Attrition XGBoost

Adrian Jenkins

9/21/2021

The dataset is composed of 1470 observations and 36 attributes. For this model we will be using 7 out of the original 35 attributes. The variables to include are the following: "Age", "Attrition", "Business Travel", "Education", "Job Level", "Monthly Income" and "OverTime".

We will create a XGBoosting model to predict "Attrition", which refers to whether the employee left or not the company. Then, we will try to optimize the model to increase the recall. We want to detect the people who left the company.

Dataset contains no missing values. Here, the structure of our data:

```
## tibble [1,470 x 7] (S3: tbl_df/tbl/data.frame)
##  $ Age           : num [1:1470] 41 49 37 33 27 32 59 30 38 36 ...
##  $ Attrition     : num [1:1470] 1 0 1 0 0 0 0 0 0 0 ...
##  $ BusinessTravel: chr [1:1470] "Travel_Rarely" "Travel_Frequently" "Travel_Rarely" "Travel_Frequentl
##  $ Education     : num [1:1470] 2 1 2 4 1 2 3 1 3 3 ...
##  $ JobLevel      : num [1:1470] 2 2 1 1 1 1 1 1 3 2 ...
##  $ MonthlyIncome : num [1:1470] 5993 5130 2090 2909 3468 ...
##  $ OverTime      : chr [1:1470] "Yes" "No" "Yes" "Yes" ...
```

We proceed to divide the data set into training and test with a proportion of 75% in the former and 25% in the latter. The independent variables selected were: "Age", "Attrition", "Business Travel", "Education", "Job Level", "Monthly Income", "Over Time".

```
## <Analysis/Assess/Total>
## <1101/369/1470>
```

This algorithm only allows independent variables of numerical type, therefore the "One Hot Encoding" process must be performed and a matrix must be provided as required.

```
trainY <-  attrition_train$Attrition == "1"
trainX <- model.matrix(Attrition ~.-1, data = attrition_train)

testY <- attrition_test$Attrition == "1"
testX <- model.matrix(Attrition ~. -1, data = attrition_test)

# DMatrix
Xmatrix_training <-  xgb.DMatrix(data = trainX, label = trainY)
Xmatrix_testing <-xgb.DMatrix(data = testX, label = testY)
```

We then proceed to create the model, make the predicions and see its results:

```
Xgboosting <- xgboost(data = Xmatrix_training,
                      objective = "multi:softmax",
                      num_class = 2,
                      nrounds = 51,
                      eval_metric="merror"
)
```

```
##        Predicted
## Actual    0   1
##   FALSE 296  13
##   TRUE   46  14
```

```
## [1] "Accuracy: 0.840108"
```

```
## [1] "Recall: 0.233333"
```

```
## [1] "Specificity: 0.957929"
```

```
## [1] "Precision: 0.518519"
```

```
## [1] "F1 Score: 0.321839"
```

According to **accuracy**, the model correctly predicted 84% of the cases.

According to **precision**, of all the people that the model classified as "Yes", only 51.8% of the cases actually left the company.

According to **sensitivity**, of all the people who left the company, the model was correct in 23.3% of the cases.

According to the **specificity**, out of a total of 309 people who did not leave the company, the model correctly predicted 95.7% of the observations.

The **"F1 - Score"** has a value of 32.1%.

In order to optimize the parameters for this model, the grid search strategy was used and 10,000 models with different parameter values were created Once the previous iteration was completed, the best performing model was selected according to the multiple error metric. The hyper parameters are as follows:

- "eta": controls the learning rate. It takes a value within the interval [0, 1].
- "max_depth": controls the depth of the tree. The greater the depth, the greater the complexity of the model, the greater the chance of overfitting. It takes a value within the interval [0, infinite].
- "gamma": controls the regularization. The higher the value, the greater the regularization. Regularization penalizes large coefficients that do not improve model performance.It takes a value within the interval [0, infinite].
- "subsample: controls the number of observations supplied to a tree. It takes a value within the interval [0, 1].
- "colsample_bytree": controls the number of features (variables) supplied to a tree. It takes a value within the interval [0, 1].
- "min_child_weight": value that determines whether the tree splitting stops.It takes a value within the interval [0, infinite].

- "alpha": controls the L1 regularization (equivalent to Lasso regression) in the weights.
- "lambda": controls the L2 regularization (equivalent to Ridge regression) in the weights. It is used to avoid over-fitting.

```
cl <- makePSOCKcluster(9) # I have found this number is optimal for my pc.
registerDoParallel(cl) # You could also run it without doParallel library, just will take longer

# Take start time to measure time of random search algorithm
start.time <- proc.time()

# Create empty lists
lowest_merror_list = list()
parameters_list = list()

# Create 10,000 rows with random hyperparameters

set.seed(20)
for (iter in 1:10000){
  param <- list(booster = "gbtree",
                objective = "multi:softmax",
                max_depth = sample(3:10, 1),
                eta = runif(1, .01, .3),
                subsample = runif(1, .7, 1),
                colsample_bytree = runif(1, .6, 1),
                min_child_weight = sample(0:10, 1),
                gamma = floor(runif(1, 0, 100)),
                alpha = runif(1, 0, 100),
                lambda = runif(1, 0, 100)

  )
  parameters <- as.data.frame(param)
  parameters_list[[iter]] <- parameters
}

# Create object that contains all randomly created hyperparameters
parameters_df = do.call(rbind, parameters_list)

# Use randomly created parameters to create 10,000 XGBoost-models

for (row in 1:nrow(parameters_df)){
  set.seed(20)
  mdcv <- xgb.train(data=Xmatrix_training,
                    booster = "gbtree",
                    objective = "multi:softmax",
                    max_depth = parameters_df$max_depth[row],
                    eta = parameters_df$eta[row],
                    subsample = parameters_df$subsample[row],
                    colsample_bytree = parameters_df$colsample_bytree[row],
                    min_child_weight = parameters_df$min_child_weight[row],
                    gamma = parameters_df$gamma[row],
                    alpha = parameters_df$alpha[row],
                    lambda = parameters_df$lambda[row],
                    nrounds= 351,
```

```
                    eval_metric = "merror",
                    early_stopping_rounds= 30,
                    print_every_n = 100,
                    num_class = 2,
                    watchlist = list(train= Xmatrix_training, val= Xmatrix_testing)
  )
  lowest_merror <-  as.data.frame(1 - min(mdcv$evaluation_log$val_merror))
  lowest_merror_list[[row]] = lowest_merror
}

# Create object that contains all accuracy's
lowest_merror_df = do.call(rbind, lowest_merror_list)

# Bind columns of accuracy values and random hyperparameter values
randomsearch = cbind(lowest_merror_df, parameters_df)

# Quickly display highest accuracy
maxacc <- max(randomsearch$`1 - min(mdcv$evaluation_log$val_merror)`)
print(randomsearch %>% filter(`1 - min(mdcv$evaluation_log$val_merror)` == maxacc))

# Stop time and calculate difference
end.time <- proc.time()
time.taken <- end.time - start.time
print(time.taken)

write_csv(randomsearch, "dataset/randomsearch.csv")

stopCluster(cl)

## Best Parameters
# 0.859079
# max_depth = 10
# eta = 0.2276653
# subsample = 0.8562708
# colsample_bytree = 0.9899229
# min_child_weight = 5
# gamma = 0
# alpha = 0.04643467
# lambda = 0.8723213
```

With these new values for the hyper parameters, the model is trained again:

```
# Using tuned params

Xgboosting2 <- xgboost(data = Xmatrix_training,
                       objective = "multi:softmax",
                       num_class = 2,
                       nrounds = 251,
                       eval_metric = "merror",
                       max_depth = 10,
                       eta = 0.2276,
                       subsample = 0.8562,
                       colsample_bytree = 0.9899,
                       min_child_weight = 5,
```

```
                        gamma = 0,
                        alpha = 0.04643,
                        lambda = 0.8723,
                        set.seed(1)
)
```

```
##        Predicted
## Actual    0   1
##   FALSE 290  19
##   TRUE   44  16
```

```
## [1] "Accuracy: 0.829268"
```

```
## [1] "Recall: 0.266667"
```

```
## [1] "Specificity: 0.938511"
```

```
## [1] "Precision: 0.457143"
```

```
## [1] "F1 Score: 0.336842"
```

According to **accuracy**, the model predicted correctly in 82.9% of the cases.

According to **precision**, of all the people that the model classified as "Yes", only 45.7% of the cases actually left the company.

According to **sensitivity**, of all the people who left the company, the model was correct in 26.6% of the cases.

According to **specificity**, out of a total of 309 people who did not leave the company, the model correctly predicted 93.8% of the observations.

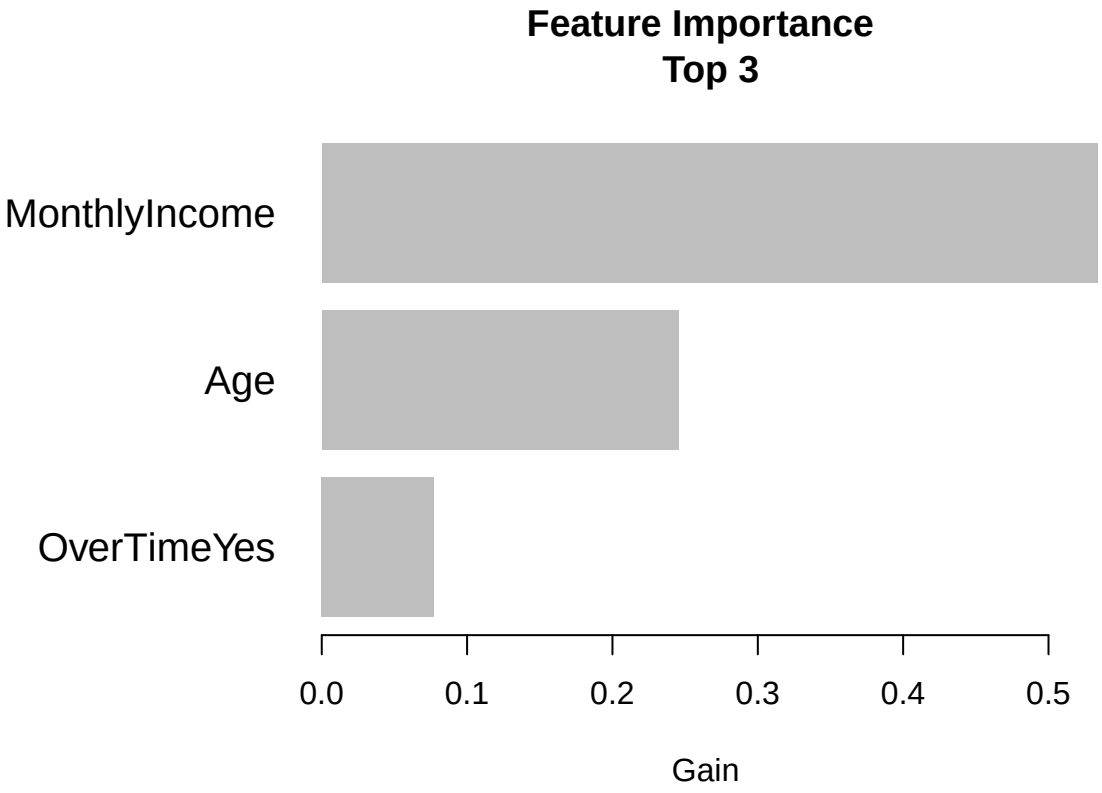
The **"F1 - Score"** has a value of 33.6%.

The importance of variables according to the model constructed is shown below:

```
# Feature Importance
importance_matrix <- xgb.importance(model = Xgboosting2)
print(importance_matrix)
```

```
##                            Feature       Gain      Cover   Frequency
## 1:                   MonthlyIncome 0.53361088 0.57341063 0.566852368
## 2:                             Age 0.24539184 0.25166404 0.257544104
## 3:                     OverTimeYes 0.07729226 0.04128326 0.034122563
## 4:                       Education 0.06296769 0.06255418 0.072771588
## 5:                        JobLevel 0.03896158 0.02497929 0.027274838
## 6: BusinessTravelTravel_Frequently 0.01974089 0.01935493 0.016713092
## 7:     BusinessTravelTravel_Rarely 0.01616202 0.01559604 0.020194986
## 8:        BusinessTravelNon-Travel 0.00587284 0.01115764 0.004526462
```

```
xgb.plot.importance(importance_matrix = importance_matrix[1:3,],
                    main = "Feature Importance \n Top 3",
                    xlab = "Gain"
)
```



The three variables with the greatest impact on the model are: "Monthly Income", "Age" and "Over Time".

Comparing Both Models

Metric	First Model	Second Model
Accuracy	84.%	82.9%
Precision	51.8%	45.7%
Recall	23.3%	26.6%
Specificity	96.7%	93.8%
"*F1 - Score*"	32.1%	33.6%

Figure 1: Comparing XGBoosting Models

Although the improvement in the recall is not that much, it did improve.

Thanks you for your attention.