Homework 2 - Group

2024-10-19

In this R markdown, you will find us going through the dataset and finding the best model to predict adopters for XYZ company.

Throughout this document, we have tried to segregate blocks of code with the function/method we are applying to the data. Moreover, you will find the training data often being balanced with the 'ovun.sample' function - for a combination of over and under sampling. More on our thinking for this before relevant blocks.

Reading the XYZ dataset and converting columns to factors

```
suppressMessages({
  suppressWarnings({
library(readxl)
library(e1071)
# install.packages("ROSE")
library(ROSE)
library(dplyr)
library(pROC)
#install.packages("rpart.plot")
library(caret)
library(rpart)
library(rpart.plot)
library(class)

# XYZData.csv file renamed to Dataset.csv due to random errors in some of our systems
XYZ <- read.csv('Dataset.csv')
XYZ$adopter <- as.factor(XYZ$adopter)
XYZ$good_country <- as.factor(XYZ$good_country)
XYZ$male <- as.factor(XYZ$male)
})})
```

Taking out the training and testing folds

```
# Setting seed for consistent data for knitting
set.seed(123)
# Choosing a 80-20 split
train_rows = createDataPartition(y = XYZ$adopter, p = 0.80, list = FALSE)
xyz_train = XYZ[train_rows,]
xyz_test = XYZ[-train_rows,]
# Balancing the main training dataset for later use
xyz_train_bal <- ovun.sample(adopter ~ ., data = xyz_train, method = "both", p = 0.5, nrow(data))$dat
```

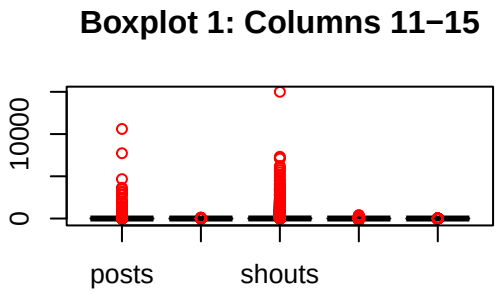
Using the ovun.sample function with the both method, we increase the representation of adopters while balancing the overall dataset, allowing our model to learn effectively from both adopters and non-adopters. Balancing the training data was deemed crucial because, without enough examples of adopters, we felt that the model might struggle to identify patterns unique to this group and may become biased towards predicting non-adopters.

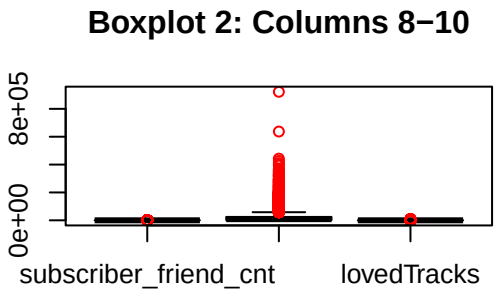
We will use this training split henceforth and use the testing fold for end model evaluation.

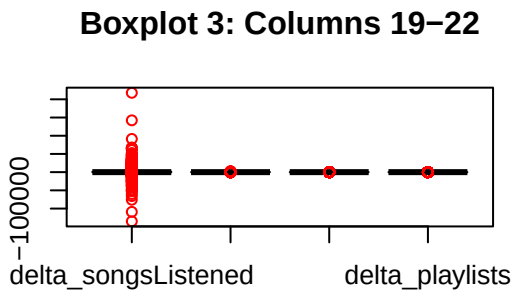
Boxplots for data spread and outlier detection

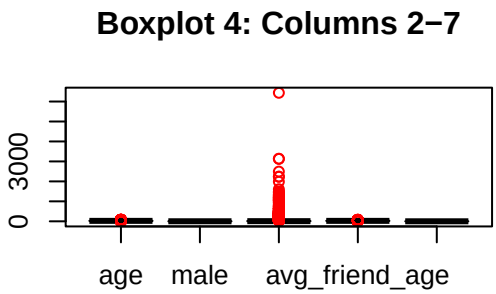
First, visualizing spread of variables and gauging the extent of outlier presence. To visualize the many variables, we created a 4-boxplot subplots below:

```
# Looking at boxplots for the fields, to check outliers and spread of data
# Set up a 2x2 layout for the 4 boxplots
par(mfrow = c(2, 2))  # 2 rows, 2 columns

# First boxplot (columns 11 to 15)
boxplot(XYZ[11:15], col = "grey", outcol = "red", las = 0.5, main = "Boxplot 1: Columns 11-15")

# Second boxplot (columns 8 to 10)
boxplot(XYZ[8:10], col = "grey", outcol = "red", las = 0.5, main = "Boxplot 2: Columns 8-10")

# Third boxplot (columns 19 to 22)
boxplot(XYZ[19:22], col = "grey", outcol = "red", las = 0.5, main = "Boxplot 3: Columns 19-22")

# Fourth boxplot (columns 2 to 7)
boxplot(XYZ[2:6], col = "grey", outcol = "red", las = 0.5, main = "Boxplot 4: Columns 2-7")
```










```
# Reset the plotting layout back to default (optional, if you want to create additional plots afterward
par(mfrow = c(1, 1))
```

Looking at the boxplots, most fields have outliers and many specific fields you see in the above boxplots have a lot of data falling outside 1.5 IQR levels.

KNN seems unlikely due to this, since using KNN in the correct way would require narrow feature selection and removal of many outliers - reducing information was something we were wary of.

To rule out KNN, we decided to test KNN with feature selection, to see gauge performance:

Feature Selection for KNN

```
library(FSelectorRcpp)
IG = information_gain(adopter ~ ., data = xyz_train)
# Selecting top 10 features, we reiterated this process by changing k value and ended up with 10 as the
topK = cut_attrs(IG, k = 10)

xyz_topK_train = xyz_train %>% select(topK, adopter)
```

```
## Warning: Using an external vector in selections was deprecated in tidyselect 1.1.0.
## i Please use ‘all_of()‘ or ‘any_of()‘ instead.
##   # Was:
##   data %>% select(topK)
```

```
##
##   # Now:
##   data %>% select(all_of(topK))
##
## See <https://tidyselect.r-lib.org/reference/faq-external-vector.html>.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```
xyz_topK_test = xyz_test %>% select(topK, adopter)
```

Normalizing the data to check KNN

```
normalize = function(x){
  return ((x - min(x))/(max(x) - min(x)))
}

# Oversampling the post feature selection training dataset
xyz_train_topk_bal <- ovun.sample(adopter ~ ., data = xyz_topK_train, method = "both", p = 0.5, nrow(da

xyz_train_norm <- xyz_train_topk_bal %>% mutate_at(c(1:10), normalize)

xyz_test_norm = xyz_topK_test %>% mutate_at(c(1:10), normalize)
```

Metrics Philosophy and Selection

Choosing Recall (primary metric), F Measure, and AUC as our metrics to gauge model performance:

We chose recall because we believe capturing as many adopters from the data would highly relevant for XYZ. Capturing adopters by targetting a bit more of non-adopters is an acceptable cost, if the return from these adopters is high - at the end we would give XYZ the capability to set their own threshold, while we would suggest what we believe is the best.

AUC of ROC and F1 measure was chosen to gauge the performance of the model as whole, first to see how well our model is able to detect true positive, as well as to keep Precision in check with F-Measure.

Our aim was to obtain as high a recall as possible while having high value for F-Measure and AUC of ROC.

Running KNN and checking our chosen metrics

We decided to run KNN on multiple values of k and find the best k-value that gives the highest AUC, and the corresponding F-Measure and Recall:

```
suppressMessages({
  suppressWarnings({

k_values <- seq(1, 20, by = 2)  # Test odd values of k from 1 to 20
auc_results <- numeric(length(k_values))
precision_k<- numeric(length(k_values))
recall_k<- numeric(length(k_values))
f1_score_k<- numeric(length(k_values))
```

```
for (i in 1:length(k_values)) {
  k <- k_values[i]

  knn_pred <- knn(
    train = xyz_train_norm[, -which(names(xyz_train_norm) == "adopter")],
    test = xyz_test_norm[, -which(names(xyz_test_norm) == "adopter")],
    cl = xyz_train_norm$adopter,
    k = k
  )

  knn_pred_numeric <- as.numeric(knn_pred) - 1

  roc_curve_knn <- roc(xyz_test_norm$adopter, knn_pred_numeric)
  cm_k = confusionMatrix(data=factor(knn_pred), reference = factor(xyz_test_norm[,11]), mode = "prec_re
  auc_results[i] <- auc(roc_curve_knn)
  precision_k[i] <- cm_k$byClass["Pos Pred Value"]
  recall_k[i] <- cm_k$byClass["Sensitivity"]

  # Calculate F1 Score (F-measure)
  f1_score_k[i] <- 2 * ((precision_k[i] * recall_k[i]) / (precision_k[i] + recall_k[i]))
}
})})
# Print the best K value with its corresponding AUC
best_k_auc <- which.max(auc_results)
print(paste("Best K for AUC =", k_values[best_k_auc], "with AUC =", auc_results[best_k_auc]))
```

```
## [1] "Best K for AUC = 9 with AUC = 0.581400974025974"
```

```
print(paste("Recall for best K =", recall_k[best_k_auc], "F1 =", f1_score_k[best_k_auc]))
```

```
## [1] "Recall for best K = 0.448051948051948 F1 = 0.101173020527859"
```

Here, we can see that the AUC hovers around 58-60% for k=9. The recall is 44% with a F-Measure of 0.10 - both these metrics were not compelling enough to us. In the subsequent code, it'll become apparent that other methods perform way better than KNN Classification.

At this stage, we will have to choose between Decision Tree and Naive Bayes Methods. Here, we use cross-validation on both Decision Tree and Naive Bayes methods, to observe mean AUC, Recall, and F1 Measure:

– Point to note: In decision tree, we did not use our feature-selection fold as decision-tree inherently performs feature selection with information gain.

Running cross-validation with folds on Decision Tree:

```
suppressMessages({
  suppressWarnings({
# Use the createFolds function to generate a list of row indexes that correspond to each fold
# We will use the same folds on Naive Bayes, for ideal comparison of performance
set.seed(111)
cv = createFolds(y = xyz_topK_train$adopter, k = 5)
```

```
auc_results_dt <- numeric(length(k))
precision_dt<- numeric(length(k))
recall_dt<- numeric(length(k))
f1_score_dt<- numeric(length(k))

# loop through each fold
for (val_rows in cv) {

  dt_train = xyz_train[-val_rows,]
  # over-under sampling locally so that validation fold is not affected
  dt_train_bal <- ovun.sample(adopter ~ ., data = dt_train, method = "both", p = 0.5, nrow(data))$dat
  dt_val = xyz_train[val_rows,]

  tree = rpart(adopter ~ ., data = dt_train_bal)
  pred_tree = predict(tree, dt_val, type = 'class')
  cm_dt = confusionMatrix(data=factor(pred_tree), reference = factor(dt_val[,27]), mode = "prec_recall"

  # Calculate F1 Score (F-measure), Recall and ROC-AUC
  precision_dt[val_rows] <- cm_dt$byClass["Pos Pred Value"]
  recall_dt[val_rows] <- cm_dt$byClass["Sensitivity"]
  f1_score_dt[val_rows] <- 2 * ((precision_dt[val_rows] * recall_dt[val_rows]) / (precision_dt[val_rows]
  pred_tree_num <- as.numeric(pred_tree) - 1
  roc_curve_dt <- roc(dt_val$adopter, pred_tree_num)
  auc_results_dt[val_rows] <- auc(roc_curve_dt)

  #print(paste("F1 Score:", f1_score_dt, recall_dt, precision_dt))
  #print(paste("AUC:", auc_results))
}
})})
print(paste("Mean AUC score:", mean(auc_results_dt)))
```

```
## [1] "Mean AUC score: 0.707003903330914"
```

```
print(paste("Mean Recall:", mean(recall_dt)))
```

```
## [1] "Mean Recall: 0.755663773613862"
```

```
print(paste("Mean F1:", mean(f1_score_dt)))
```

```
## [1] "Mean F1: 0.142340119093827"
```

The results from Decision Tree were promising - we are seeing high AUC of ROC and high recall, with slightly higher F-Measure than KNN.

Running cross validation with same folds on Naive Bayes:

```
suppressMessages({
  suppressWarnings({
auc_results_nb <- numeric(length(k))
```

```
precision_nb<- numeric(length(k))
recall_nb<- numeric(length(k))
f1_score_nb<- numeric(length(k))
for (val_rows in cv) {
  # Using our feature selected training fold here
  nb_train = xyz_topK_train[-val_rows,]
  # over-under sampling locally so that validation fold is not affected
  nb_train_bal <- ovun.sample(adopter ~ ., data = nb_train, method = "both", p = 0.5, nrow(data))$dat
  nb_val = xyz_topK_train[val_rows,]
  NB_model = naiveBayes(adopter ~ ., data = nb_train_bal)

  pred_nb = predict(NB_model, nb_val)
  prob_pred_nb = predict(NB_model, nb_val, type = 'raw')

  cmnb = confusionMatrix(data = pred_nb,
                 reference = nb_val$adopter,
                 mode = "prec_recall",
                 positive = "1")
  # Calculate F1 Score (F-measure), Recall and ROC-AUC
  precision_nb[val_rows] <- cmnb$byClass["Pos Pred Value"]
  recall_nb[val_rows] <- cmnb$byClass["Sensitivity"]

  pred_nb_num <- as.numeric(pred_nb) - 1
  roc_curve_dt <- roc(nb_val$adopter, pred_nb_num)
  auc_results_nb[val_rows] <- auc(roc_curve_dt)

  f1_score_nb[val_rows] <- 2 * ((precision_nb[val_rows] * recall_nb[val_rows]) / (precision_nb[val_rows]

  #print(paste("F1 Score:", f1_score_nb, recall_nb, precision_nb))
  #print(paste("AUC:", auc_results))
}
})})
print(paste("Mean AUC score:", mean(auc_results_nb)))
```

```
## [1] "Mean AUC score: 0.600090992656019"
```

```
print(paste("Mean Recall:", mean(recall_nb)))
```

```
## [1] "Mean Recall: 0.293869686548797"
```

```
print(paste("Mean F1:", mean(f1_score_nb)))
```

```
## [1] "Mean F1: 0.157995830037864"
```

Naive Bayes model performs quite poorly compared to Decision Tree. Decision Tree has higher Recall for a respectable F-Measure and also has higher AUC.

At this stage, we choose the Decision Tree as the best method to create a model for XYZ. We can improve Recall and manipulate the threshold to give results based on what budget and capabilities XYZ has.

We use the entire training dataset (balanced) to train our ultimate model, and find the performance on our initial testing fold.

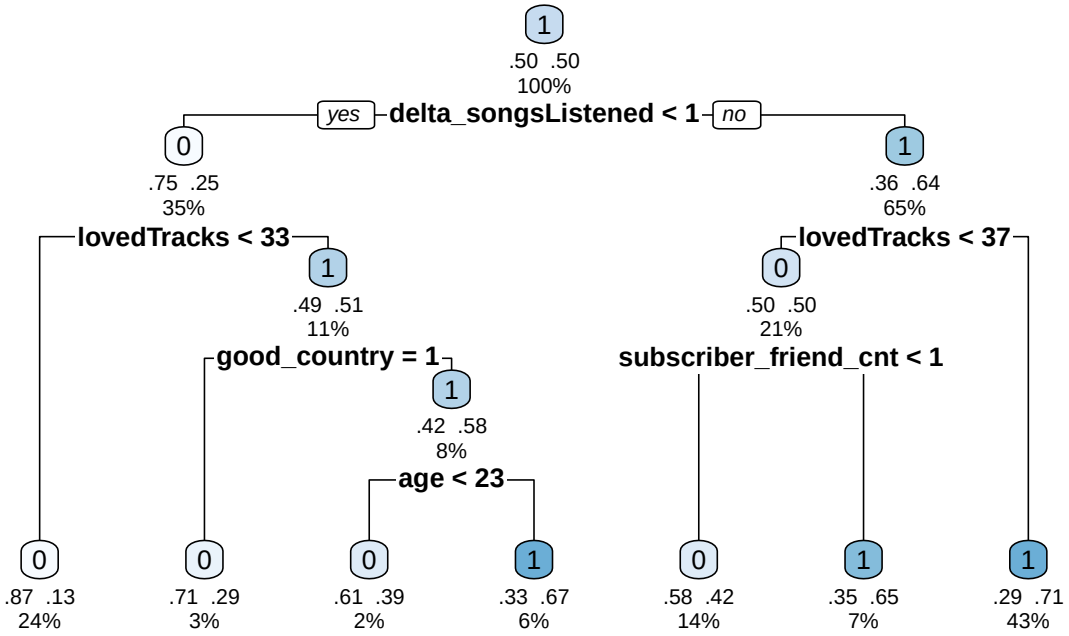
Finalizing decision tree, training on balanced data and predicting on testing data:

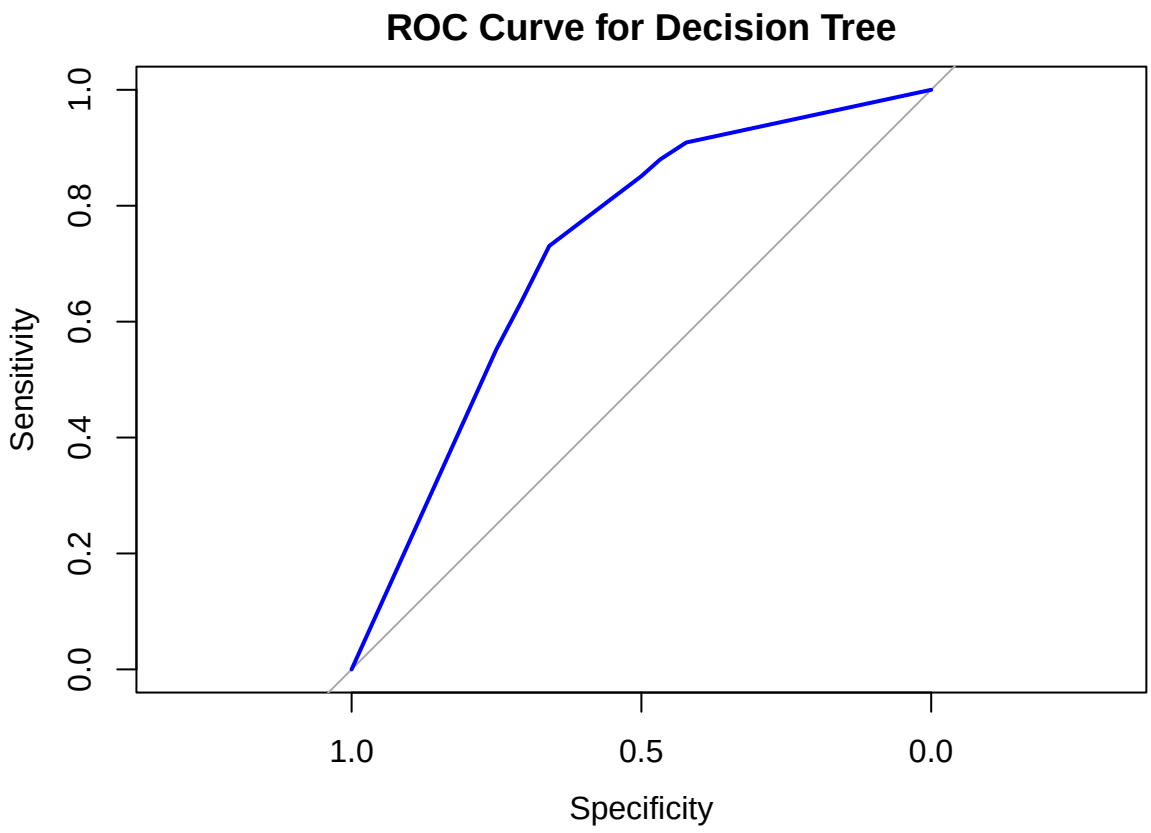
```
suppressMessages({
  suppressWarnings({
tree = rpart(adopter ~ ., data = xyz_train_bal)
pred_tree = predict(tree, xyz_test, type = 'prob')[, 2]
# Ability to manipulate this to increase/decrease recall in conjunction with precision
# Lower threshold will help XYZ capture more adopters, but will have to target more non-adopters as wel
threshold = 0.4
pred_custom_threshold = ifelse(pred_tree > threshold, 1, 0)
cm = confusionMatrix(data=factor(pred_custom_threshold), reference = factor(xyz_test[,27]), mode = "pre

# Calculate F1 Score (F-measure), Recall, AUC of ROC
precision <- cm$byClass["Pos Pred Value"]
recall <- cm$byClass["Sensitivity"]
f1_score <- 2 * ((precision * recall) / (precision + recall))

#print(paste("F1 Score:", f1_score))
#print (recall)
#print (precision)

roc_curve <- roc(xyz_test$adopter, pred_tree)
auc_value <- auc(roc_curve)

print (cm)
print(paste("AUC: ", auc_value))

rpart.plot(tree,
           type = 2,          # Draw split labels all the way to the left
           extra = 104,       # Display class probabilities and percentage of observations
           under = TRUE,      # Show labels under the nodes for readability
           faclen = 0,        # Do not abbreviate factor labels
           cex = 0.8,         # Adjust font size for clarity
           box.palette = "Blues",  # Use a color palette for visual ease
           main = "Decision Tree for Predicting Adopters")
# Plot ROC curve
plot(roc_curve, col = "blue", main = "ROC Curve for Decision Tree")
})})
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 4004   46
##          1 3996  262
##
##                Accuracy : 0.5135
##                  95% CI : (0.5027, 0.5243)
##     No Information Rate : 0.9629
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.049
```

```
##
##  Mcnemar's Test P-Value : <2e-16
##
##               Precision : 0.06153
##                  Recall : 0.85065
##                      F1 : 0.11476
##              Prevalence : 0.03707
##          Detection Rate : 0.03154
##    Detection Prevalence : 0.51252
##       Balanced Accuracy : 0.67557
##
##        'Positive' Class : 1
##
## [1] "AUC:  0.724801136363636"
```

Decision Tree for Predicting Adopters





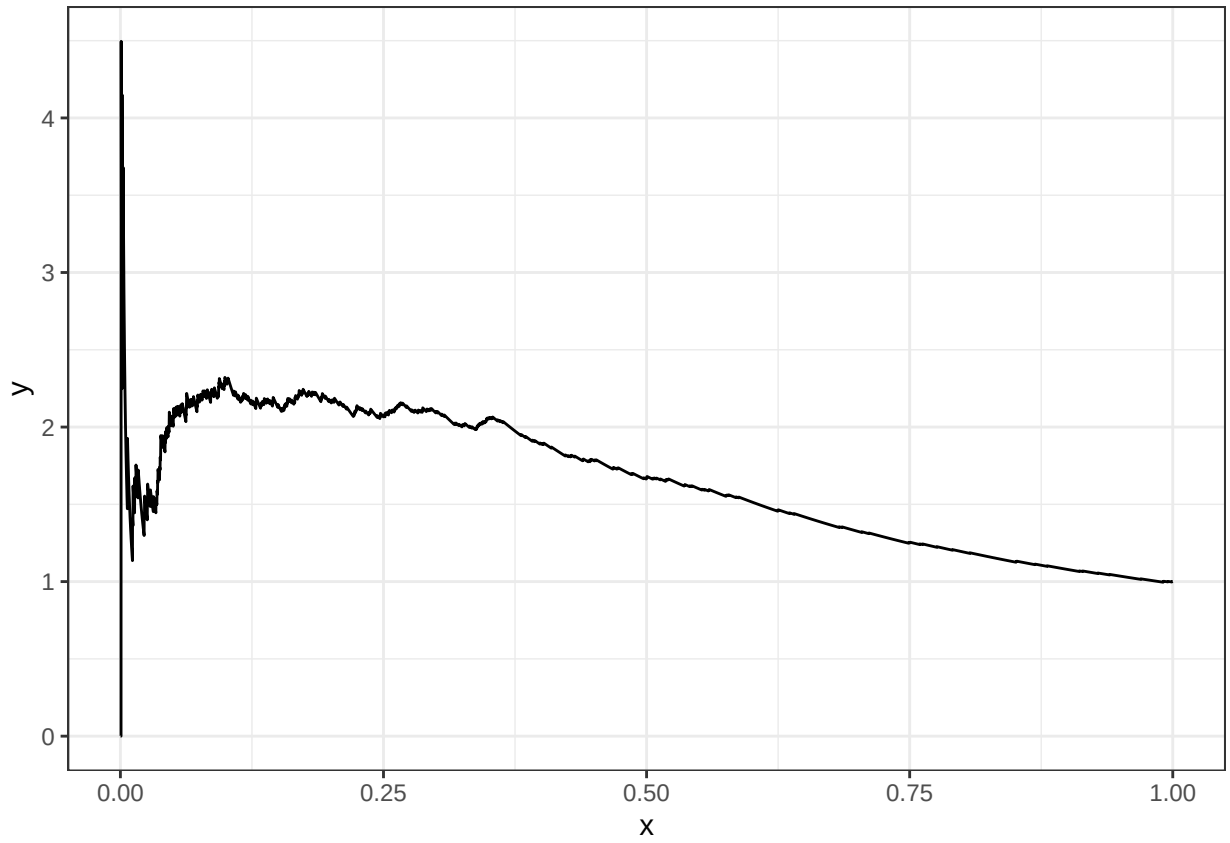
This is the final performance of our model. We suggest keeping a threshold value of 0.4 - this value gives the most efficient breakdown of adopters and non-adopters correctly identified (Look at confusion matrix above). On the testing fold, we capture more than 50% of non-adopters correctly, while capturing 85% of the adopters correctly.

If threshold is decreased Recall will increase but so will the number of non-adopters targetted disproportionately - suggested if XYZ has high budget and wants to capture all adopters regardless of cost.

If threshold is increased Recall will decrease but so will the number of adopters targetted disproportionately - suggested if XYZ has low budget and wants to capture as many adopters with lowest cost.

Plotting and observing lift curve:

```
XYZ_lift = xyz_test %>%
  mutate(prob = pred_tree) %>%
  arrange(desc(prob)) %>%
  mutate(adopter_yes = ifelse(adopter==1,1,0)) %>%

  mutate(x = row_number()/nrow(xyz_test),
         y = (cumsum(adopter_yes)/sum(adopter_yes))/x)

ggplot(data = XYZ_lift, aes(x = x, y = y)) +
  geom_line() +
  theme_bw()
```



Here we can deduce that there is a lift of for the top 25% customers. This means that the top 25% of users, as ranked by your model's predictions, are twice as likely to adopt premium service compared to a random selection of users.

Targeting this top 25% should yield a significantly higher return on investment for the marketing campaign since it effectively isolates users who are twice as likely to subscribe.

Also, with limited marketing resources, focusing on this top 25% is a smart strategy. Instead of randomly selecting potential customers, the model-based selection increases the likelihood of conversions, potentially reducing wasted resources on unlikely adopters.

Example Implementation:

Implementing this lift curve insight into the test data as an example for how XYZ can identify top 25% users:

```
xyz_test$prediction <- pred_tree
xyz_test <- xyz_test[order(-xyz_test$prediction), ]

# Calculate the number of rows to select (25% of the total rows)
top_25_percent <- round(0.25 * nrow(xyz_test))

# Select the top 25% rows
top_users <- xyz_test[1:top_25_percent, ]
```

'top_users' contains the top 25% users that are twice as likely to adopt premium service.

Addendum (additional insights we found): Realization of irrelevant features in Decision Tree

Adding to the previous note that decision tree inherently performs feature selection - and we can see the irrelevant feature below with the Importance function in R:

```
tree = rpart(adopter ~ ., data = xyz_train_bal)
importance <- varImp(tree, scale = FALSE)
print(importance)
```

```
##                                Overall
## age                          440.08354
## avg_friend_age               848.94033
## delta_friend_cnt              94.51213
## delta_lovedTracks           2104.96205
## delta_songsListened         2444.59847
## friend_cnt                  1177.00578
## friend_country_cnt           231.56538
## good_country                 126.92297
## lovedTracks                 3555.98624
## playlists                    279.55448
## posts                         84.04969
## subscriber_friend_cnt       2553.59120
## user_id                        0.00000
## male                           0.00000
## avg_friend_male                0.00000
## songsListened                  0.00000
## shouts                         0.00000
## delta_avg_friend_age           0.00000
## delta_avg_friend_male          0.00000
## delta_friend_country_cnt       0.00000
## delta_subscriber_friend_cnt    0.00000
## delta_posts                    0.00000
## delta_playlists                0.00000
## delta_shouts                   0.00000
## tenure                         0.00000
## delta_good_country             0.00000
```

We can see that all the fields with 0 importance have negligible contribution to the model.

The varImp function calculates importance based on how often a field is used in splits and the improvement it brings in node purity.

If we use just the fields/features with >0 importance, we would get the same results - demonstrated below:

```
xyz_final = xyz_train_bal %>% select(age, avg_friend_age,delta_friend_cnt, delta_lovedTracks,delta_songs

tree = rpart(adopter ~ ., data = xyz_final)
pred_tree = predict(tree, xyz_test, type = 'prob')[, 2]
threshold = 0.4
pred_custom_threshold = ifelse(pred_tree > threshold, 1, 0)
cm = confusionMatrix(data=factor(pred_custom_threshold), reference = factor(xyz_test[,27]), mode = "pre
precision <- cm$byClass["Pos Pred Value"]
recall <- cm$byClass["Sensitivity"]
```

```r
# Calculate F1 Score (F-measure)
f1_score <- 2 * ((precision * recall) / (precision + recall))

#print(paste("F1 Score:", f1_score))
#print (recall)
#print (precision)

roc_curve <- roc(xyz_test$adopter, pred_tree)
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

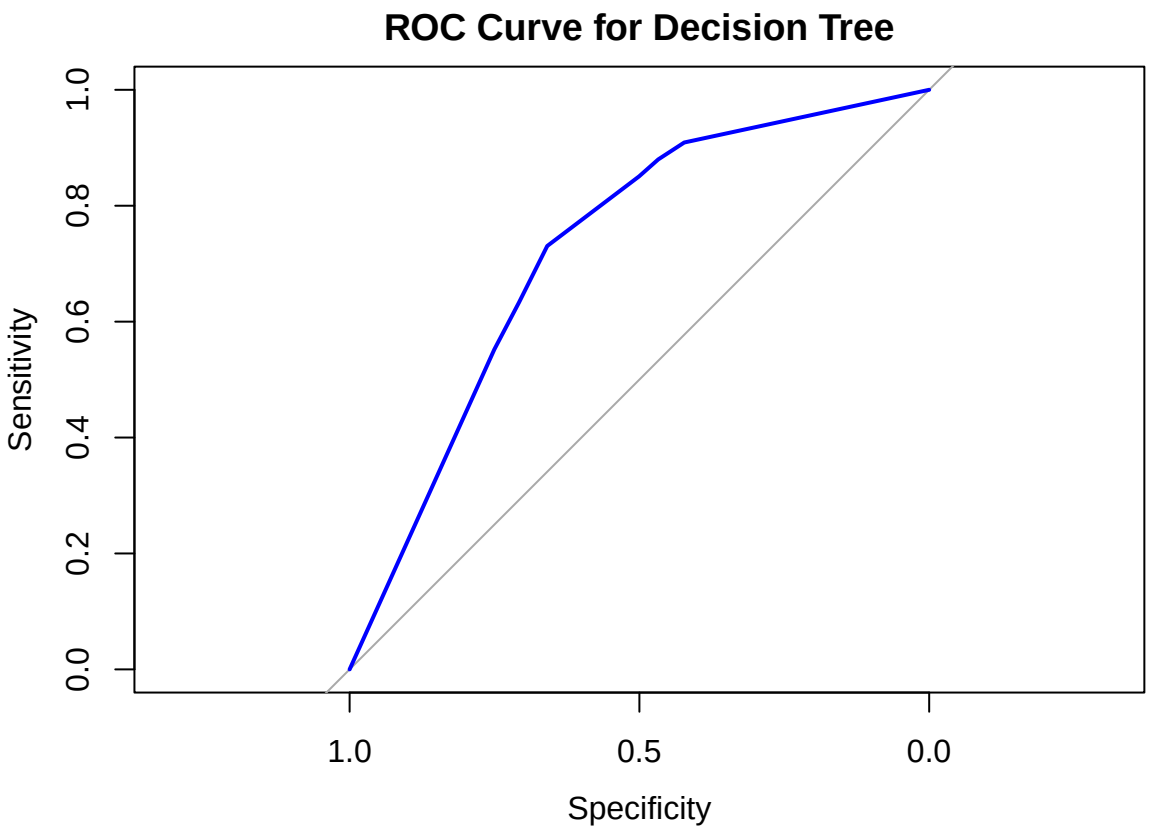
```r
auc_value <- auc(roc_curve)

print (cm)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 4004   46
##          1 3996  262
##
##                Accuracy : 0.5135
##                  95% CI : (0.5027, 0.5243)
##     No Information Rate : 0.9629
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.049
##
##  Mcnemar's Test P-Value : <2e-16
##
##               Precision : 0.06153
##                  Recall : 0.85065
##                      F1 : 0.11476
##              Prevalence : 0.03707
##          Detection Rate : 0.03154
##    Detection Prevalence : 0.51252
##       Balanced Accuracy : 0.67557
##
##        'Positive' Class : 1
##
```

```r
print(paste("AUC: ", auc_value))
```

```
## [1] "AUC:  0.724801136363636"
```

```r
# Plot ROC curve
plot(roc_curve, col = "blue", main = "ROC Curve for Decision Tree")
```



You'll find the same confusion matrix with same AUC of ROC for a lot lesser columns analyzed - reinforcing the automatic feature selection done by the decision tree.