ML 4375 Homework 7: Ensemble Learning

Supratik Pochampally

Abstract

The purpose of this notebook is to try to improve performance for the Census Income dataset from the UCI Machine Learning Repository by using Random Forest, Boosting, AdaBoost, and XGBoost and comparing the speed of the algorithms.

Let's start by reading in the dataset and cleaning it, as well as split into training and testing data sets the same way we did in Project 2:

```
# Read in the .csv file of the data set
df <- read.csv("CensusIncome.csv", header = TRUE)
df <- df[-c(3, 4, 8, 11, 12, 14)]
names(df)
```

```
## [1] "age"            "workclass"      "education.num"  "marital.status"
## [5] "occupation"     "race"           "sex"            "hours.per.week"
## [9] "income"
```

```
df$workclass <- as.factor(df$workclass)
df$marital.status <- as.factor(df$marital.status)
df$occupation <- as.factor(df$occupation)
df$race <- as.factor(df$race)
df$sex <- as.factor(df$sex)
df$income <- as.factor(df$income)
# Set seed to ensure the same split of training and testing sets
set.seed(1234)
# Split the data
i <- sample(1:nrow(df), nrow(df) * 0.75, replace = FALSE)
train <- df[i, ]
test <- df[-i, ]
```

a. Random Forest

Let's start by running the Random Forest algorithm:

```
# Import the randomForest library
library(randomForest)
```

```
## Warning: package 'randomForest' was built under R version 4.0.5
```

```
## randomForest 4.6-14
```

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

```
# Set the seed to 1234 to ensure the same output every run
set.seed(1234)
# Run the randomForest algorithm
start1 <- Sys.time()
rf <- randomForest(income~., data= train, importance = TRUE)
end1 <- Sys.time()
# Print the random forest
rf
```

```
##
## Call:
##  randomForest(formula = income ~ ., data = train, importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 15.88%
## Confusion matrix:
##         <=50K  >50K class.error
##  <=50K  17099  1476  0.07946164
##  >50K    2403  3442  0.41112062
```

Now let's calculate our accuracy and mcc metrics:

```
# Import the mltools library
library(mltools)
```

```
## Warning: package 'mltools' was built under R version 4.0.5
```

```
# Predict from the random forest algorithm
pred1 <- predict(rf, newdata = test, type = "response")
# Calculate accuracy, mcc, and runtime metrics
acc1 <- mean(pred1 == test$income)
mcc1 <- mcc(factor(pred1), test$income)
runtime1 <- end1 - start1
# Print metrics
print(paste("accuracy:", acc1))
```

```
## [1] "accuracy: 0.839700282520575"
```

```
print(paste("mcc:", mcc1))
```

```
## [1] "mcc: 0.541942930940001"
```

```
print(paste("run time:", runtime1))
```

```
## [1] "run time: 21.2903079986572"
```

b. Boosting

Let's start by running the Boosting algorithm:

```
# Import the adabag library
library(adabag)
```

```
## Warning: package 'adabag' was built under R version 4.0.5
```

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

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

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

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

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

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

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

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

```
## Warning: package 'doParallel' was built under R version 4.0.5
```

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

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

```
# Set the seed to 1234 to ensure the same output every run
set.seed(1234)
# Run the boosting algorithm
start2 <- Sys.time()
adab <- boosting(income~., data = train, boos = TRUE, mfinal = 20, coeflearn = 'Breiman')
end2 <- Sys.time()
# Print the summary of the boosting algorithm
summary(adab)
```

```
##            Length Class   Mode
## formula         3  formula call
## trees          20  -none-  list
## weights        20  -none-  numeric
## votes       48840  -none-  numeric
## prob        48840  -none-  numeric
## class       24420  -none-  character
## importance      8  -none-  numeric
## terms           3  terms   call
## call            6  -none-  call
```

Now let's calculate our accuracy and mcc metrics:

```
# Predict from the boosting algorithm
pred2 <- predict(adab, newdata = test, type = "response")
# Calculate accuracy, mcc, and runtime metrics
acc2 <- mean(pred2$class == test$income)
mcc2 <- mcc(factor(pred2$class), test$income)
runtime2 <- end2 - start2
# Print metrics
print(paste("accuracy:", acc2))
```

```
## [1] "accuracy: 0.831347500307088"
```

```
print(paste("mcc:", mcc2))
```

```
## [1] "mcc: 0.515156520565998"
```

```
print(paste("run time:", runtime2))
```

```
## [1] "run time: 16.0274579524994"
```

c. AdaBoost

Let's start by running the AdaBoost algorithm:

```
# Import the fastAdaboost library
library(fastAdaboost)
```

```
## Warning: package 'fastAdaboost' was built under R version 4.0.5
```

```
# Set the seed to 1234 to ensure the same output every run
set.seed(1234)
# Run the Adaboosting algorithm
start3 <- Sys.time()
fadab <- adaboost(income~., train, 10)
end3 <- Sys.time()
# Print the summary of the Adaboosting algorithm
summary(fadab)
```

```
##                   Length Class   Mode
## formula            3     formula call
## trees             10     -none-  list
## weights           10     -none-  numeric
## classnames         2     -none-  character
## dependent_variable 1     -none-  character
## call               4     -none-  call
```

Now let's calculate our accuracy and mcc metrics:

```r
# Predict from the boosting algorithm
pred3 <- predict(fadab, newdata = test, type = "response")
# Calculate accuracy, mcc, and runtime metrics
acc3 <- mean(pred3$class == test$income)
mcc3 <- mcc(pred3$class, test$income)
runtime3 <- end3 - start3
# Print metrics
print(paste("accuracy:", acc3))
```

```
## [1] "accuracy: 0.808991524382754"
```

```r
print(paste("mcc:", mcc3))
```

```
## [1] "mcc: 0.468052557694284"
```

```r
print(paste("run time:", runtime3))
```

```
## [1] "run time: 5.34798288345337"
```

d. XGBoost

Let's start by running the XGBoost algorithm:

```r
# Import the xgboost library
library(xgboost)
```

```
## Warning: package 'xgboost' was built under R version 4.0.5
```

```r
# Assign the training and testing set labels and matrices
train_label <- ifelse(as.character(train$income) == " >50K", 1, 0)
test_label <- ifelse(as.character(test$income) == " >50K", 1, 0)
train_matrix <- data.matrix(train)
test_matrix <- data.matrix(test)
# Run the XGBoost algorithm
start4 <- Sys.time()
xgmodel <- xgboost(data = train_matrix, label = train_label, nrounds = 100, objective = 'binary:logisti
```

```
## [17:47:27] WARNING: amalgamation/../src/learner.cc:1061: Starting in XGBoost 1.3.0, the default evalu
## [1]  train-logloss:0.437550
## [2]  train-logloss:0.296367
## [3]  train-logloss:0.207408
## [4]  train-logloss:0.147866
## [5]  train-logloss:0.106680
## [6]  train-logloss:0.077564
## [7]  train-logloss:0.056703
## [8]  train-logloss:0.041614
## [9]  train-logloss:0.030627
## [10] train-logloss:0.022589
## [11] train-logloss:0.016688
```

```
## [12] train-logloss:0.012347
## [13] train-logloss:0.009146
## [14] train-logloss:0.006783
## [15] train-logloss:0.005037
## [16] train-logloss:0.003746
## [17] train-logloss:0.002792
## [18] train-logloss:0.002085
## [19] train-logloss:0.001561
## [20] train-logloss:0.001174
## [21] train-logloss:0.000886
## [22] train-logloss:0.000673
## [23] train-logloss:0.000515
## [24] train-logloss:0.000397
## [25] train-logloss:0.000309
## [26] train-logloss:0.000243
## [27] train-logloss:0.000194
## [28] train-logloss:0.000157
## [29] train-logloss:0.000128
## [30] train-logloss:0.000106
## [31] train-logloss:0.000104
## [32] train-logloss:0.000101
## [33] train-logloss:0.000100
## [34] train-logloss:0.000098
## [35] train-logloss:0.000097
## [36] train-logloss:0.000083
## [37] train-logloss:0.000082
## [38] train-logloss:0.000082
## [39] train-logloss:0.000082
## [40] train-logloss:0.000082
## [41] train-logloss:0.000082
## [42] train-logloss:0.000082
## [43] train-logloss:0.000081
## [44] train-logloss:0.000081
## [45] train-logloss:0.000081
## [46] train-logloss:0.000081
## [47] train-logloss:0.000081
## [48] train-logloss:0.000081
## [49] train-logloss:0.000081
## [50] train-logloss:0.000081
## [51] train-logloss:0.000081
## [52] train-logloss:0.000081
## [53] train-logloss:0.000081
## [54] train-logloss:0.000081
## [55] train-logloss:0.000081
## [56] train-logloss:0.000081
## [57] train-logloss:0.000081
## [58] train-logloss:0.000081
## [59] train-logloss:0.000081
## [60] train-logloss:0.000081
## [61] train-logloss:0.000081
## [62] train-logloss:0.000081
## [63] train-logloss:0.000081
## [64] train-logloss:0.000081
## [65] train-logloss:0.000081
```

```
## [66] train-logloss:0.000081
## [67] train-logloss:0.000081
## [68] train-logloss:0.000081
## [69] train-logloss:0.000081
## [70] train-logloss:0.000081
## [71] train-logloss:0.000081
## [72] train-logloss:0.000081
## [73] train-logloss:0.000081
## [74] train-logloss:0.000081
## [75] train-logloss:0.000081
## [76] train-logloss:0.000081
## [77] train-logloss:0.000081
## [78] train-logloss:0.000081
## [79] train-logloss:0.000081
## [80] train-logloss:0.000081
## [81] train-logloss:0.000081
## [82] train-logloss:0.000081
## [83] train-logloss:0.000081
## [84] train-logloss:0.000081
## [85] train-logloss:0.000081
## [86] train-logloss:0.000081
## [87] train-logloss:0.000081
## [88] train-logloss:0.000081
## [89] train-logloss:0.000081
## [90] train-logloss:0.000081
## [91] train-logloss:0.000081
## [92] train-logloss:0.000081
## [93] train-logloss:0.000081
## [94] train-logloss:0.000081
## [95] train-logloss:0.000081
## [96] train-logloss:0.000081
## [97] train-logloss:0.000081
## [98] train-logloss:0.000081
## [99] train-logloss:0.000081
## [100]    train-logloss:0.000081
```

```
end4 <- Sys.time()
# Plot the model
```

Now let's calculate our accuracy and mcc metrics:

```
# Predict from the boosting algorithm
probs <- predict(xgmodel, test_matrix)
pred4 <- ifelse(probs > 0.5, 1, 0)
# Calculate accuracy, mcc, and runtime metrics
acc4 <- mean(pred4 == test_label)
mcc4 <- mcc(pred4 , test_label)
runtime4 <- end4 - start4
# Print metrics
print(paste("accuracy:", acc4))
```

```
## [1] "accuracy: 1"
```

```
print(paste("mcc:", mcc4))
```

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

```
print(paste("run time:", runtime4))
```

```
## [1] "run time: 0.178171873092651"
```

Summary of results

The following ar the accuracy, mcc, and run time values for the four algorithms we ran:

- Random Forest-
  - accuracy: 0.839700282520575
  - mcc: 0.541942930940001
  - run time: 21.9593360424042
- Boosting-
  - accuracy: 0.831347500307088
  - mcc: 0.515156520565998
  - run time: 16.9347229003906
- AdaBoost-
  - accuracy: 0.808991524382754
  - mcc: 0.468052557694284
  - run time: 5.46624493598938
- XGBoost-
  - accuracy: 1
  - mcc: 1
  - run time: 0.177515029907227

Looking at the metrics, we can see that the best performing algorithms based on accuracy and mcc values in order are XGBoost, Random Forest, Boosting, and AdaBoost. The fastest algorithms in order are XGBoost, AdaBoost, Boosting, and Random Forest. XGBoost's accuracy of 1 could be because XGBoost was able split hundreds of trees and aggregate them to weigh the best predictors in a way that allowed it to perfectly predict the data. However, this could have occurred because of poor feature selection. In our feature selection process, we tried to pick the features that would help best predict the class of our data, and removed predictors that we deemed were unnecessary or would not be able to help predict from the data. During this process, it was possible that one or two predictors could obviously predict the data, and we did not notice it like XGBoost managed too. This is all just speculation, however, and we may be able to see the better results if we tried using cross-validation or printed and observed the XGBoost tree structure. XGBoost naturally took the least amount of time, as it is able to run faster using multithreading. Random forest took the longest time, but showed the fruits of it's labor through it's performance, with the second best accuracy and mcc values. Similarly, AdaBoost was faster than Boosting thanks to it's C++ implementation being ~100 times faster than regular Boosting, but made the sacrifice of a worse performance than Boosting.