Prediction Challenge

```r
# -----------------------
# Bigdata challenge

# Load data
bigdata_train <- read.csv(paste0(file_path, "ST310_2023_bigdata_train.csv"))
bigdata_test <- read.csv(paste0(file_path, "ST310_2023_bigdata_test.csv"))

#Create matrix objects if needed
y_train <- as.matrix(bigdata_train$y)
x_train <- as.matrix(bigdata_train[, -1]) # leave out y
x_test <- as.matrix(bigdata_test) # doesn't come with y
```

```r
#Split training data into 'train' and 'test' for cross-validation
prop_train <- 8/10 # proportion that is training data
n <- nrow(bigdata_train)
#Sample the indexes for training and test data
train <- sample(1:n, floor(prop_train*n), replace = F)
test <- setdiff(1:n, train)
bigdata_cv_train <- bigdata_train[train, ]
bigdata_cv_test <- bigdata_train[test, ]
```

```r
# Baseline linear regression model
lm_model <- lm(y ~ ., data = bigdata_cv_train) # train model
lm_predict <- predict(lm_model, newdata = bigdata_cv_test[, -1]) # predict values
lm_mse <- mean((lm_predict - bigdata_cv_test$y)^2) # calculate mean squared error
lm_mse
```

```
## [1] 24.70178
```

```r
#General additive model
num_predictors <- ncol(bigdata_train) - 1
formula_string <- "y ~"
#Create formula by appending " + s(variable)" to the end
for (i in 2:(num_predictors + 1)) { #
  formula_string <- paste(formula_string, "+ s(", colnames(bigdata_train)[i], ")", sep = "")
}
gam_model <- gam(as.formula(formula_string), data = bigdata_cv_train) # train model
gam_predict <- predict(gam_model, newdata = bigdata_cv_test[, -1]) # get predicted values
gam_mse <- mean((gam_predict - bigdata_cv_test$y)^2) # calculate mean squared error
gam_mse
```

```
## [1] 12.13144
```

```r
#Random forest model
train_ctrl <- trainControl(method = "cv", number = 5) # set cross-validation controls
#Train the model
rf_model <- train(y ~ .,
                  data = bigdata_cv_train,
                  method = "rf",
                  trControl = train_ctrl
                  )

rf_predict <- predict(rf_model, newdata = bigdata_cv_test[, -1]) # get predicted values
rf_mse <- mean((rf_predict - bigdata_cv_test$y)^2) # calculate mse
rf_mse
```

```
## [1] 4.243187
```

```r
#Boosted trees model
#Convert data into DMatrix (necessary for xgboost)
bt_train_cv <- xgb.DMatrix(data = as.matrix(bigdata_cv_train[, -1]),
                           label = bigdata_cv_train$y)
bt_test_cv <- xgb.DMatrix(data = as.matrix(bigdata_cv_test[, -1]),
                          label = bigdata_cv_test$y)
#Set list of parameters for the model
params <- list(
  objective = "reg:squarederror",  # objective function which minimises mean squared error
  eval_metric = "rmse",
  eta = 0.05, # learning rate
  max_depth = 6 # maximum depth of each tree
)

#Perform cross validation to select optimal number of iterations
cv_results <- xgb.cv(
  params = params,
  data = bt_train_cv,
  nfold = 10, # number of folds for k-fold cross validation
  nrounds = 800, # number of rounds over which to perform CV
  early_stopping_rounds = 20, # CV will stop after this many rounds of no improvement
  verbose = 0
)
cv_results
```

```
## ##### xgb.cv 10-folds
##     iter train_rmse_mean train_rmse_std test_rmse_mean test_rmse_std
##        1       9.0427886     0.02535521       9.049844     0.2188426
##        2       8.6786564     0.02599467       8.695703     0.2147709
##        3       8.3351894     0.02744485       8.361567     0.2148423
##        4       8.0110224     0.02834296       8.047646     0.2119503
##        5       7.7053945     0.02952966       7.750308     0.2094285
## ---
##      796       0.5405056     0.01909922       2.153892     0.1606242
##      797       0.5396864     0.01860379       2.153824     0.1606270
##      798       0.5385845     0.01844967       2.153720     0.1606308
##      799       0.5377449     0.01854641       2.153690     0.1606812
##      800       0.5370553     0.01844390       2.153728     0.1606789
```

```
## Best iteration:
##  iter train_rmse_mean train_rmse_std test_rmse_mean test_rmse_std
##   799       0.5377449     0.01854641        2.15369     0.1606812
```

```r
bt_model <- xgboost(params = params,
                    data = bt_train_cv,
                    nrounds = cv_results$best_iteration,
                    verbose = 0
                    )
bt_predict <- predict(bt_model, newdata = bt_test_cv)
bt_mse <- mean((bt_predict - bigdata_cv_test$y)^2)
bt_mse
```

```
## [1] 4.173157
```

```r
# Choose the best model (boosted trees)
# This time we are training on the full training data set
bt_train <- xgb.DMatrix(data = x_train, label = y_train)
bt_test <- xgb.DMatrix(data = x_test)
bigdata_best <- xgboost(params = params,
                        data = bt_train,
                        nrounds = cv_results$best_iteration,
                        verbose = 0
                        )

# Save best predictions
bigdata_test_y <- data.frame(y = predict(bigdata_best, newdata = x_test))
names(bigdata_test_y) <- "y"

# Check for correct number of predictions
nrow(bigdata_test_y) == nrow(bigdata_test)
```

```
## [1] TRUE
```

```r
# Check that they have the same format
cbind(head(bigdata_test_y$y), head(bigdata_train$y))
```

```
##           [,1]  [,2]
## [1,] 18.138050  0.00
## [2,]  7.455416  3.64
## [3,]  5.167072 17.07
## [4,]  2.784262  7.38
## [5,]  6.896758 11.19
## [6,]  5.565750 10.59
```

```r
# -----------------------
# High-dim. challenge

# Load data
highdim_train <- read.csv(paste0(file_path, "ST310_2023_highdim_train.csv"))
highdim_test <- read.csv(paste0(file_path, "ST310_2023_highdim_test.csv"))
```

```r
#Create matrix objects
y_train <- highdim_train[, 1]
x_train <- as.matrix(highdim_train[, -1])
x_test <- as.matrix(highdim_test)
#Split into train and test data
n <- nrow(highdim_train)
train <- sample(1:n, floor(prop_train*n), replace = F)
test <- setdiff(1:n, train)
highdim_cv_train <- highdim_train[train, ]
highdim_cv_test <- highdim_train[test, ]
```

```r
# Ridge model
# Perform cross-validation for the optimal penalty value
# alpha = 0 makes this a ridge regression
cv_ridge <- cv.glmnet(x = x_train, y = y_train, alpha = 0)
best_lambda <- cv_ridge$lambda.min
#Train model with the optimal value of lambda
ridge_model <- glmnet(x = as.matrix(highdim_cv_train[, -1]),
                      y = highdim_cv_train$y,
                      alpha = 0,
                      lambda = best_lambda
                      )
# Get predicted values
ridge_predict <- predict(ridge_model, newx = as.matrix(highdim_cv_test[, -1]))
ridge_mse <- mean((ridge_predict - highdim_cv_test$y)^2) # calculate mean sqaured error
ridge_mse
```

```
## [1] 77.23012
```

```r
#Lasso model (same procedure as for ridge)
# alpha = 1 makes this a lasso regression
cv_lasso <- cv.glmnet(x = x_train, y = y_train, alpha = 1)
best_lambda <- cv_lasso$lambda.min
lasso_model <- glmnet(x = as.matrix(highdim_cv_train[, -1]),
                      y = highdim_cv_train$y,
                      alpha = 1,
                      lambda = best_lambda
                      )
lasso_predict <- predict(lasso_model, newx = as.matrix(highdim_cv_test[, -1]))
lasso_mse <- mean((lasso_predict - highdim_cv_test$y)^2)
lasso_mse
```

```
## [1] 28.37717
```

```r
#Random forest model (same procedure as for the bigdata challenge)
train_ctrl <- trainControl(method = "cv", number = 5)
rf_highdim <- train(y ~ .,
                  data = highdim_cv_train,
                  method = "rf",
                  trControl = train_ctrl
                  )
```

```r
rf_predict <- predict(rf_highdim, newdata = highdim_cv_test[, -1])
rf_mse <- mean((rf_predict - highdim_cv_test$y)^2)
rf_mse
```

```
## [1] 91.88073
```

```r
#Boosted trees model (same procedure as for the bigdata challenge)
bt_train_cv <- xgb.DMatrix(data = as.matrix(highdim_cv_train[, -1]),
                           label = highdim_cv_train$y)
bt_test_cv <- xgb.DMatrix(data = as.matrix(highdim_cv_test[, -1]),
                          label = highdim_cv_test$y)
params <- list(
  objective = "reg:squarederror",
  eval_metric = "rmse",
  eta = 0.5,
  max_depth = 6,
  verbose = 0
)


cv_results <- xgb.cv(
  params = params,
  data = bt_train_cv,
  nfold = 10,
  nrounds = 800,
  early_stopping_rounds = 50,
  verbose = 0
)
cv_results
```

```
## ##### xgb.cv 10-folds
##  iter train_rmse_mean train_rmse_std test_rmse_mean test_rmse_std
##     1    6.1110995351   1.931794e-01       9.478609      1.824921
##     2    4.1159374130   1.948184e-01       9.476407      1.941530
##     3    2.7616187814   1.024903e-01       9.591259      1.910402
##     4    1.9201081413   1.495347e-01       9.571114      1.895088
##     5    1.3121370729   8.702021e-02       9.461321      1.869674
##     6    0.9646394595   7.863962e-02       9.406983      1.889862
##     7    0.7089294954   6.652889e-02       9.405973      1.899884
##     8    0.5346240803   4.831110e-02       9.401981      1.889459
##     9    0.3915134934   3.505238e-02       9.382675      1.897782
##    10    0.3064857978   2.990114e-02       9.386391      1.895010
##    11    0.2292464844   3.156855e-02       9.384138      1.893085
##    12    0.1717155169   2.317038e-02       9.376315      1.887426
##    13    0.1317108445   2.387046e-02       9.373818      1.890239
##    14    0.0993895294   2.035968e-02       9.372300      1.885242
##    15    0.0736160956   1.502154e-02       9.369481      1.889333
##    16    0.0537540575   9.012396e-03       9.369650      1.890590
##    17    0.0419485543   8.211299e-03       9.367417      1.890134
##    18    0.0315625492   6.598962e-03       9.366293      1.889721
##    19    0.0238267237   6.229897e-03       9.366344      1.889602
##    20    0.0176325484   4.213917e-03       9.364733      1.889046
##    21    0.0133248537   3.377019e-03       9.364854      1.888831
```

```
##    22    0.0104321512   2.399706e-03       9.365288      1.888413
##    23    0.0077177219   1.683098e-03       9.365259      1.888710
##    24    0.0058170586   1.224810e-03       9.365201      1.888894
##    25    0.0044688814   8.506761e-04       9.365534      1.888816
##    26    0.0033871369   7.869107e-04       9.365614      1.888726
##    27    0.0025512346   5.131806e-04       9.365586      1.888731
##    28    0.0019733858   3.984271e-04       9.365575      1.888714
##    29    0.0015076318   3.147959e-04       9.365600      1.888678
##    30    0.0011000531   2.363943e-04       9.365600      1.888610
##    31    0.0008591039   1.752470e-04       9.365620      1.888606
##    32    0.0006360037   1.148810e-04       9.365589      1.888602
##    33    0.0004967936   9.028587e-05       9.365583      1.888614
##    34    0.0003923203   5.966159e-05       9.365589      1.888606
##    35    0.0003328877   3.666849e-05       9.365588      1.888608
##    36    0.0003030400   2.965714e-05       9.365588      1.888613
##    37    0.0002956912   1.343417e-05       9.365586      1.888610
##    38    0.0002950952   1.231254e-05       9.365587      1.888612
##    39    0.0002950967   1.230590e-05       9.365587      1.888612
##    40    0.0002950994   1.230029e-05       9.365587      1.888612
##    41    0.0002951012   1.229710e-05       9.365587      1.888612
##    42    0.0002951036   1.229506e-05       9.365587      1.888612
##    43    0.0002951055   1.229278e-05       9.365587      1.888612
##    44    0.0002951067   1.229111e-05       9.365587      1.888612
##    45    0.0002951078   1.228951e-05       9.365587      1.888612
##    46    0.0002951087   1.228808e-05       9.365587      1.888612
##    47    0.0002951099   1.228723e-05       9.365587      1.888612
##    48    0.0002951108   1.228649e-05       9.365587      1.888612
##    49    0.0002951110   1.228588e-05       9.365587      1.888612
##    50    0.0002951113   1.228540e-05       9.365587      1.888612
##    51    0.0002951115   1.228492e-05       9.365587      1.888612
##    52    0.0002951117   1.228450e-05       9.365587      1.888612
##    53    0.0002951119   1.228406e-05       9.365587      1.888612
##    54    0.0002951122   1.228365e-05       9.365587      1.888612
##    55    0.0002951123   1.228322e-05       9.365587      1.888612
##    56    0.0002951124   1.228277e-05       9.365587      1.888612
##    57    0.0002951126   1.228237e-05       9.365587      1.888612
##    58    0.0002951128   1.228198e-05       9.365587      1.888612
##    59    0.0002951129   1.228160e-05       9.365587      1.888612
##    60    0.0002951131   1.228123e-05       9.365587      1.888612
##    61    0.0002951132   1.228090e-05       9.365587      1.888612
##    62    0.0002951133   1.228057e-05       9.365587      1.888612
##    63    0.0002951133   1.228020e-05       9.365587      1.888612
##    64    0.0002951133   1.227983e-05       9.365587      1.888612
##    65    0.0002951133   1.227962e-05       9.365587      1.888612
##    66    0.0002951134   1.227939e-05       9.365587      1.888612
##    67    0.0002951136   1.227921e-05       9.365587      1.888612
##    68    0.0002951138   1.227904e-05       9.365587      1.888612
##    69    0.0002951139   1.227887e-05       9.365587      1.888612
##    70    0.0002951141   1.227871e-05       9.365587      1.888612
##  iter train_rmse_mean train_rmse_std test_rmse_mean test_rmse_std
## Best iteration:
##  iter train_rmse_mean train_rmse_std test_rmse_mean test_rmse_std
##    20      0.01763255    0.004213917       9.364733      1.889046
```

```r
bt_highdim <- xgboost(params = params,
                      data = bt_train_cv,
                      nrounds = cv_results$best_iteration,
                      verbose = 0
                      )
```

```
## [15:44:29] WARNING: src/learner.cc:767:
## Parameters: { "verbose" } are not used.
```

```r
bt_predict <- predict(bt_highdim, newdata = bt_test_cv)
bt_mse <- mean((bt_predict - bigdata_cv_test$y)^2)
```

```
## Warning in bt_predict - bigdata_cv_test$y: longer object length is not a
## multiple of shorter object length
```

```r
bt_mse
```

```
## [1] 105.1872
```

```r
# Choose the best model (lasso)
highdim_best <- glmnet(x = x_train, y = y_train, alpha = 1, lambda = best_lambda)

# Save best predictions
highdim_test_y <- data.frame(y = predict(highdim_best, newx = x_test))
names(highdim_test_y) <- "y"

# Check for correct number of predictions
nrow(highdim_test_y) == nrow(highdim_test)
```

```
## [1] TRUE
```

```r
# Check that they have the same format
cbind(head(highdim_test_y$y), head(highdim_train$y))
```

```
##              [,1]  [,2]
## [1,]  -5.474945 -4.72
## [2,]  -7.820743  0.22
## [3,]   2.525989 22.31
## [4,] -23.197325 17.78
## [5,]  -2.649795 16.51
## [6,]  -7.116262  8.78
```

```r
# -----------------------
# Classify challenge

# Load data
classify_train <- read.csv(paste0(file_path, "ST310_2023_classify_train.csv"))
# Transform y into a factor, since it is a binary variable
classify_train$y <- as.factor(classify_train$y)
classify_test <- read.csv(paste0(file_path, "ST310_2023_classify_test.csv"))
```

```r
y_train <- classify_train[, 1]
x_train <- as.matrix(classify_train[, -1])
x_test <- as.matrix(classify_test)
#Split into train and test data
n <- nrow(classify_train)
train <- sample(1:n, floor(0.7*n), replace = F)
test <- setdiff(1:n, train)
classify_cv_train <- classify_train[train, ]
classify_cv_test <- classify_train[test, ]
#Check for class balance
table(classify_train$y)
```

```
##
##   0   1
## 349 135
```

```r
# Logistic regression model
# "binomial" corresponds to logistic regression
glm_model <- glm(y ~ ., data = classify_cv_train, family = "binomial")
# Get predicted probabilities
glm_probs <- predict(glm_model, newdata = classify_cv_test[, -1], type = "response")
# Assign a classification cutoff (trial and error)
cutoff <- mean(classify_cv_train$y == 1) + 0.2
glm_predict <- as.factor(as.numeric(glm_probs > cutoff)) # classify data based on cutoff
# Calculate the proportion of correctly classified points
accuracy <- mean(glm_predict == classify_cv_test$y)
accuracy
```

```
## [1] 0.7123288
```

```r
# Logistic regression model with upsampling of the minority class
# Create balanced data
balanced_data <- upSample(x = classify_cv_train[, -1], y = classify_cv_train$y)
glm_subsampled <- glm(Class ~ ., data = balanced_data, family = "binomial")
glm_probs <- predict(glm_subsampled, newdata = classify_cv_test[, -1], type = "response")
cutoff <- mean(balanced_data$Class == 1) + 0.16
glm_predict <- as.factor(as.numeric(glm_probs > cutoff))
accuracy <- mean(glm_predict == classify_cv_test$y)
accuracy
```

```
## [1] 0.7465753
```

```r
#Random forest model (same procedure as for bigdata and highdim)
train_ctrl <- trainControl(method = "cv", number = 10)
rf_classify <- train(y ~ .,
                     data = classify_cv_train,
                     method = "rf",
                     trControl = train_ctrl)
rf_predict <- predict(rf_classify, newdata = classify_cv_test)
accuracy <- mean(rf_predict == classify_cv_test$y)
accuracy
```

```
## [1] 0.7328767
```

```r
#Boosted trees model
bt_train_cv <- xgb.DMatrix(data = as.matrix(classify_cv_train[, -1]),
                           label = as.numeric(classify_cv_train$y) - 1)
bt_test_cv <- xgb.DMatrix(data = as.matrix(classify_cv_test[, -1]),
                          label = classify_cv_test$y)
params <- list(
  objective = "binary:logistic",  # objective function corresponding to logistic regression
  eta = 0.3,
  max_depth = 6,
  verbose = 0
)


cv_results <- xgb.cv(
  params = params,
  data = bt_train_cv,
  nfold = 10,
  nrounds = 500,
  early_stopping_rounds = 50,
  verbose = 0
)
cv_results
```

```
## ##### xgb.cv 10-folds
##  iter train_logloss_mean train_logloss_std test_logloss_mean test_logloss_std
##     1         0.54910921       0.009260691         0.6370234       0.02746381
##     2         0.46174658       0.011330458         0.6094067       0.04622935
##     3         0.39492498       0.015501017         0.6016265       0.05470416
##     4         0.34252429       0.014500621         0.5886672       0.06001376
##     5         0.30287431       0.014076686         0.5868274       0.07013597
##     6         0.26820283       0.011103958         0.5923080       0.08057164
##     7         0.24110112       0.010156827         0.6018349       0.08641414
##     8         0.21777895       0.008701048         0.6054490       0.09113397
##     9         0.19907284       0.010198362         0.6163575       0.10507794
##    10         0.18093352       0.008363810         0.6327260       0.11112829
##    11         0.16683016       0.006990920         0.6383408       0.11443358
##    12         0.15463834       0.007913120         0.6385123       0.11996585
##    13         0.14439514       0.007723676         0.6484937       0.12267420
##    14         0.13501307       0.006631419         0.6543327       0.11976747
##    15         0.12632982       0.006891837         0.6572206       0.13149868
##    16         0.11851733       0.005974769         0.6599265       0.13168663
##    17         0.11199449       0.005958417         0.6688225       0.13543469
##    18         0.10555905       0.005723747         0.6724251       0.13588432
##    19         0.09995021       0.005877525         0.6774701       0.14127824
##    20         0.09431978       0.005291273         0.6832465       0.14838452
##    21         0.08989848       0.004998752         0.6852205       0.14704908
##    22         0.08562985       0.004957856         0.6926462       0.15062493
##    23         0.08142105       0.004450885         0.6952490       0.15098209
##    24         0.07754940       0.004453592         0.7035517       0.15054398
##    25         0.07406945       0.003968848         0.7032503       0.14991583
##    26         0.07094436       0.003468147         0.7148772       0.14762371
##    27         0.06813262       0.003526041         0.7202830       0.15097388
```

```
## 28 0.06553058 0.003317228 0.7252297 0.15456696
## 29 0.06291579 0.003203378 0.7272629 0.15983435
## 30 0.06074512 0.003030668 0.7298687 0.16383041
## 31 0.05877015 0.003098879 0.7369683 0.16924137
## 32 0.05672686 0.002969363 0.7398284 0.16753627
## 33 0.05499958 0.002903525 0.7449500 0.16667916
## 34 0.05326133 0.002678705 0.7495507 0.16426469
## 35 0.05157205 0.002536892 0.7547461 0.16331175
## 36 0.05006245 0.002538184 0.7600286 0.16672124
## 37 0.04872530 0.002397510 0.7643771 0.16613071
## 38 0.04758869 0.002332412 0.7670784 0.16790123
## 39 0.04647654 0.002279825 0.7697217 0.17007700
## 40 0.04531367 0.002195794 0.7765621 0.17177639
## 41 0.04420918 0.002224275 0.7796307 0.17236477
## 42 0.04314856 0.002088319 0.7826536 0.17271222
## 43 0.04215877 0.002027445 0.7864162 0.17793397
## 44 0.04124559 0.002017809 0.7879089 0.17947092
## 45 0.04035219 0.001986088 0.7904173 0.18284180
## 46 0.03946440 0.001947801 0.7911886 0.18302701
## 47 0.03878657 0.001815967 0.7920315 0.18304483
## 48 0.03797748 0.001717664 0.7930629 0.18168138
## 49 0.03728549 0.001634470 0.7978494 0.18206512
## 50 0.03662264 0.001554071 0.7987870 0.18179683
## 51 0.03598994 0.001499356 0.7986999 0.18262798
## 52 0.03540129 0.001466127 0.8013223 0.18210576
## 53 0.03476397 0.001375411 0.8030084 0.18283357
## 54 0.03417659 0.001331831 0.8065253 0.18216844
## 55 0.03364712 0.001318842 0.8095281 0.18210651
## iter train_logloss_mean train_logloss_std test_logloss_mean test_logloss_std
## Best iteration:
## iter train_logloss_mean train_logloss_std test_logloss_mean test_logloss_std
## 5 0.3028743 0.01407669 0.5868274 0.07013597
```

```
bt_classify <- xgboost(params = params,
                       data = bt_train_cv,
                       nrounds = cv_results$best_iteration,
                       verbose = 0
                       )
```

```
## [16:06:38] WARNING: src/learner.cc:767:
## Parameters: { "verbose" } are not used.
```

```
bt_probs <- predict(bt_classify, newdata = bt_test_cv) # get predicted probabilities
cutoff <- mean(classify_cv_train$y == 1) + 0.2
bt_predict <- as.factor(as.numeric(bt_probs > cutoff)) # classify data based on cutoff
accuracy <- mean(bt_predict == classify_cv_test$y)
accuracy
```

```
## [1] 0.6986301
```

```
# Choose the best model (logistic regression with upsampling)
# Create balanced data
```

```r
balanced_data <- upSample(x = classify_train[, -1], y = classify_train$y)
classify_best <- glm(Class ~ ., data = balanced_data, family = "binomial") # train model

# Save best predictions
probs_test_y <- predict(classify_best, newdata = classify_test, type = "response") #
cutoff <- mean(balanced_data$Class == 1) + 0.16
classify_test_y <- data.frame(
  y = as.factor(as.numeric(probs_test_y > cutoff))
)
names(classify_test_y) <- "y"

# Check for correct number of predictions
nrow(classify_test_y) == nrow(classify_test)
```

```
## [1] TRUE
```

```r
# Check that predictions are classifications with same format
sort(unique(classify_test_y$y)) == sort(unique(classify_train$y))
```

```
## [1] TRUE TRUE
```