# Kernel and Ensemble Methods

# Ensemble Techniques

CS 4375 - Intro to Machine Learning

Dr. Karen Mazidi

Author:

- Leo Nguyen - ldn190002
- Cory Pekkala - cdp190005

# Skin Segmentation Data Set

**Data Set Information**

- The skin dataset is collected by randomly sampling B,G,R values from face images of various age groups (young, middle, and old), race groups (white, black, and asian), and genders obtained from FERET database and PAL database. Total learning sample size is 245057; out of which 50859 is the skin samples and 194198 is non-skin samples. Color FERET Image Database: [Web Link], PAL Face Database from Productive Aging Laboratory, The University of Texas at Dallas: [Web Link].

**Citation:**

- Rajen Bhatt, Abhinav Dhall, 'Skin Segmentation Dataset', UCI Machine Learning Repository
- https://archive.ics.uci.edu/ml/datasets/Skin+Segmentation (https://archive.ics.uci.edu/ml/datasets/Skin+Segmentation)

**Attribute Information:**

- This dataset is of the dimension 245057 * 4 where first three columns are B,G,R (x1,x2, and x3 features) values and fourth column is of the class labels (decision variable y).

## Load the data

```
df_origin <- read.csv("data/Skin Segmentation.csv", header=TRUE)
str(df_origin)
```

```
## 'data.frame':    245057 obs. of  4 variables:
##  $ B   : int  74 73 72 70 70 69 70 70 76 76 ...
##  $ G   : int  85 84 83 81 81 80 81 81 87 87 ...
##  $ R   : int  123 122 121 119 119 118 119 119 125 125 ...
##  $ Skin: int  1 1 1 1 1 1 1 1 1 1 ...
```

```
dim(df_origin)
```

```
## [1] 245057      4
```

# Data Cleaning

**Create a subsample by produce a subset randomly from a data frame**

- Just randomly pick up 10000 observations to create a subset

```
set.seed(1234)
df <- df_origin[sample(1:nrow(df_origin), 10000, replace = FALSE),]
str(df)
```

```
## 'data.frame':    10000 obs. of  4 variables:
##  $ B   : int  183 12 102 37 170 89 59 50 63 20 ...
##  $ G   : int  180 13 149 0 185 91 58 54 57 17 ...
##  $ R   : int  135 3 223 146 231 39 20 19 14 2 ...
##  $ Skin: int  2 2 1 2 1 2 2 2 2 2 ...
```

```
dim(df)
```

```
## [1] 10000     4
```

- Convert column Skin to factor, and label them as Skin and NonSkin

```
df$Skin <- factor(df$Skin)
levels(df$Skin) <- c("Skin", "NonSkin")
str(df)
```

```
## 'data.frame':    10000 obs. of  4 variables:
##  $ B   : int  183 12 102 37 170 89 59 50 63 20 ...
##  $ G   : int  180 13 149 0 185 91 58 54 57 17 ...
##  $ R   : int  135 3 223 146 231 39 20 19 14 2 ...
##  $ Skin: Factor w/ 2 levels "Skin","NonSkin": 2 2 1 2 1 2 2 2 2 2 ...
```

# Divide into train, test (80/20)

```
set.seed(1234)
i <- sample(1:nrow(df), 0.8*nrow(df), replace=FALSE)
train <- df[i,]
test <- df[-i,]
```

# Data Exploration

1. View the summary of entire training data set

```
summary(train)
```

```
##        B               G               R              Skin
##  Min.   :  0.0   Min.   :  0.0   Min.   :  0.0   Skin   :1674
##  1st Qu.: 69.0   1st Qu.: 87.0   1st Qu.: 73.0   NonSkin:6326
##  Median :140.0   Median :153.0   Median :127.0
##  Mean   :125.4   Mean   :132.3   Mean   :123.1
##  3rd Qu.:176.0   3rd Qu.:177.0   3rd Qu.:164.0
##  Max.   :255.0   Max.   :255.0   Max.   :255.0
```

2. Skin factor encoding

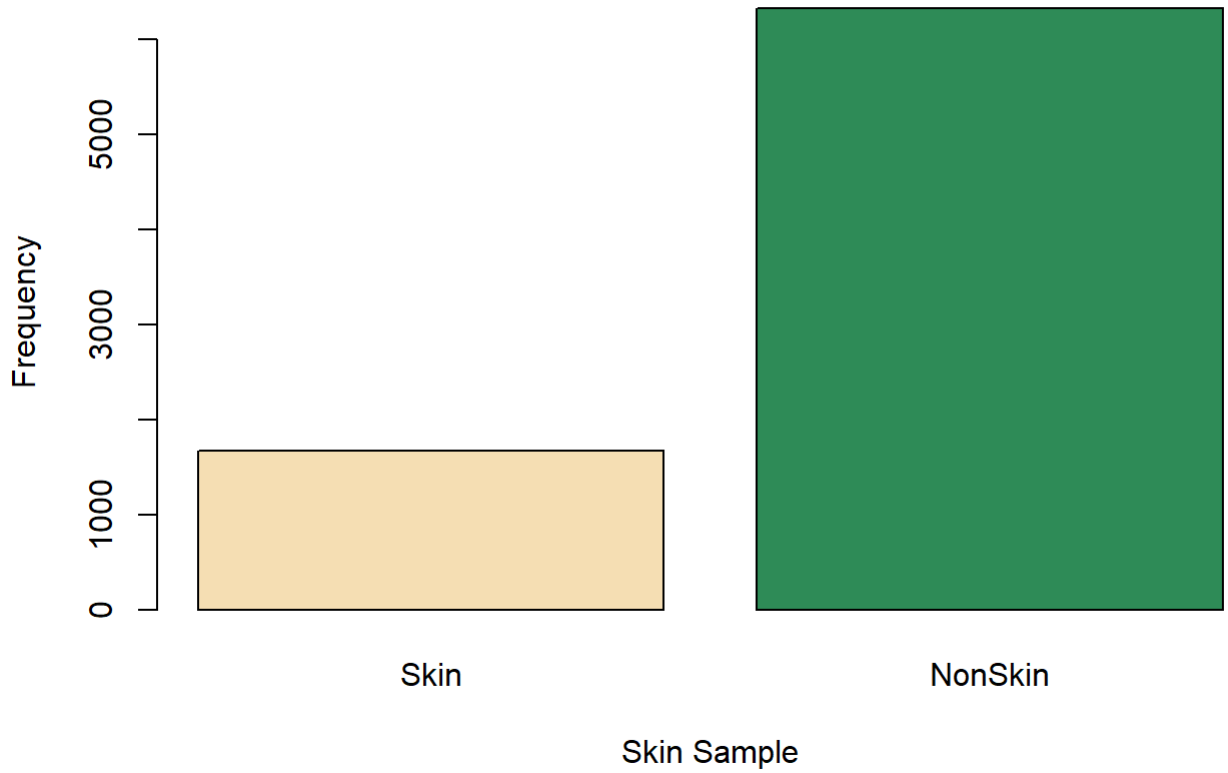
```
contrasts(train$Skin)
```

```
##         NonSkin
## Skin          0
## NonSkin       1
```

# Data Visualization using informative graph

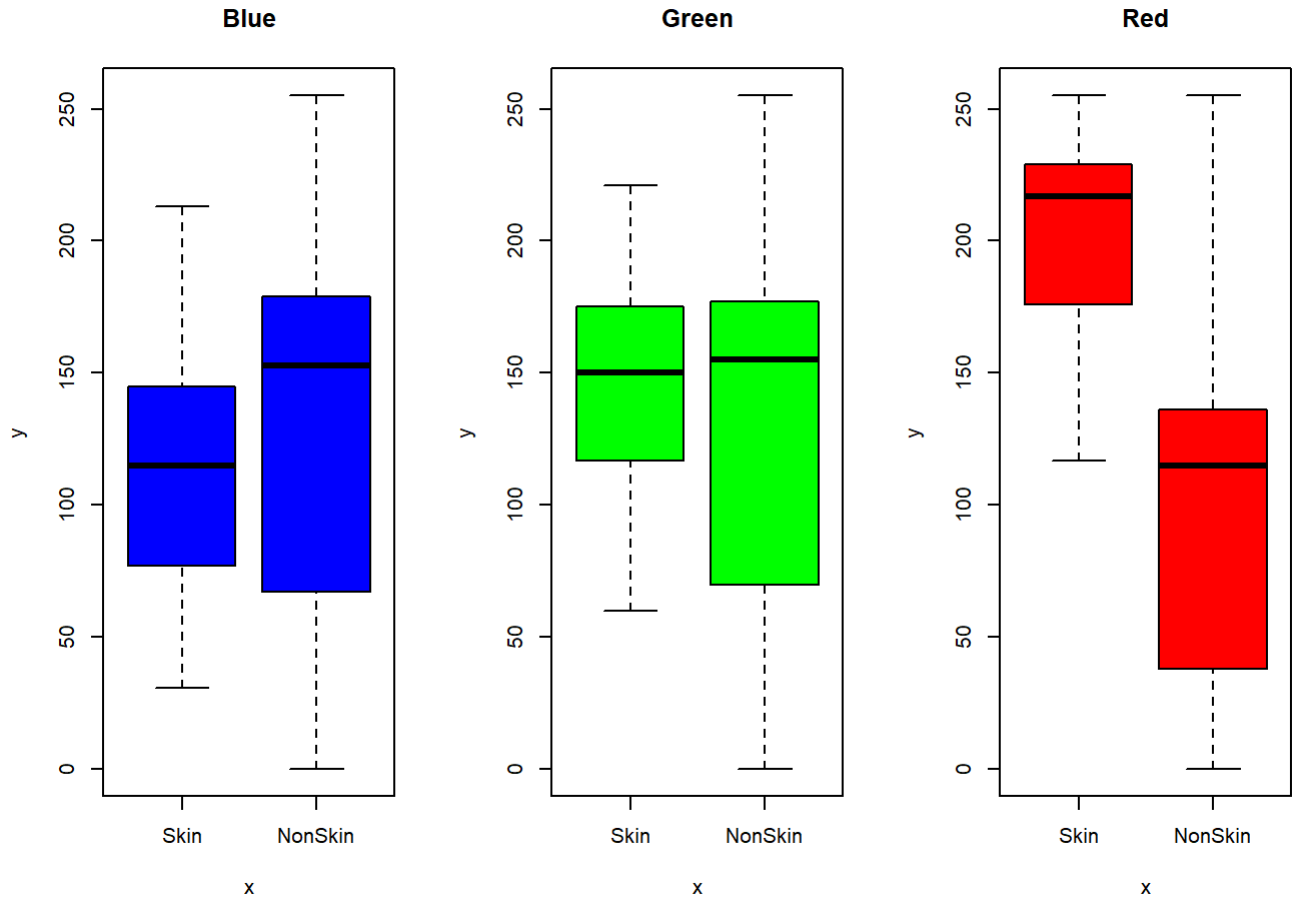
1. Using barplots to visualize the 2 factor level of Skin Sample

```
counts <- table(train$Skin)
barplot(counts, xlab="Skin Sample", ylab="Frequency", col=c("wheat", "seagreen"))
```



2. Visualize the relationship between Skin and B, G, R

```
par(mfrow=c(1,3))
plot(train$Skin, train$B, data = train, main = "Blue", col="blue")
plot(train$Skin, train$G, data = train, main = "Green", col="green")
plot(train$Skin, train$R, data = train, main = "Red", col="red")
```




# Decision Tree

**Install mltools: Machine Learning Tool**

- This package will be used to calculate the MATTHEW'S CORRELATION COEFFICIENT (mcc) which an important to indicate the differences in class distribution
- You can install the package by typing install.packages("mltools") on the console

```
library(mltools)
```

**Building Decision Tree as baseline model**

```
library(tree)
tree <- tree(Skin~., data=train)
tree
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 8000 8207.00 NonSkin ( 0.2092500 0.7907500 )
##     2) R < 170.5 6121 2821.00 NonSkin ( 0.0612645 0.9387355 )
##       4) B < 108.5 2312 2043.00 NonSkin ( 0.1613322 0.8386678 )
##         8) R < 116.5 1875    0.00 NonSkin ( 0.0000000 1.0000000 ) *
##         9) R > 116.5 437  364.00 Skin ( 0.8535469 0.1464531 )
##          18) G < 48 38    0.00 NonSkin ( 0.0000000 1.0000000 ) *
##          19) G > 48 399  192.30 Skin ( 0.9348371 0.0651629 ) *
##       5) B > 108.5 3809   34.21 NonSkin ( 0.0005251 0.9994749 ) *
##     3) R > 170.5 1879 2323.00 Skin ( 0.6913252 0.3086748 )
##       6) G < 103.5 208    0.00 NonSkin ( 0.0000000 1.0000000 ) *
##       7) G > 103.5 1671 1772.00 Skin ( 0.7773788 0.2226212 )
##        14) G < 195.5 1365  849.50 Skin ( 0.9062271 0.0937729 )
##          28) B < 179.5 1328  663.50 Skin ( 0.9314759 0.0685241 )
##            56) B < 46.5 25   13.94 NonSkin ( 0.0800000 0.9200000 ) *
##            57) B > 46.5 1303  534.00 Skin ( 0.9478127 0.0521873 ) *
##          29) B > 179.5 37    0.00 NonSkin ( 0.0000000 1.0000000 ) *
##        15) G > 195.5 306  308.50 NonSkin ( 0.2026144 0.7973856 )
##          30) B < 214.5 146  199.10 NonSkin ( 0.4246575 0.5753425 )
##            60) R < 237.5 61    0.00 NonSkin ( 0.0000000 1.0000000 ) *
##            61) R > 237.5 85   99.25 Skin ( 0.7294118 0.2705882 )
##             122) B < 143 22    0.00 NonSkin ( 0.0000000 1.0000000 ) *
##             123) B > 143 63   10.27 Skin ( 0.9841270 0.0158730 ) *
##          31) B > 214.5 160    0.00 NonSkin ( 0.0000000 1.0000000 ) *
```

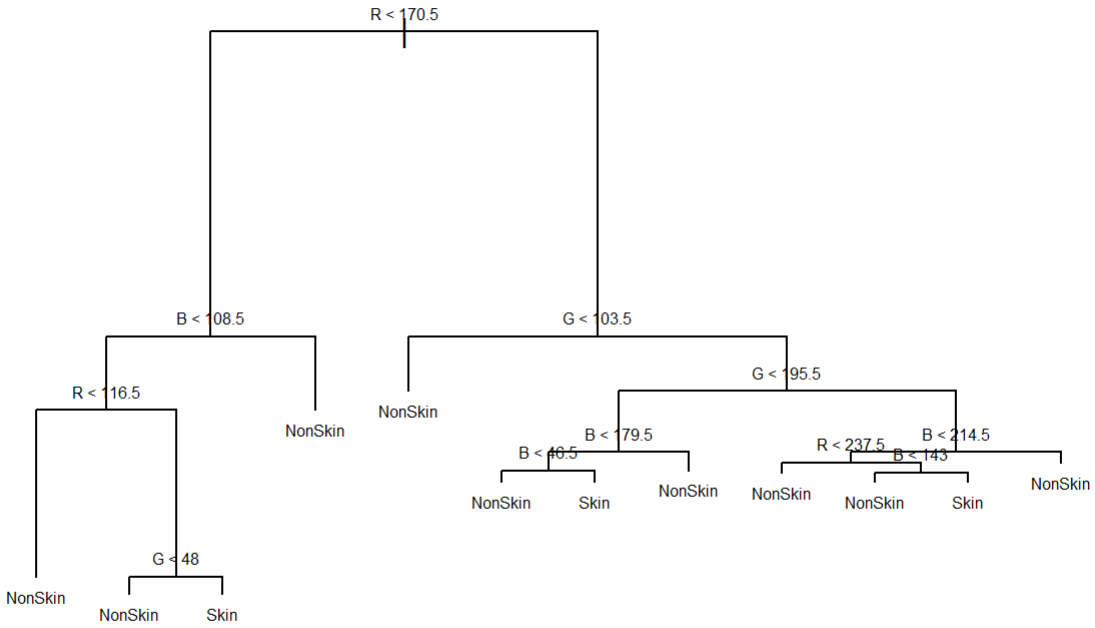
```
summary(tree)
```

```
##
## Classification tree:
## tree(formula = Skin ~ ., data = train)
## Number of terminal nodes:  12
## Residual mean deviance:  0.09823 = 784.7 / 7988
## Misclassification error rate: 0.01238 = 99 / 8000
```

### Plotting Decision Tree

- Plotting Decision Tree to get a visualization on the model

```
plot(tree)
text(tree, cex=0.5, pretty=0)
```




### Evaluate Decision Tree

```
startTime <- Sys.time()
pred_dt <- predict(tree, newdata = test, type="class")
endTime <- Sys.time()

table(pred_dt, test$Skin)
```

```
##
## pred_dt   Skin NonSkin
##   Skin     442      30
##   NonSkin    2    1526
```

```
acc_dt <- mean(pred_dt==test$Skin)
mcc_dt <- mcc(pred_dt, test$Skin)
runtime_dt <- endTime - startTime

print(paste('Accuracy:', acc_dt))
```

```
## [1] "Accuracy: 0.984"
```

```
print(paste('mcc:', mcc_dt))
```

```
## [1] "mcc: 0.955452740408774"
```

```
print(paste('Runtime in seconds:', runtime_dt))
```

```
## [1] "Runtime in seconds: 0.0510590076446533"
```

- The Decision Tree (DT) get a very high accuracy = 0.984. This indicate DT is very good model. Moreover, the mcc=0.95545 is also very high which further more confirm that. It is because, the accuracy by itself is not work well when the data is in balance which is the case of our dataset (mentioned in the dataset description above: 50859 is the skin samples and 194198 is non-skin samples). In this case, the accuracy alone will provide an overoptimistic estimation based on the majority class. By combining the mcc into the evaluation we can double check and verify this overoptimistic estimation. The mcc=0.95545 very close to 1 which indicate the total agreement between test data and predict data

# Random Forest

#### Building Random Forest Model

```
library(randomForest)
```

```
## randomForest 4.7-1.1
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
set.seed(1234)
rf <- randomForest(Skin~., data=train, importance=TRUE)
rf
```

```
##
## Call:
##  randomForest(formula = Skin ~ ., data = train, importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 0.16%
## Confusion matrix:
##         Skin NonSkin class.error
## Skin    1671       3 0.001792115
## NonSkin   10    6316 0.001580778
```

```
summary(rf)
```

```
##                 Length Class  Mode
## call               4  -none- call
## type               1  -none- character
## predicted        8000  factor numeric
## err.rate         1500  -none- numeric
## confusion           6  -none- numeric
## votes           16000  matrix numeric
## oob.times        8000  -none- numeric
## classes             2  -none- character
## importance         12  -none- numeric
## importanceSD        9  -none- numeric
## localImportance     0  -none- NULL
## proximity           0  -none- NULL
## ntree               1  -none- numeric
## mtry                1  -none- numeric
## forest             14  -none- list
## y                8000  factor numeric
## test                0  -none- NULL
## inbag               0  -none- NULL
## terms               3  terms  call
```

### Evaluate Random Forest

```
startTime <- Sys.time()
pred_rf <- predict(rf, newdata = test, type="response")
endTime <- Sys.time()

table(pred_rf, test$Skin)
```

```
##
## pred_rf   Skin NonSkin
##   Skin     444       2
##   NonSkin    0    1554
```

```
acc_rf <- mean(pred_rf==test$Skin)
mcc_rf <- mcc(pred_rf, test$Skin)
runtime_rf <- endTime - startTime

print(paste('Accuracy:', acc_rf))
```

```
## [1] "Accuracy: 0.999"
```

```
print(paste('mcc:', mcc_rf))
```

```
## [1] "mcc: 0.99711389114366"
```

```
print(paste('Runtime in seconds:', runtime_rf))
```

```
## [1] "Runtime in seconds: 0.0711369514465332"
```

- As an ensemble model,the Random Forest (RF) improve the accuracy and mcc further more by using parallel ensemble method-bagging with DT as individual model. Both accuracy=0.999 and mcc=0.99711 is very high, they are almost 100%. When looking at the RT model summary, we can see that by using 500 different DT, we can get different value on accuracy and mcc. Then when combine all this prediction together, we can achieve a final prediction which much higher accuracy and mcc compare to base model.

# XGBoost

### Building XGBoost

```
library(xgboost)
train_label <- ifelse(as.integer(train$Skin)==2, 1, 0)
train_matrix <- data.matrix(train[, -4])
xg <- xgboost(data=train_matrix, label=train_label,
                nrounds=100, objective='binary:logistic')
```

```
## [1]  train-logloss:0.449872
## [2]  train-logloss:0.315487
## [3]  train-logloss:0.232375
## [4]  train-logloss:0.176998
## [5]  train-logloss:0.136066
## [6]  train-logloss:0.102109
## [7]  train-logloss:0.078668
## [8]  train-logloss:0.061593
## [9]  train-logloss:0.049342
## [10] train-logloss:0.039979
## [11] train-logloss:0.032988
## [12] train-logloss:0.025679
## [13] train-logloss:0.020309
## [14] train-logloss:0.016618
## [15] train-logloss:0.013840
## [16] train-logloss:0.012151
## [17] train-logloss:0.010465
## [18] train-logloss:0.008853
## [19] train-logloss:0.007646
## [20] train-logloss:0.006842
## [21] train-logloss:0.006251
## [22] train-logloss:0.005747
## [23] train-logloss:0.005401
## [24] train-logloss:0.005071
## [25] train-logloss:0.004848
## [26] train-logloss:0.004533
## [27] train-logloss:0.004185
## [28] train-logloss:0.003879
## [29] train-logloss:0.003642
## [30] train-logloss:0.003448
## [31] train-logloss:0.003266
## [32] train-logloss:0.003128
## [33] train-logloss:0.002983
## [34] train-logloss:0.002870
## [35] train-logloss:0.002811
## [36] train-logloss:0.002722
## [37] train-logloss:0.002630
## [38] train-logloss:0.002530
## [39] train-logloss:0.002445
## [40] train-logloss:0.002391
## [41] train-logloss:0.002339
## [42] train-logloss:0.002289
## [43] train-logloss:0.002247
## [44] train-logloss:0.002195
## [45] train-logloss:0.002140
## [46] train-logloss:0.002094
## [47] train-logloss:0.002055
## [48] train-logloss:0.002008
## [49] train-logloss:0.001978
## [50] train-logloss:0.001951
## [51] train-logloss:0.001917
## [52] train-logloss:0.001892
```

```
## [53] train-logloss:0.001871
## [54] train-logloss:0.001845
## [55] train-logloss:0.001827
## [56] train-logloss:0.001799
## [57] train-logloss:0.001780
## [58] train-logloss:0.001758
## [59] train-logloss:0.001737
## [60] train-logloss:0.001720
## [61] train-logloss:0.001698
## [62] train-logloss:0.001677
## [63] train-logloss:0.001650
## [64] train-logloss:0.001635
## [65] train-logloss:0.001619
## [66] train-logloss:0.001599
## [67] train-logloss:0.001584
## [68] train-logloss:0.001566
## [69] train-logloss:0.001553
## [70] train-logloss:0.001539
## [71] train-logloss:0.001527
## [72] train-logloss:0.001519
## [73] train-logloss:0.001503
## [74] train-logloss:0.001491
## [75] train-logloss:0.001479
## [76] train-logloss:0.001464
## [77] train-logloss:0.001449
## [78] train-logloss:0.001438
## [79] train-logloss:0.001427
## [80] train-logloss:0.001412
## [81] train-logloss:0.001403
## [82] train-logloss:0.001394
## [83] train-logloss:0.001383
## [84] train-logloss:0.001375
## [85] train-logloss:0.001366
## [86] train-logloss:0.001357
## [87] train-logloss:0.001349
## [88] train-logloss:0.001339
## [89] train-logloss:0.001334
## [90] train-logloss:0.001325
## [91] train-logloss:0.001320
## [92] train-logloss:0.001313
## [93] train-logloss:0.001306
## [94] train-logloss:0.001301
## [95] train-logloss:0.001297
## [96] train-logloss:0.001289
## [97] train-logloss:0.001284
## [98] train-logloss:0.001277
## [99] train-logloss:0.001272
## [100]    train-logloss:0.001265
```

```
xg
```

```
## ##### xgb.Booster
## raw: 135.6 Kb
## call:
##   xgb.train(params = params, data = dtrain, nrounds = nrounds,
##     watchlist = watchlist, verbose = verbose, print_every_n = print_every_n,
##     early_stopping_rounds = early_stopping_rounds, maximize = maximize,
##     save_period = save_period, save_name = save_name, xgb_model = xgb_model,
##     callbacks = callbacks, objective = "binary:logistic")
## params (as set within xgb.train):
##   objective = "binary:logistic", validate_parameters = "TRUE"
## xgb.attributes:
##   niter
## callbacks:
##   cb.print.evaluation(period = print_every_n)
##   cb.evaluation.log()
## # of features: 3
## niter: 100
## nfeatures : 3
## evaluation_log:
##     iter train_logloss
##        1   0.449871958
##        2   0.315487388
## ---
##       99   0.001271546
##      100   0.001264606
```

### Evaluate XGBoost

```
startTime <- Sys.time()
test_label <- ifelse(as.integer(test$Skin)==2, 1, 0)
test_matrix <- data.matrix(test[, -4])
probs <- predict(xg, test_matrix)
pred <- ifelse(probs>0.5, 1, 0)
endTime <- Sys.time()

acc_xg <- mean(pred==test_label)
mcc_xg <- mcc(pred, test_label)
runtime_xg <- endTime - startTime

print(paste("accuracy=", acc_xg))
```

```
## [1] "accuracy= 0.9985"
```

```
print(paste("mcc=", mcc_xg))
```

```
## [1] "mcc= 0.995677412505583"
```

```
print(paste('Runtime in seconds:', runtime_xg))
```

```
## [1] "Runtime in seconds: 0.0167930126190186"
```

- The XGBoost is also an ensemble method, that is why we can see improvement on accuracy=0.9985 and mcc=0.99567 compare to DT. However, it is slightly lower than the Random Forest. This could be because in this specific XGBoost model, we only use 100 iterations which is much less than 500 trees in Random Forest. It mean we still have room to improve the accuracy and mcc by increasing the number of iteration when building the model. Due to it unique algorithm, the XGBoost also the fast model in all 4 example models: Decision Tree, Random Forest, AdaBoost and XGBoost. It has the lowest runtime among 4 models.

# AdaBoost

**Building Adaboost**

```
library(adabag)
```

```
## Loading required package: rpart
```

```
## Loading required package: caret
```

```
## Loading required package: ggplot2
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked from 'package:randomForest':
##
##     margin
```

```
## Loading required package: lattice
```

```
## Loading required package: foreach
```

```
## Loading required package: doParallel
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
adab1 <- boosting(Skin~., data=train, boos=TRUE, mfinal=20, coeflearn='Breiman')
summary(adab1)
```

```
##            Length Class   Mode
## formula         3  formula call
## trees          20  -none-  list
## weights        20  -none-  numeric
## votes       16000  -none-  numeric
## prob        16000  -none-  numeric
## class        8000  -none-  character
## importance      3  -none-  numeric
## terms           3  terms   call
## call            6  -none-  call
```

**Evaluate Adaboost**

```
startTime <- Sys.time()
pred_adabag <- predict(adab1, newdata = test, type="response")
endTime <- Sys.time()

table(pred_adabag$class, test$Skin)
```

```
##
##            Skin NonSkin
##   NonSkin    0    1554
##   Skin     444       2
```

```
acc_adabag <- mean(pred_adabag$class==test$Skin)
mcc_adabag <- mcc(factor(pred_adabag$class), test$Skin)
runtime_adabag <- endTime - startTime

print(paste('Accuracy:', acc_adabag))
```

```
## [1] "Accuracy: 0.999"
```

```
print(paste('mcc:', mcc_adabag))
```

```
## [1] "mcc: 0.99711389114366"
```

```
print(paste('Runtime in seconds:', runtime_adabag))
```

```
## [1] "Runtime in seconds: 0.167912006378174"
```

- The AdaBoost also achieve a very high accuracy=0.999 and mcc=0.99711 which is no surprise there. It is because AdaBoost is an ensemble model. However, it use different methods compare to the Random Forest(parallel), AdaBoost is sequential. There is some trade off to achieve this high accuracy and mcc value, the AdaBoost the is the lowest model among 4. It is because of its natural, a sequential method.

# Conclusion

- The Decision Tree can provide a good accuracy and good mcc value. By using other ensemble model we can achieve an even higher accuracy and mcc value. Then using the DT as a base model, we can see that Random Forest, XGBoost and AdaBoost provide a much higher accuracy and mcc value. However, there is a trade off on the runtime. If the ensemble model is parallel (like in Random Forest), we can get both good accuracy and good runtime. If the model is sequenstial (like AdaBoost), we can get a good accuracy but the runtime increase compare to base model (Decision Tree). Another case is XGBoost model. It is the fastest model with lowest runtime compare from both base model and ensemble model. However, there some trade off as the accuracy and mcc slightly reduce compare with other ensemble model, but it is still higher than base model.