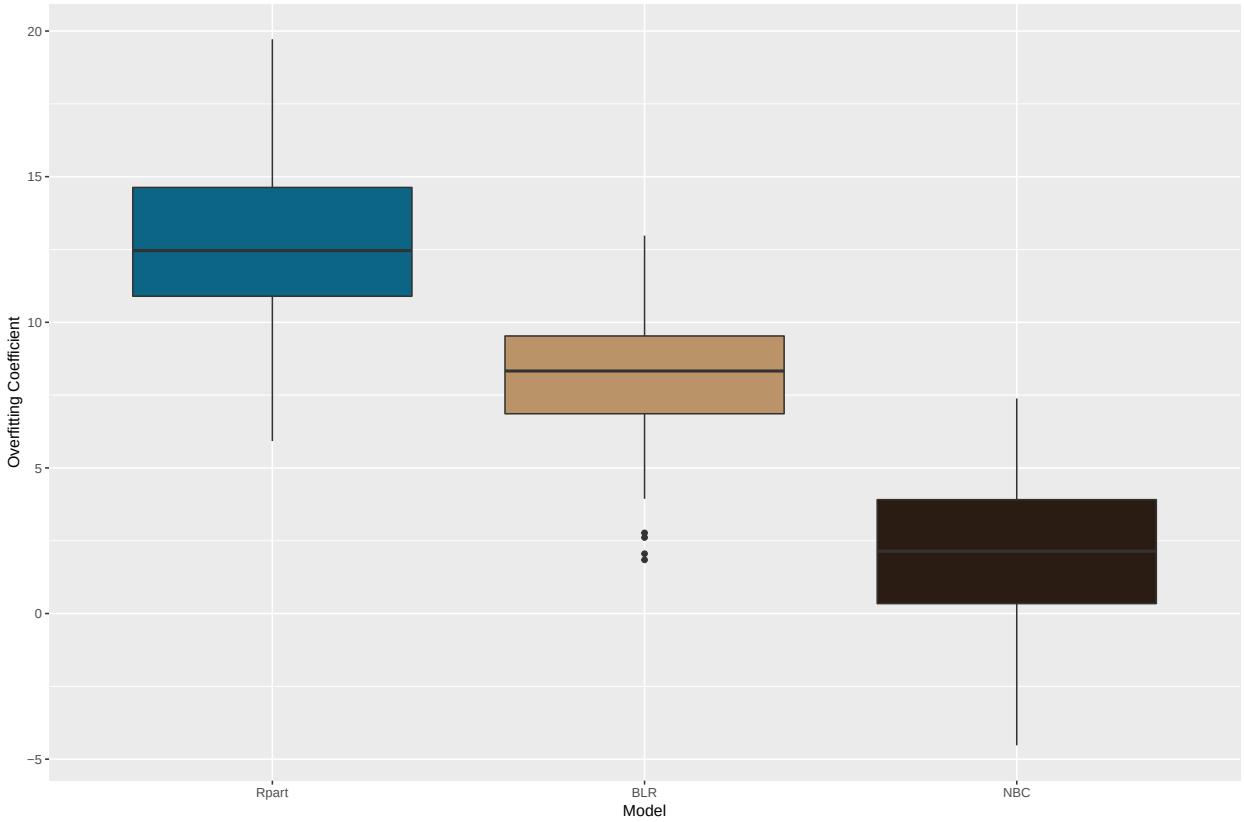
```
compute_accuracy <- function(model){
    if(model == "rpart"){
        rpart_model <- rpart(as.factor(survived)~., data = train)

rpart_preds_test <- ifelse(predict(rpart_model,test)[,2] > 0.5, 1, 0)
rpart_preds_train <- ifelse(predict(rpart_model,train)[,2] > 0.5, 1, 0)

tr_acc <- round(1 - mean(rpart_preds_train != train$survived), 4)
te_acc <- round(1 - mean(rpart_preds_test  != test$survived), 4)
    }else if(model == "blr"){
        blr_model <- glm(survived ~., family=binomial(link='logit'), data = train)

blr_preds_test <- ifelse(predict(blr_model,test, type = "response") > 0.5, 1, 0)
blr_preds_train <- ifelse(predict(blr_model,train, type = "response") > 0.5, 1, 0)

tr_acc <- round(1 - mean(blr_preds_train != train$survived), 4)
te_acc <- round(1 - mean(blr_preds_test  != test$survived), 4)
    }else{
nbc_model <- naiveBayes(as.factor(survived)~., data = train)

nbc_preds_test <- predict(nbc_model,test)
nbc_preds_train <- predict(nbc_model,train)

tr_acc <- round(1 - mean(nbc_preds_train != train$survived), 4)
```

```
te_acc <- round(1 - mean(nbc_preds_test  != test$survived), 4)
    }
oc <- round(100*(tr_acc - te_acc)/tr_acc,6)
return(list(oc = oc,tr_acc = tr_acc,te_acc = te_acc))
}
```

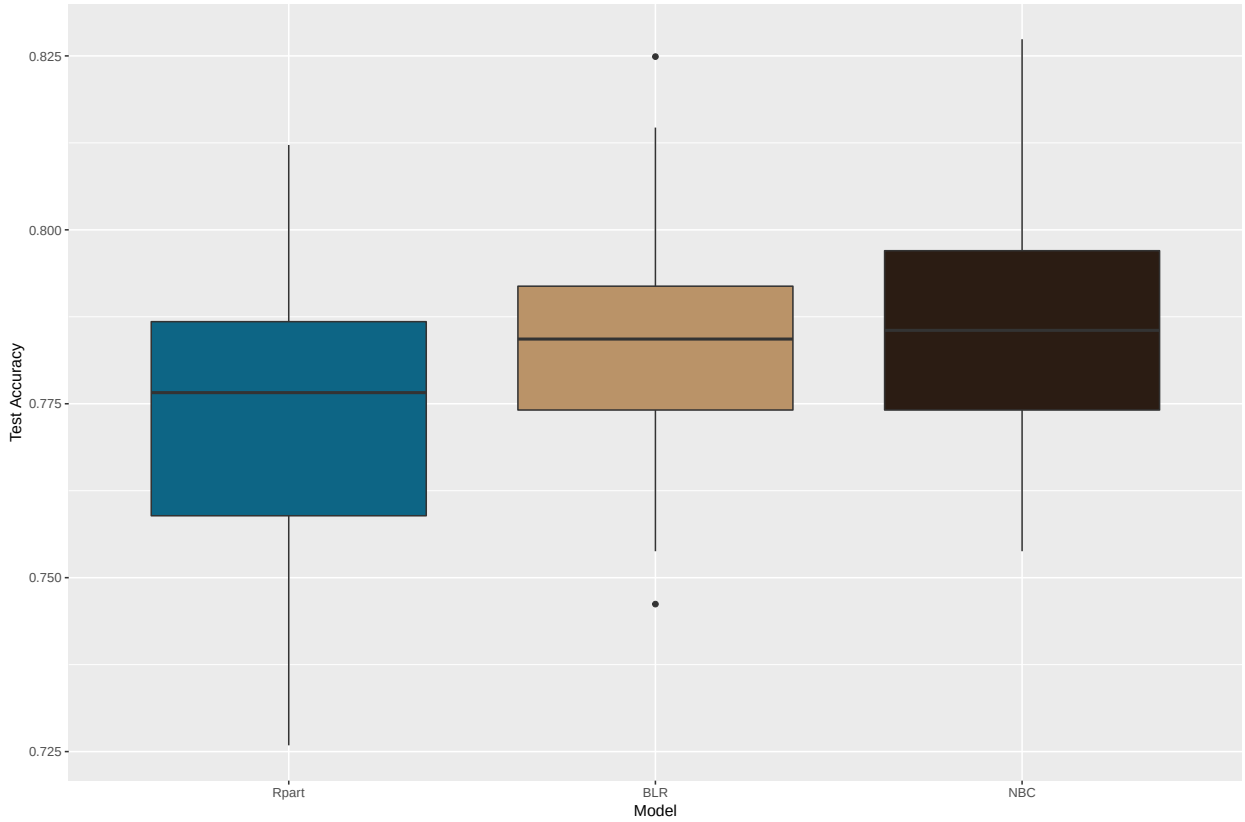
```
n_iters <- 100

rpart_oc <- c()
blr_oc <- c()
nbc_oc <- c()

rpart_te <- c()
blr_te <- c()
nbc_te <- c()
for(i in 1:n_iters){
    tr_id <- sample(1:nrow(titanic), nrow(titanic)*0.7)
    train <- titanic[tr_id,]
    test  <- titanic[-tr_id,]

    rpart_oc[[i]] <- compute_accuracy(model = "rpart")[["oc"]]
    blr_oc[[i]]   <- compute_accuracy(model = "blr")[["oc"]]
    nbc_oc[[i]]   <- compute_accuracy(model = "nbc")[["oc"]]

    rpart_te[[i]] <- compute_accuracy(model = "rpart")[["te_acc"]]
    blr_te[[i]]   <- compute_accuracy(model = "blr")[["te_acc"]]
    nbc_te[[i]]   <- compute_accuracy(model = "nbc")[["te_acc"]]
}
```

And visualize the results in handy boxplots

```
tibble(Rpart = rpart_oc, BLR = blr_oc, NBC = nbc_oc) %>%
    melt() %>%
    ggplot(aes(x = variable, y = value)) +
    geom_boxplot(aes(fill = variable)) +
    scale_fill_hp(discrete = TRUE, house = "Ravenclaw") +
    theme(legend.position="none") +
    xlab("Model") +
    ylab("Overfitting Coefficient")
```




```
tibble(Rpart = rpart_te, BLR = blr_te, NBC = nbc_te) %>%
    melt() %>%
    ggplot(aes(x = variable, y = value)) +
    geom_boxplot(aes(fill = variable)) +
    scale_fill_hp(discrete = TRUE, house = "Ravenclaw") +
    theme(legend.position="none") +
    xlab("Model") +
    ylab("Test Accuracy")
```



Now we can fairly say that the *Naive Bayes Classifier* tends to less overfitting than the others. And consequently its Test Accuracy tends to be higher.

ROC Analysis and AUC

Now we are going to repeat the same analysis, but for the `AUC` The `AUC` is another interesting metric. Sometimes even preferred in binary classification instead of the Accuracy, for a number of different reasons. First let's talk about what `AUC` actually is.

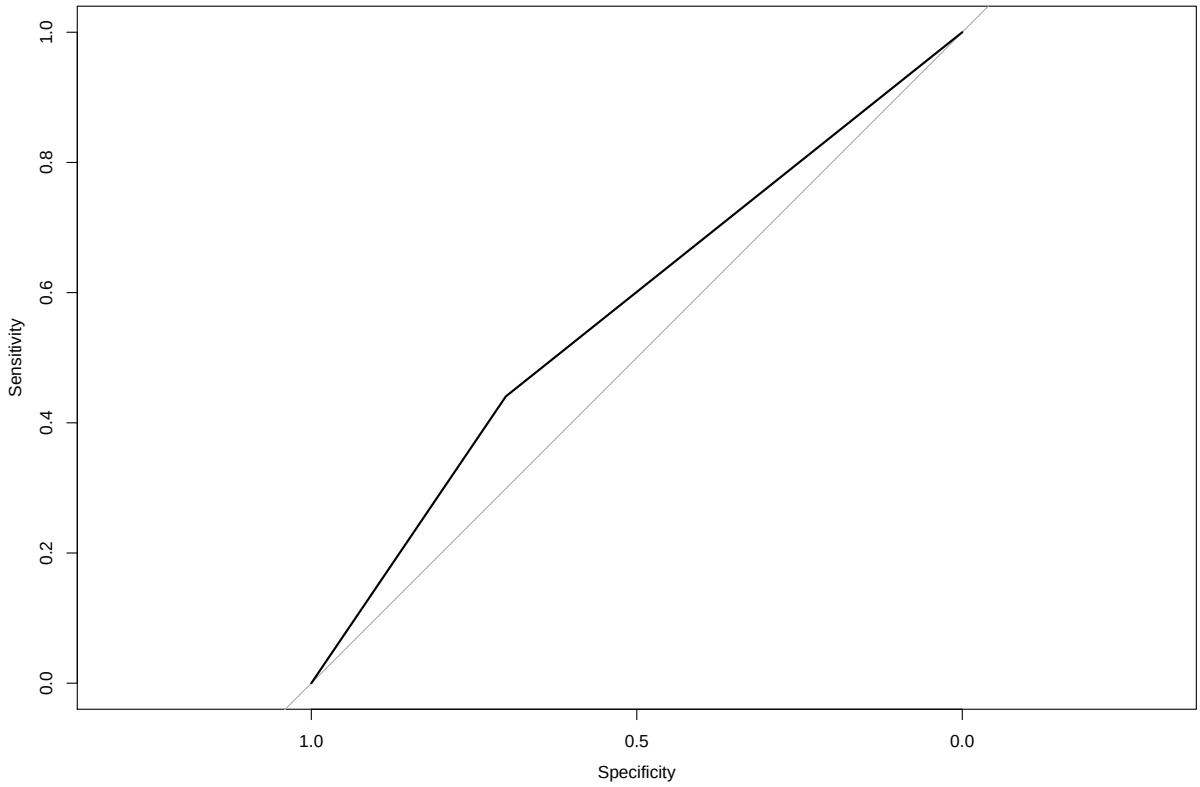
`AUC` stands for `Area Under the Curve`. Which curve? The `ROC` curve. Which stands for `Receiver Operating Characteristic`. The implicit goal of `AUC` is to deal with situations where you have a very skewed sample distribution, and don't want to overfit to a single class.

A great example of this is in illness detection. Imagine that you are building a model that its objective is to predict illnesses. Whatever data set you may have, it is going to be *extremely* biased towards `non-sick`. Let's be honest, most people stay in perfectly health conditions the vast majority of their time. If your data is 99% healthy people, you can get a pretty damn good accuracy just by stating "*everyone's healthy!*"; and that would be actually accurate. But, would be useful a model that states the obvious but does not tell you anything you didn't already know?

Back to the `ROC` curve. The `ROC` curve is drawn using two metrics. The `TPR` (True Positive Rate), and the `FPR` (False Positive Rate). The `ROC` curve is obtained by drawing a plot of the `TPR` vs `FPR`. Hence, if our model would draw a complete diagonal line, that would mean that our model is no better than guessing randomly between `0` and `1`. What would result in the *area* under that line to be 0.5. On the other hand, a ROC curve that shadows an area of 1 is the perfect ideal.

An example:

```r
plot.roc(test$survived,nbc_preds_test)
```




```r
print(paste0("AUC = ", auc(test$survived, nbc_preds_test)))
```

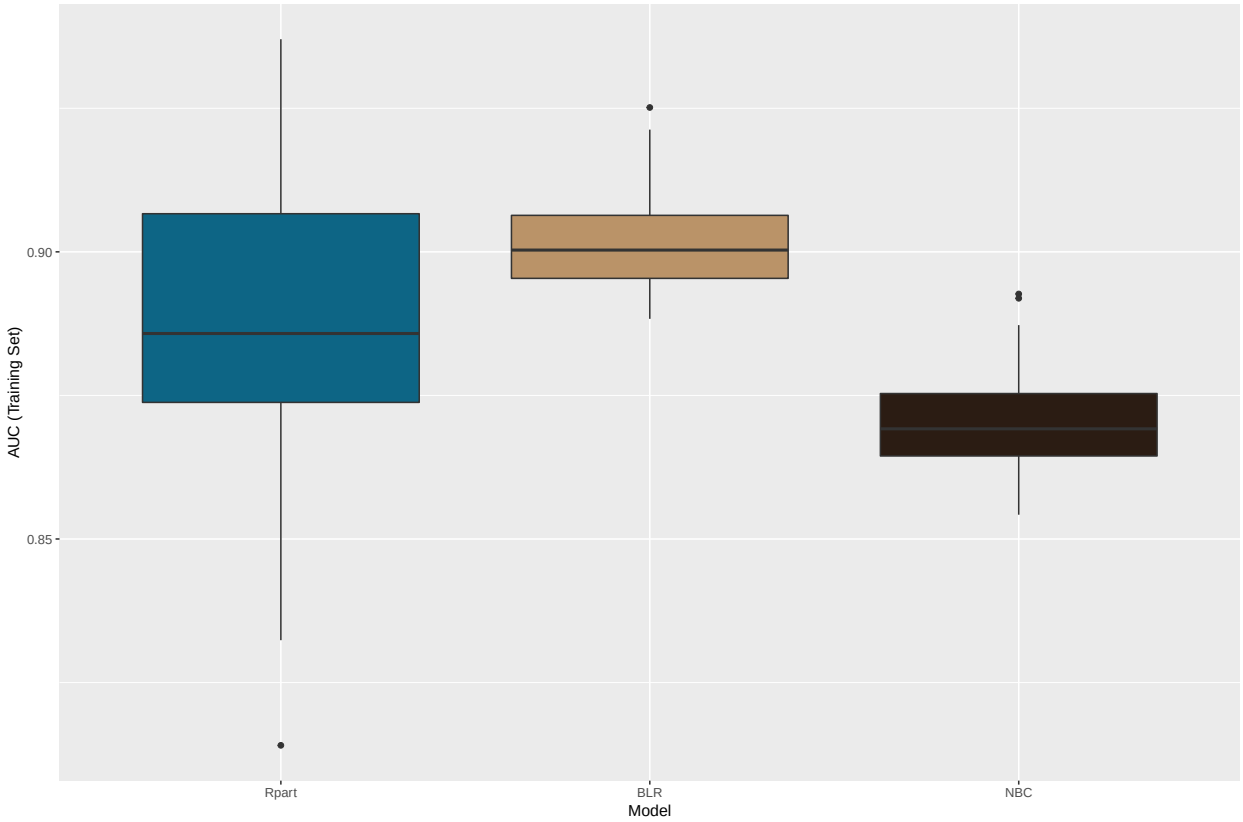
```
## [1] "AUC = 0.570877329841473"
```

```r
compute_auc <- function(model){
    if(model == "rpart"){
        rpart_model <- rpart(as.factor(survived)~., data = train)

        rpart_preds_test <-  predict(rpart_model,test)[,2]
        rpart_preds_train <- predict(rpart_model,train)[,2]

        te_auc <- auc(test$survived,rpart_preds_test)
        tr_auc <- auc(train$survived,rpart_preds_train)
    }else if(model == "blr"){
        blr_model <- glm(survived ~., family=binomial(link='logit'), data = train)

        blr_preds_test <-  predict(blr_model,test, type = "response")
        blr_preds_train <- predict(blr_model,train, type = "response")

        te_auc <- auc(test$survived,blr_preds_test)
        tr_auc <- auc(train$survived,blr_preds_train)
    }else{
        nbc_model <- naiveBayes(as.factor(survived)~., data = train)

        nbc_preds_test <- predict(nbc_model,test, type = "raw")[,2]
        nbc_preds_train <- predict(nbc_model,train, type = "raw")[,2]
```

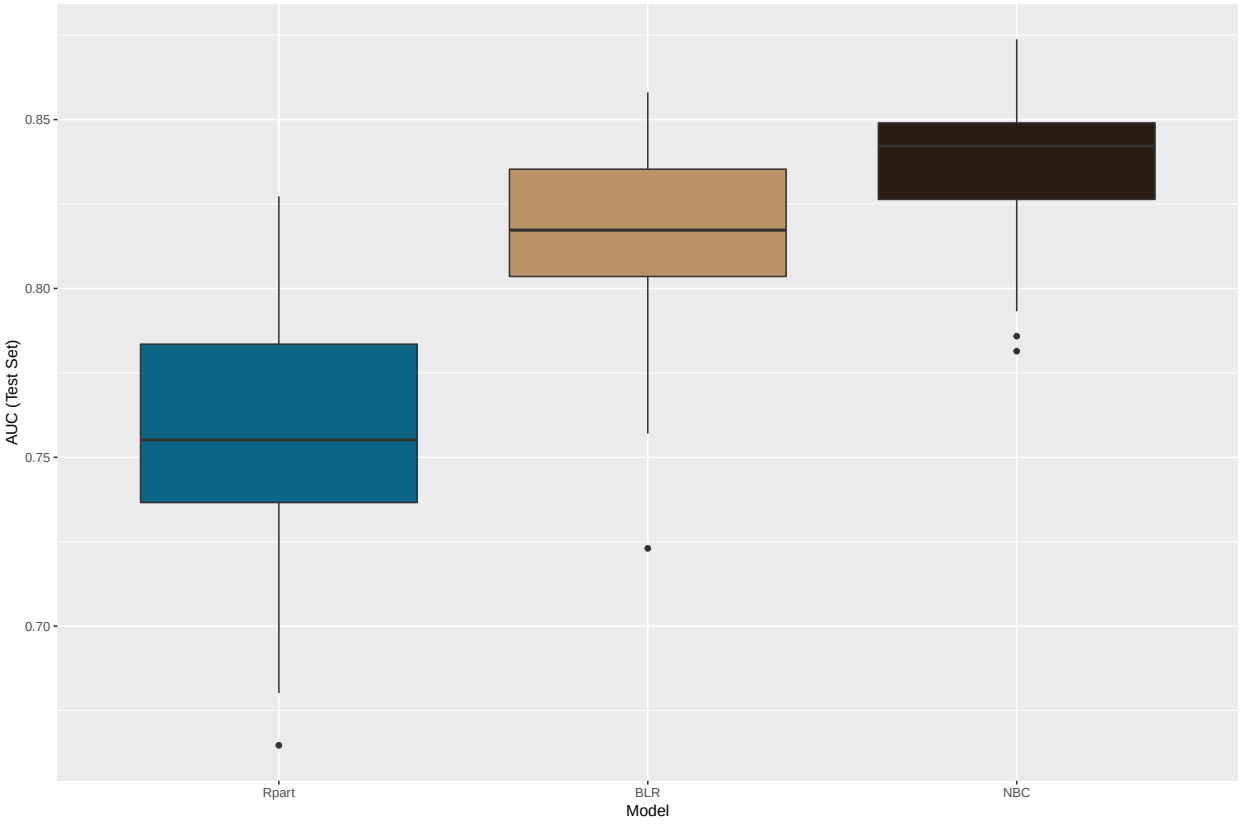
```
        te_auc <- auc(test$survived,nbc_preds_test)
        tr_auc <- auc(train$survived,nbc_preds_train)
    }
return(list(tr_auc = tr_auc,te_auc = te_auc))
}
```

```
rpart_auc_tr <- c()
blr_auc_tr   <- c()
nbc_auc_tr   <- c()

rpart_auc_te <- c()
blr_auc_te   <- c()
nbc_auc_te   <- c()

n_iters   <- 100

for(i in 1:n_iters){
    tr_id <- sample(1:nrow(titanic), nrow(titanic)*0.7)
    train <- titanic[tr_id,]
    test  <- titanic[-tr_id,]

    rpart_auc_tr[[i]] <- compute_auc(model = "rpart")[["tr_auc"]]
    blr_auc_tr[[i]]   <- compute_auc(model = "blr")[["tr_auc"]]
    nbc_auc_tr[[i]]   <- compute_auc(model = "nbc")[["tr_auc"]]

    rpart_auc_te[[i]] <- compute_auc(model = "rpart")[["te_auc"]]
    blr_auc_te[[i]]   <- compute_auc(model = "blr")[["te_auc"]]
    nbc_auc_te[[i]]   <- compute_auc(model = "nbc")[["te_auc"]]
}
```

```
tibble(Rpart = rpart_auc_tr, BLR = blr_auc_tr, NBC = nbc_auc_tr) %>%
    melt() %>%
    ggplot(aes(x = variable, y = value)) +
    geom_boxplot(aes(fill = variable)) +
    scale_fill_hp(discrete = TRUE, house = "Ravenclaw") +
    theme(legend.position="none") +
    xlab("Model") +
    ylab("AUC (Training Set)")
```




```
tibble(Rpart = rpart_auc_te, BLR = blr_auc_te, NBC = nbc_auc_te) %>%
    melt() %>%
    ggplot(aes(x = variable, y = value)) +
    geom_boxplot(aes(fill = variable)) +
    scale_fill_hp(discrete = TRUE, house = "Ravenclaw") +
    theme(legend.position="none") +
    xlab("Model") +
    ylab("AUC (Test Set)")
```


0.85
0.80
0.75
0.70
AUC (Test Set)
Rpart
BLR
NBC
Model