Optimize Premium User Acquisition Cost and Conversion Rate with Predictive Modeling for Music Streaming Service

Yi Hsiang Yen (Royce Yen)

2024-10-25

Introduction

Here is a simple outline of our overall steps:

1. Data processing
2. Model selection
3. Feature selection
4. Finalize model tuning for the presentation to XYZ

As mentioned in the description and by examining this data with R and Excel, we have identified two key findings:

1. The values of interval variables vary significantly. For example, for the variable "songsListened," the maximum value is 922370, while for "playlists," it is only 98. Therefore, we will need to normalize our variables for certain models.
2. There are only 1,540 adopters out of 41,540 records, indicating that this dataset is imbalanced.

Based on these findings, we have two conclusions.

1. We will use ROC/AUC as our performance evaluation metric.

   Relying solely on accuracy, precision, or recall as performance evaluation metrics can be misleading.

   - For accuracy, when one class is significantly more prevalent than the other, high accuracy can be misleading. For example, if 95% of the instances belong to the majority class, a model could achieve 95% accuracy simply by predicting the majority class every time, without actually learning to identify the minority class. However, in this case, we are interested in learning about the characteristics that predict the minority class - adopters.
   - For precision, in cases of imbalanced datasets, a model can achieve high precision by only predicting a few positive adopters correctly, but it may still miss a significant number of actual adopters.
   - For recall, for imbalanced datasets, focusing solely on recall can result in a model that identifies many negative instances as positive, which is also a poor performance indicator when viewing it alone.
   - F1 score and ROC/AUC are better metrics in this case because it evaluates performance across all possible classification thresholds, providing insights into the model's ability to correctly classify instances from both the majority and minority classes. This makes it less sensitive to class distribution compared to metrics like accuracy.

2. We will perform oversampling on our training data to address class imbalance.

Now, let's start with part 1: Data processing.

1. Data Processing

1.1 Import all necessary packages

```
# Check and install packages
# List of packages
packages <- c("e1071", "caret", "smotefamily", "ROSE", "klaR",
              "rpart", "rpart.plot", "dplyr", "ggplot2", "pROC",
              "class", "FSelectorRcpp")

# Function to check and install packages
for (pkg in packages) {
  if (!require(pkg, character.only = TRUE)) {
    install.packages(pkg)
    library(pkg, character.only = TRUE)
  }
}
```

```
## Loading required package: e1071

## Loading required package: caret

## Loading required package: ggplot2

## Loading required package: lattice

## Loading required package: smotefamily

## Loading required package: ROSE

## Loaded ROSE 0.0-4

## Loading required package: klaR

## Loading required package: MASS

## Loading required package: rpart

## Loading required package: rpart.plot

## Loading required package: dplyr

##
## Attaching package: 'dplyr'

## The following object is masked from 'package:MASS':
##
##     select
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

## Loading required package: pROC

## Type 'citation("pROC")' for a citation.

##
## Attaching package: 'pROC'

## The following objects are masked from 'package:stats':
##
##     cov, smooth, var

## Loading required package: class

## Loading required package: FSelectorRcpp
```

```r
# Load libraries
library(e1071)
library(caret)
library(smotefamily) # For SMOTE
library(ROSE)        # Alternative for oversampling
library(klaR)        # For Naive Bayes
library(rpart)       # For Decision Tree
library(rpart.plot)
library(dplyr)
library(ggplot2)
library(pROC)
library(class)
library(FSelectorRcpp)
```

1.2 Load and examine data

```r
# Load data
data = read.csv("/Users/yenyihsiang/Desktop/UMN/Fall Semester/Intro to BA in R/Group9_Homework2_Code_fi

# Check if there are missing values
missing_values_per_column <- colSums(is.na(data))
print(missing_values_per_column)
```

```
##                    user_id                        age
##                          0                          0
##                       male                friend_cnt
```

```
##                          0                          0
##              avg_friend_age             avg_friend_male
##                          0                          0
##          friend_country_cnt       subscriber_friend_cnt
##                          0                          0
##               songsListened                lovedTracks
##                          0                          0
##                       posts                  playlists
##                          0                          0
##                      shouts            delta_friend_cnt
##                          0                          0
##        delta_avg_friend_age       delta_avg_friend_male
##                          0                          0
##    delta_friend_country_cnt delta_subscriber_friend_cnt
##                          0                          0
##         delta_songsListened           delta_lovedTracks
##                          0                          0
##                 delta_posts             delta_playlists
##                          0                          0
##                delta_shouts                     tenure
##                          0                          0
##                good_country          delta_good_country
##                          0                          0
##                     adopter
##                          0
```

```
# Convert categorical variables into factors
data$male <- as.factor(data$male)
data$good_country <- as.factor(data$good_country)
data$delta_good_country <- as.factor(data$delta_good_country)
data$adopter <- as.factor(data$adopter)

# Ensure adopter is a factor in the original data
data$user_id <- NULL # Remove user_id since it is not a predictive feature
```

According to the results, we can see there are no missing values.

1.3 Data partition

```
# Use createDataPartition so that it preserves distribution of the
# target variable in train/test partitions
set.seed(1) #For reproducibility
train_rows = createDataPartition(y = data$adopter, p = 0.70, list = FALSE)
data_train = data[train_rows,]
data_test = data[-train_rows,]
```

1.4 Perform oversampling with OVUN to increase the proportion of the minority class to be 33% percent of the entire dataset

We believe that in many real-world scenarios, achieving equal class sizes is neither feasible nor reflective of the actual distribution of data. Therefore, we decided on setting the minority class to 33%, because it is

significant enough to enhance the model's ability to learn from the minority class without overwhelming the majority class. This helps in avoiding the potential pitfalls of overfitting that can occur when the minority class is oversampled to an equal ratio. In terms of the oversampling function, we tested with ROSE, SMOTE, and OVUN, and we eventually decided to go with OVUN because it gives us the highest AUC scores by roughly around 10%.

```
# Note: ovun.sample balances the data by either oversampling the minority class,
# undersampling the majority class, or a combination. It Uses random sampling to
# duplicate minority class samples or remove majority class samples until
# balance is achieved.
oversampled_data <- ovun.sample(adopter ~., data = data_train,
                    method = "over", p=0.33, seed = 1)
oversampled_data_train = oversampled_data$data

# Check the class distribution after oversampling
cat("Class distribution after oversampling:\n")
```

```
## Class distribution after oversampling:
```

```
print(table(oversampled_data_train$adopter))
```

```
##
##     0     1
## 28000 13824
```

By checking the results, our data partitioning/factoring variables/and oversampling worked. That concludes our first step - data processing. Now we'll move on to the next step.

2. Model Selection

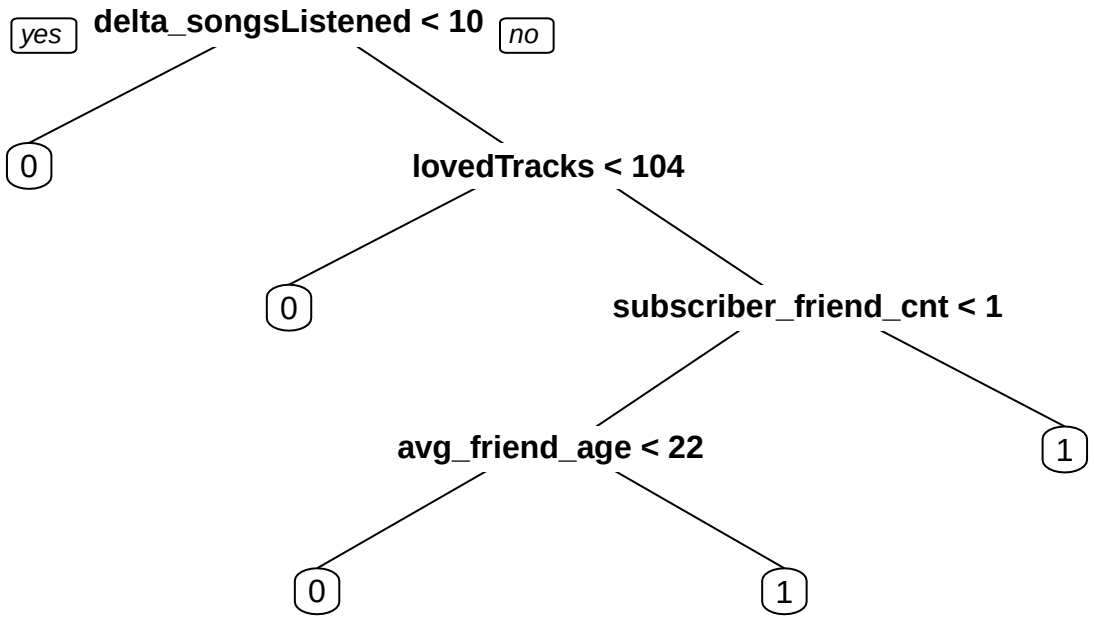
Since there are a lot of features and we have no clue which feature might have a larger impact on the model's performance, we will start by using all of them.

We start with decision tree because decision trees inherently perform feature selection during the model-building process. They evaluate all features and determine the most informative ones for making splits, which means they naturally ignore less relevant features. Furthermore, decision trees are generally not sensitive to the scale of the features. They create splits based on thresholds and do not rely on distances or magnitudes of the values, meaning we don't have to normalize our dataset for it.

2-1. Build and evaluate Decision Tree Model

```
# Fit the initial decision tree
  tree <- rpart(adopter ~ ., data = oversampled_data_train,
                method = "class", parms = list(split = "information"))

# Use model to predict class label
pred_tree = predict(tree, data_test, type = "class")
prp(tree, varlen = 0)
```




```
# check the confusion matrix to evaluate the performance of decision tree model
confusionMatrix(data = as.factor(pred_tree),
                reference = as.factor(data_test$adopter),
                mode = "prec_recall",
                positive = '1')
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     0     1
##          0 10725   301
##          1  1275   161
##
##                Accuracy : 0.8735
##                  95% CI : (0.8676, 0.8793)
##     No Information Rate : 0.9629
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.1203
##
##  Mcnemar's Test P-Value : <2e-16
##
##               Precision : 0.11212
##                  Recall : 0.34848
##                      F1 : 0.16965
##              Prevalence : 0.03707
```

```
##          Detection Rate : 0.01292
##    Detection Prevalence : 0.11523
##       Balanced Accuracy : 0.62112
##
##        'Positive' Class : 1
##
```

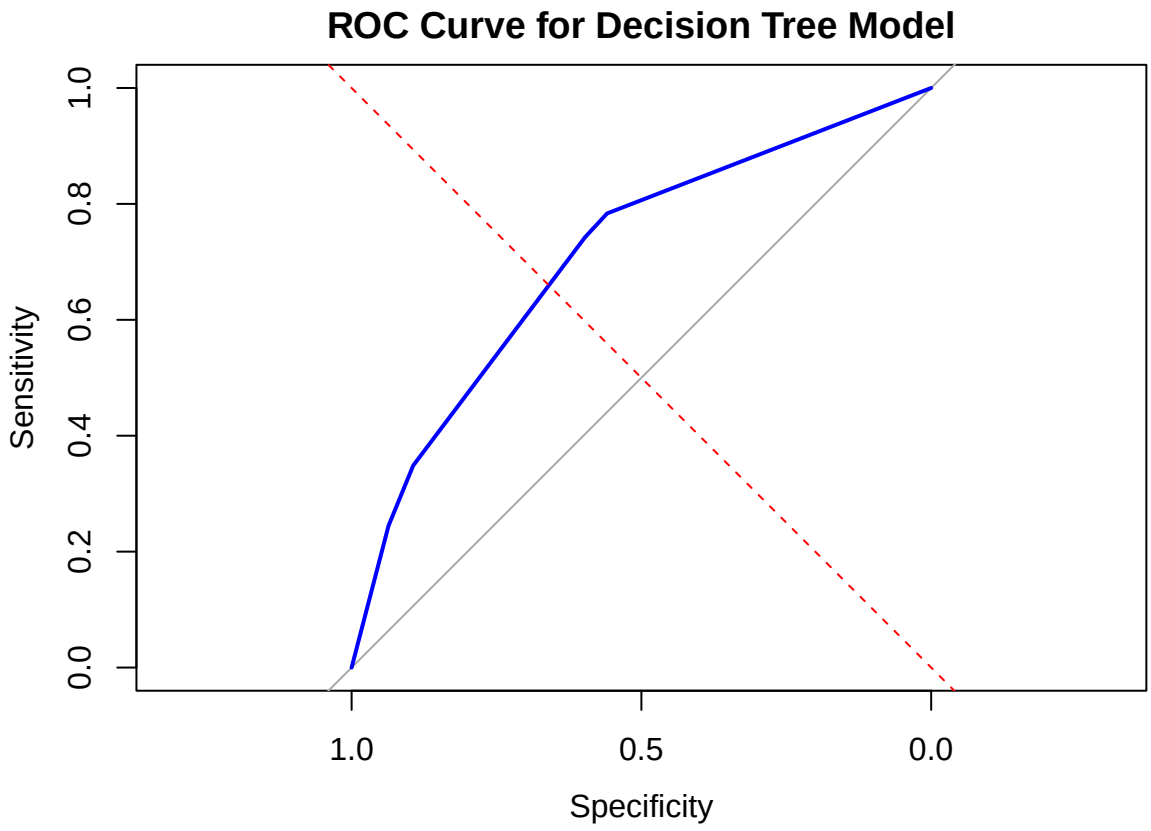
```r
# Get the predicted probabilities for the positive class
predicted_probs <- predict(tree, data_test, type = "prob")[, 2]
# Probability for the positive class (adopter == 1)

# Calculate ROC curve
roc_curve <- roc(data_test$adopter, predicted_probs)
```

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

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

```r
# Plot the ROC curve
plot(roc_curve, main = "ROC Curve for Decision Tree Model",
     col = "blue", lwd = 2)
abline(a = 0, b = 1, lty = 2, col = "red")
```



```r
# Add a diagonal line for reference

# Calculate AUC
auc_value <- auc(roc_curve)

# Print the AUC value
cat("AUC Value:", auc_value, "\n")
```

```
## AUC Value: 0.7098488
```

As we can see, with F1 score being 0.16965 and AUC being 0.7098488, this decision tree model shows potential for identifying adopters (the minority class) with reasonable recall, but it struggles with precision, leading to a high number of false positives.

Next, we considered evaluating the performance of k-Nearest Neighbors (kNN) and Naive Bayes, but ultimately decided to exclude kNN from our final presentation for the following reasons:

1. Time Consumption in Parameter Tuning: Finding the optimal number of neighbors (k) requires significant computational effort. We would need to experiment with a wide range of k values, which can be time-consuming and resource-intensive. After spending a lot of time experimenting on this, we decided to exclude this model from the final presentation.
2. Inherent Complexity of kNN: kNN is inherently a time-consuming algorithm, especially as the dataset grows in size. The need to compute distances between all points makes it less efficient compared to other models.
3. Challenges with Feature Selection: If we were to explore feature selection, we would need to assess various combinations of features alongside multiple k values. This would exponentially increase the complexity of the analysis and the time required for model training and evaluation.

Instead, we'll see how naive bayes model performs.

2-2 Build and evaluate Naive Bayes Model

```r
# Identify numerical columns
numerical_cols <- sapply(oversampled_data_train, is.numeric)

# Create the normalization function using Min-Max scaling
normalize_data <- function(data, numerical_cols) {
  for (col in numerical_cols) {
    min_val <- min(data[[col]], na.rm = TRUE)
    max_val <- max(data[[col]], na.rm = TRUE)
    data[[col]] <- (data[[col]] - min_val) / (max_val - min_val)
  }
  return(data)
}

# Normalize the oversampled training and test sets
normalized_train <-
  normalize_data(oversampled_data_train, names(numerical_cols)[numerical_cols])
normalized_train$adopter <-
  oversampled_data_train$adopter  # Add target variable back
```

```r
normalized_test <-
  normalize_data(data_test, names(numerical_cols)[numerical_cols])
normalized_test$adopter <-
  data_test$adopter  # Add target variable back

# Build & evaluate Naive Bayes Model with confusion matrix
NB_model <- naiveBayes(adopter ~ ., data = normalized_train)
pred_nb = predict(NB_model, normalized_test)  # Ensure to use normalized_test
prob_pred_nb = predict(NB_model, normalized_test, type = "raw")

# Confusion Matrix
confusionMatrix(data = pred_nb,
                reference = normalized_test$adopter,
                mode = "prec_recall",
                positive = '1')
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     0     1
##          0    12     0
##          1 11988   462
##
##                Accuracy : 0.038
##                  95% CI : (0.0347, 0.0415)
##     No Information Rate : 0.9629
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 1e-04
##
##  Mcnemar's Test P-Value : <2e-16
##
##               Precision : 0.03711
##                  Recall : 1.00000
##                      F1 : 0.07156
##              Prevalence : 0.03707
##          Detection Rate : 0.03707
##    Detection Prevalence : 0.99904
##       Balanced Accuracy : 0.50050
##
##        'Positive' Class : 1
##
```

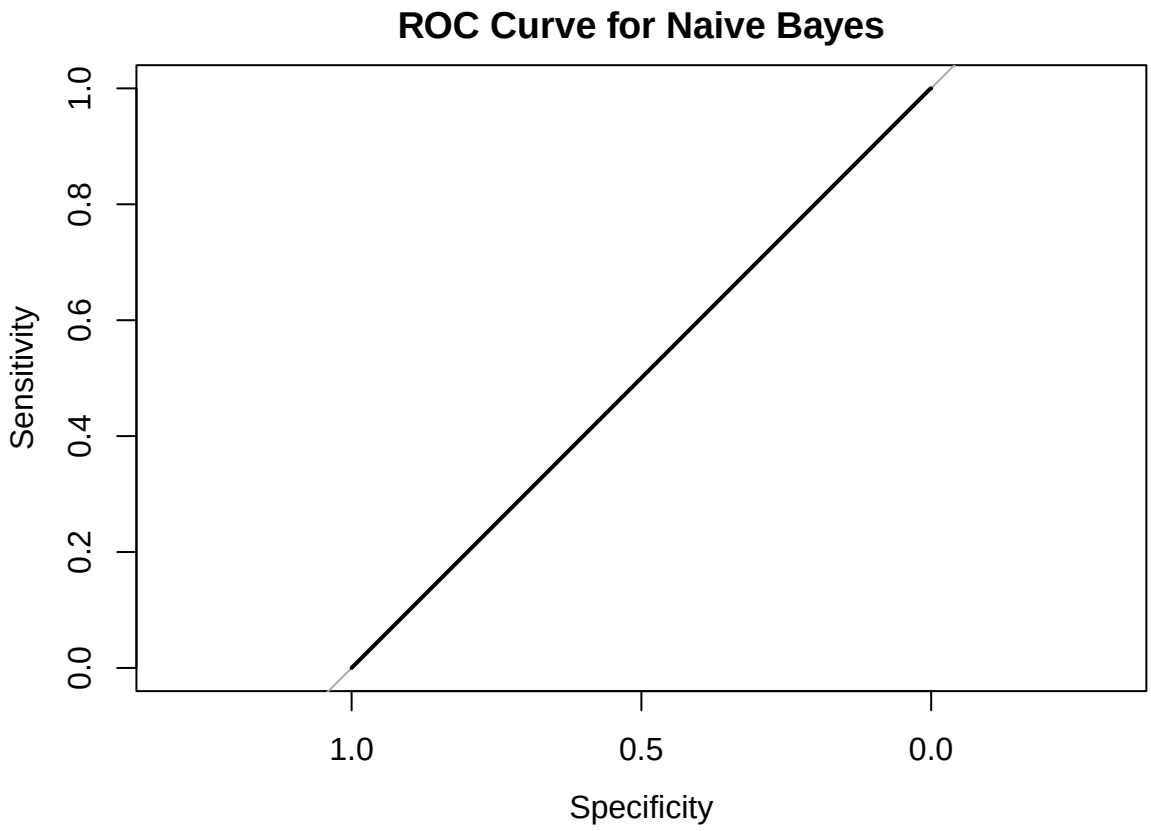
```r
# Create a data frame for ROC analysis
roc_data <-
  data.frame(acc_yes = normalized_test$adopter,
                       prob = prob_pred_nb[, "1"])
                       # Assuming "1" is the positive class

# Calculate the ROC curve
roc_nb <- roc(response = roc_data$acc_yes, predictor = roc_data$prob)
```

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

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

```
# Plot the ROC curve
plot(roc_nb, main = "ROC Curve for Naive Bayes")
```




```
# Calculate and print the AUC
auc_value <- auc(roc_nb)
print(paste("AUC:", auc_value))
```

```
## [1] "AUC: 0.5005"
```

According to our result, naive bayes have an AUC of only 0.5005.

As a result, among these three models, we will explore feature selection only with the decision tree model.

3 Feature Selection

There are three common feature selection methods: Forward Selection, Backward Elimination, and Filter Approach.

To mitigate computation costs, reduce noise, minimize overfitting risk, and enhance interpretability, we will use the Filter Approach to identify the most important features. Specifically, we will sequentially add features based on their importance ranking from the Filter Approach, assessing the AUC performance for each configuration (e.g., using only the top feature, then the top two, and so on, up to the top 25 features).

3-1. Filter Approach with Information Gain

```
# Calculate Information Gain for each feature
IG <- information_gain(adopter ~ ., data = data_train)
# Rank the features based on Information Gain
top_features <- cut_attrs(IG, k = 25)
print("Top features by Information Gain:")
```

```
## [1] "Top features by Information Gain:"
```

```
print(top_features)
```

```
##  [1] "lovedTracks"                "delta_songsListened"
##  [3] "delta_lovedTracks"          "subscriber_friend_cnt"
##  [5] "songsListened"              "friend_cnt"
##  [7] "friend_country_cnt"         "delta_friend_cnt"
##  [9] "delta_avg_friend_male"      "delta_shouts"
## [11] "avg_friend_male"            "delta_subscriber_friend_cnt"
## [13] "posts"                      "shouts"
## [15] "delta_friend_country_cnt"   "playlists"
## [17] "delta_avg_friend_age"       "avg_friend_age"
## [19] "age"                        "male"
## [21] "delta_posts"                "good_country"
## [23] "delta_good_country"         "delta_playlists"
## [25] "tenure"
```

As we can see, among the top ten most important features, there are mainly two categories:

1. User behavior -> "lovedTracks", "delta_songsListened", "delta_lovedTracks", "songsListened", "delta_shouts"
2. Friend Information -> "subscriber_friend_cnt", "friend_cnt", "friend_country_cnt", "delta_friend_cnt", "delta_avg_friend_male"

We are curious whether applying the top ten features by categories will have an impact on our model, so we tried it out.

3-2-1. Decision Tree Model with features only from category "User behavior"

```
# Select only "User behavior" features
user_behavior_features <- c("lovedTracks", "delta_songsListened",
                            "delta_lovedTracks", "songsListened",
                            "delta_shouts")

# Train Decision Tree Model with User Behavior features
tree_user_behavior <-
  rpart(adopter ~ .,
        data = oversampled_data_train[, c(user_behavior_features, "adopter")],
                            method = "class",
                            parms = list(split = "information"))
```

```r
# Predict and evaluate
pred_tree_user_behavior <-
  predict(tree_user_behavior, data_test[, user_behavior_features],
          type = "class")
confusionMatrix(data = pred_tree_user_behavior,
                reference = data_test$adopter,
                mode = "prec_recall",
                positive = '1')
```

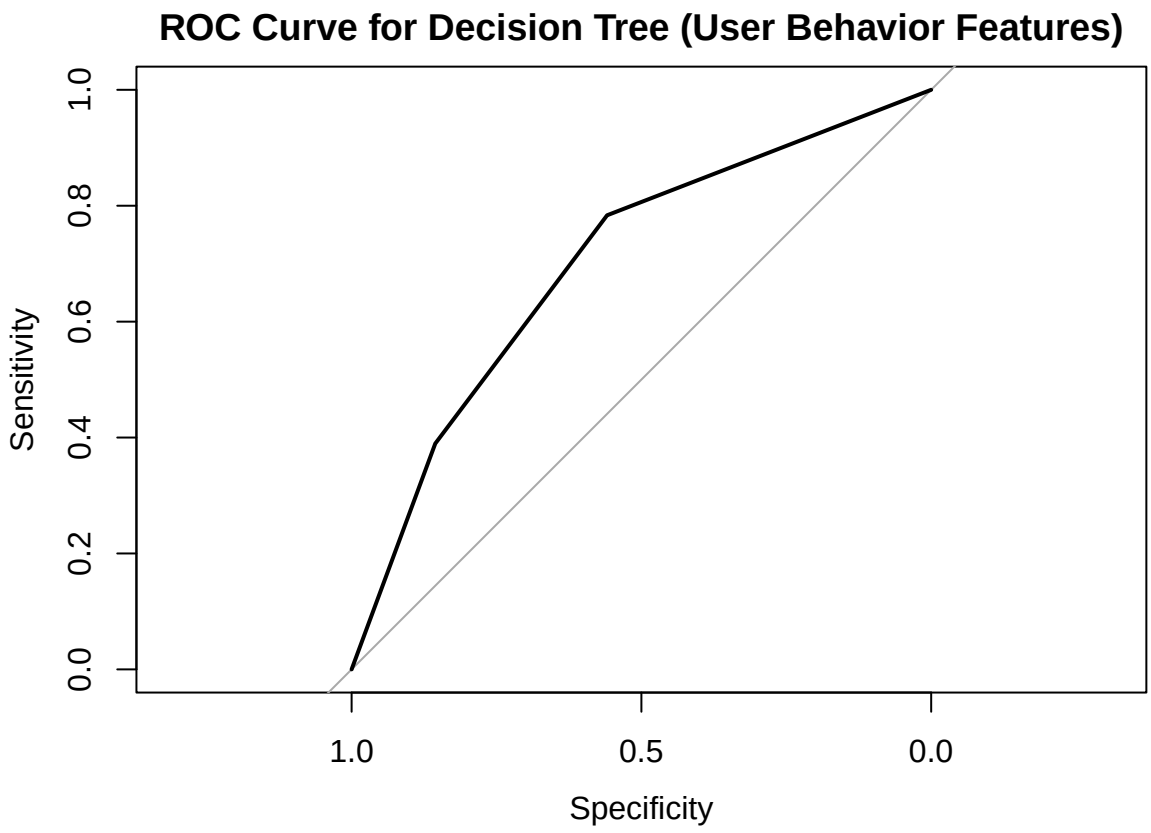
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     0     1
##          0 10268   282
##          1  1732   180
##
##                Accuracy : 0.8384
##                  95% CI : (0.8318, 0.8448)
##     No Information Rate : 0.9629
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.0978
##
##  Mcnemar's Test P-Value : <2e-16
##
##               Precision : 0.09414
##                  Recall : 0.38961
##                      F1 : 0.15164
##              Prevalence : 0.03707
##          Detection Rate : 0.01444
##    Detection Prevalence : 0.15343
##       Balanced Accuracy : 0.62264
##
##        'Positive' Class : 1
##
```

```r
# Calculate AUC
predicted_probs_tree_ub <-
  predict(tree_user_behavior,
          data_test[, user_behavior_features], type = "prob")[, 2]
roc_tree_user_behavior <- roc(data_test$adopter, predicted_probs_tree_ub)
```

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

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

```r
plot(roc_tree_user_behavior,
     main = "ROC Curve for Decision Tree (User Behavior Features)")
```




```
auc_tree_user_behavior <- auc(roc_tree_user_behavior)
print(paste("AUC (User Behavior - Decision Tree):", auc_tree_user_behavior))
```

```
## [1] "AUC (User Behavior - Decision Tree): 0.700688672438672"
```

3-2-2. Decision Tree Model with features only from category "Friend Information"

```
# Select only "Friend Information" features
friend_info_features <- c("subscriber_friend_cnt", "friend_cnt",
                          "friend_country_cnt", "delta_friend_cnt",
                          "delta_avg_friend_male")

# Train Decision Tree Model with Friend Information features
tree_friend_info <-
  rpart(adopter ~ .,
        data = oversampled_data_train[, c(friend_info_features, "adopter")],
        method = "class", parms = list(split = "information"))

# Predict and evaluate
pred_tree_friend_info <-
  predict(tree_friend_info, data_test[, friend_info_features], type = "class")

confusionMatrix(data = pred_tree_friend_info,
```

```
                reference = data_test$adopter,
                mode = "prec_recall", positive = '1')
```

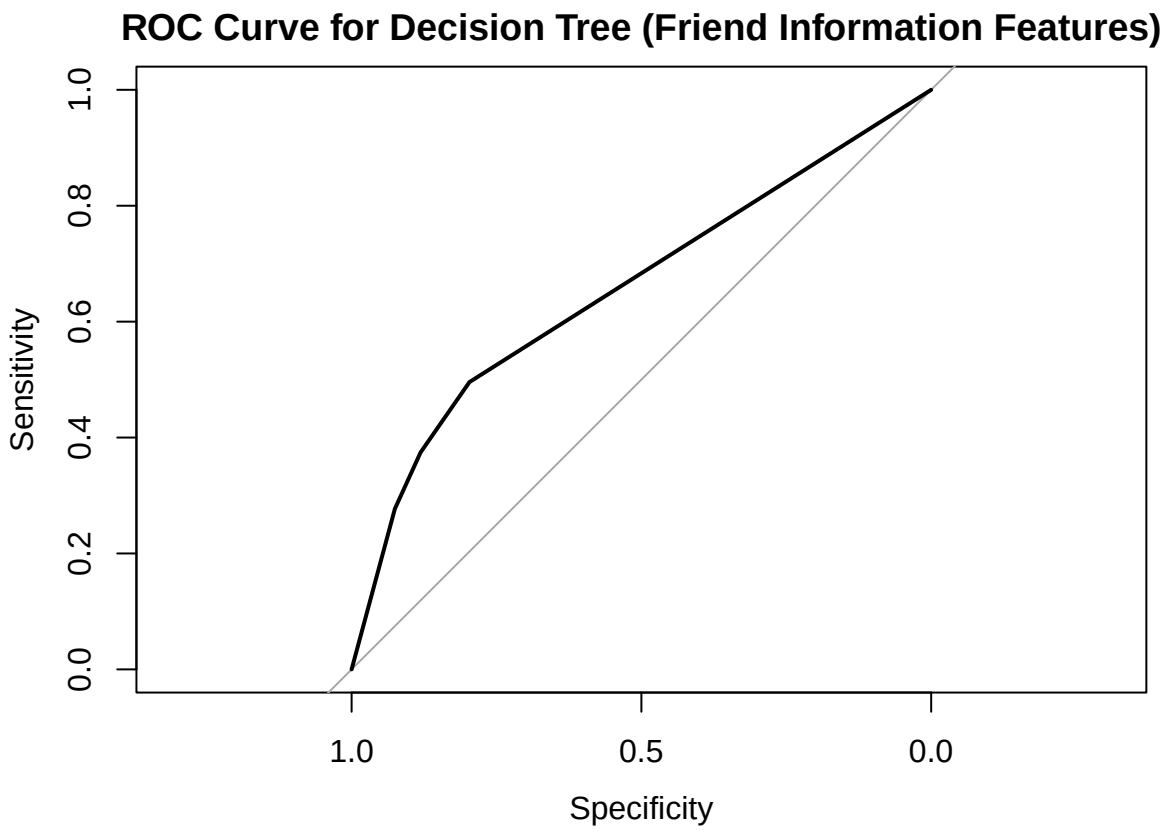
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     0     1
##          0 10571   289
##          1  1429   173
##
##                Accuracy : 0.8621
##                  95% CI : (0.856, 0.8681)
##     No Information Rate : 0.9629
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.1168
##
##  Mcnemar's Test P-Value : <2e-16
##
##               Precision : 0.10799
##                  Recall : 0.37446
##                      F1 : 0.16764
##              Prevalence : 0.03707
##          Detection Rate : 0.01388
##    Detection Prevalence : 0.12855
##       Balanced Accuracy : 0.62769
##
##        'Positive' Class : 1
##
```

```
# Calculate AUC
predicted_probs_tree_fi <-
  predict(tree_friend_info,
          data_test[, friend_info_features], type = "prob")[, 2]
roc_tree_friend_info <-
  roc(data_test$adopter, predicted_probs_tree_fi)
```

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

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

```
plot(roc_tree_friend_info,
     main = "ROC Curve for Decision Tree (Friend Information Features)")
```




```
auc_tree_friend_info <- auc(roc_tree_friend_info)
print(paste("AUC (Friend Information - Decision Tree):", auc_tree_friend_info))
```

```
## [1] "AUC (Friend Information - Decision Tree): 0.657268398268398"
```

Counterintuitively, when we select the features intentionally by categories, the performance (AUC) dropped. Instead, we'll stick with the scientific method and sequentially add features based on their importance ranking from the Filter Approach, assessing the AUC performance for each configuration (e.g., using only the top feature, then the top two, and so on, up to the top 25 features).

3-2-3. Decision Tree Model (sequentially add features based on their importance ranking from the Filter Approach)

```
# List to store AUC results for Decision Tree
tree_auc_results <- data.frame(
  Top_features = character(),
  AUC = numeric(),
  stringsAsFactors = FALSE
)

# Iterate through the top 1 to 25 features
for (i in 1:25) {
  # Select top i features
```

```
  top_features_i <- head(top_features, i)

  # Create a formula for the model
  formula_tree <-
    as.formula(paste("adopter ~", paste(top_features_i, collapse = " + ")))

  # Train Decision Tree Model
  tree_model <-
    rpart(formula_tree,
          data = oversampled_data_train,
          method = "class",
          parms = list(split = "information"),
          na.action = na.omit)

  # Predict probabilities and evaluate
  predicted_probs_tree <-
    predict(tree_model,
            newdata = data_test[, top_features_i, drop = FALSE],
            type = "prob")[, 2]

  # Calculate AUC
  roc_tree <- roc(data_test$adopter, predicted_probs_tree)
  auc_tree <- auc(roc_tree)

  # Store the result in the data frame
  tree_auc_results <- rbind(
    tree_auc_results,
    data.frame(Top_features = paste("top", i, "feature(s)", sep = " "), AUC = auc_tree)
  )
}
```

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases
```

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases
```

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases
```

```
# Print the final data frame
print(tree_auc_results)
```

```
##         Top_features       AUC
## 1   top 1 feature(s) 0.6969444
## 2   top 2 feature(s) 0.7006887
## 3   top 3 feature(s) 0.7006887
## 4   top 4 feature(s) 0.7006887
## 5   top 5 feature(s) 0.7006887
## 6   top 6 feature(s) 0.7068404
## 7   top 7 feature(s) 0.7068404
## 8   top 8 feature(s) 0.7068404
## 9   top 9 feature(s) 0.7068404
## 10 top 10 feature(s) 0.7068404
## 11 top 11 feature(s) 0.7068404
## 12 top 12 feature(s) 0.7068404
## 13 top 13 feature(s) 0.7068404
```

```
## 14 top 14 feature(s) 0.7070258
## 15 top 15 feature(s) 0.7070258
## 16 top 16 feature(s) 0.7070258
## 17 top 17 feature(s) 0.7070258
## 18 top 18 feature(s) 0.7098488
## 19 top 19 feature(s) 0.7098488
## 20 top 20 feature(s) 0.7098488
## 21 top 21 feature(s) 0.7098488
## 22 top 22 feature(s) 0.7098488
## 23 top 23 feature(s) 0.7098488
## 24 top 24 feature(s) 0.7098488
## 25 top 25 feature(s) 0.7098488
```

As we can see, AUC performance peaks at 0.7098488 when using the top 18 features. However, overall AUC performance stabilizes at roughly 0.7 when using only the top two features, "lovedTracks" and "delta_songsListened," achieving an AUC of 0.7006887. Beyond this point, the added complexity yields minimal additional benefit.

Given our feature selection goals, retaining 18 of the 25 features does not improve interpretability.

Therefore, for building and selecting our final model, we will use only the top two features, "lovedTracks" and "delta_songsListened".

4 Finalize Model Tuning

Finally, we will build a decision tree model with only the top 2 important features, perform 5-fold cross validation to evaluate the credibility of our model's performance, and further adjust on how we will be pruning the final model.

4-1. Decision Tree Model with only the top 2 features, tested with 5 fold cross validation

```
set.seed(2)
AUC_cv_tree = c()
cv_tree = createFolds(y = data$adopter, k = 5)

# Define the top 2 features
selected_cols <- c( "lovedTracks", "delta_songsListened")

for (test_rows in cv_tree) {
  dataTrain = data[-test_rows, ]
  dataTest = data[test_rows, ]

  # Oversample the training data
  dataTrain <-
    ovun.sample(adopter ~ ., data = dataTrain, method = "over", seed = 123)$data

  # Filter selected columns (excluding "adopter")
  dataTrain_filtered <- dataTrain[, c(selected_cols, "adopter")]
  dataTest_filtered <- dataTest[, c(selected_cols, "adopter")]

  # Train the Decision Tree model
  tree_f = rpart(adopter ~ ., data = dataTrain_filtered,
```

```
                  method = "class",
                  parms = list(split = "information"))

  # Make predictions
  pred_tree_f =
    predict(tree_f, dataTest_filtered, type = "prob")[, "1"]
  # Assuming "1" is the positive class

  roc_tree_f =
    roc(response = dataTest_filtered$adopter, predictor = pred_tree_f)

  # Store AUC
  AUC_cv_tree = c(AUC_cv_tree, auc(roc_tree_f))
}
```

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases
```

```
mean_AUC_tree = mean(AUC_cv_tree)
cat("Mean AUC for Decision Tree across 5 folds:", mean_AUC_tree, "\n")
```

```
## Mean AUC for Decision Tree across 5 folds: 0.7230672
```

As we can see, according to the 5-fold cross validation, our decision tree using the top 2 most important features have an average AUC performance around 0.7230672. We can be confident in only using the top 2 most important features in our final model.

The final step is to see if adjusting pruning parameters affect our AUC performance. We'll start with assessing the AUC of the model without any further pruning settings.

4-2. Use Decision Tree Model with only the top 2 features on our validation dataset with no further pruning settings

```r
# Load and prepare the dataset
data = read.csv("/Users/yenyihsiang/Desktop/UMN/Fall Semester/Intro to BA in R/Group9_Homework2_Code_fi

# Convert categorical variables into factors
data$male <- as.factor(data$male)
data$good_country <- as.factor(data$good_country)
data$delta_good_country <- as.factor(data$delta_good_country)
data$adopter <- as.factor(data$adopter)  # Ensure adopter is a factor
data$user_id <- NULL  # Remove user_id since it is not a predictive feature

# Create training and testing datasets
set.seed(1)  # For reproducibility
train_rows = createDataPartition(y = data$adopter, p = 0.70, list = FALSE)
data_train = data[train_rows, ]
data_test = data[-train_rows, ]

# Oversample the training data
oversampled_data <- ovun.sample(adopter ~ ., data = data_train,
                                method = "over", p = 0.33, seed = 1)
oversampled_data_train = oversampled_data$data

# Define the top 2 features
selected_cols <- c( "lovedTracks", "delta_songsListened")

# Ensure that the selected columns exist in the oversampled dataset
missing_cols <- setdiff(selected_cols, colnames(oversampled_data_train))
if (length(missing_cols) > 0) {
  stop(paste("The following selected columns are
             missing in the oversampled dataset:",
             paste(missing_cols, collapse = ", ")))
}

# Fit the Decision Tree model with no pruning
tree_model_no_pruning <-
  rpart(adopter ~ .,
        data =
          oversampled_data_train[, c(selected_cols, "adopter")],
                                method = "class",
        parms = list(split = "information"))

# Make predictions on the validation dataset
predicted_probs_no_pruning <-
  predict(tree_model_no_pruning,
          data_test[, selected_cols], type = "prob")[, "1"]

# Calculate ROC and AUC
roc_no_pruning <-
  roc(response = data_test$adopter, predictor = predicted_probs_no_pruning)
```

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

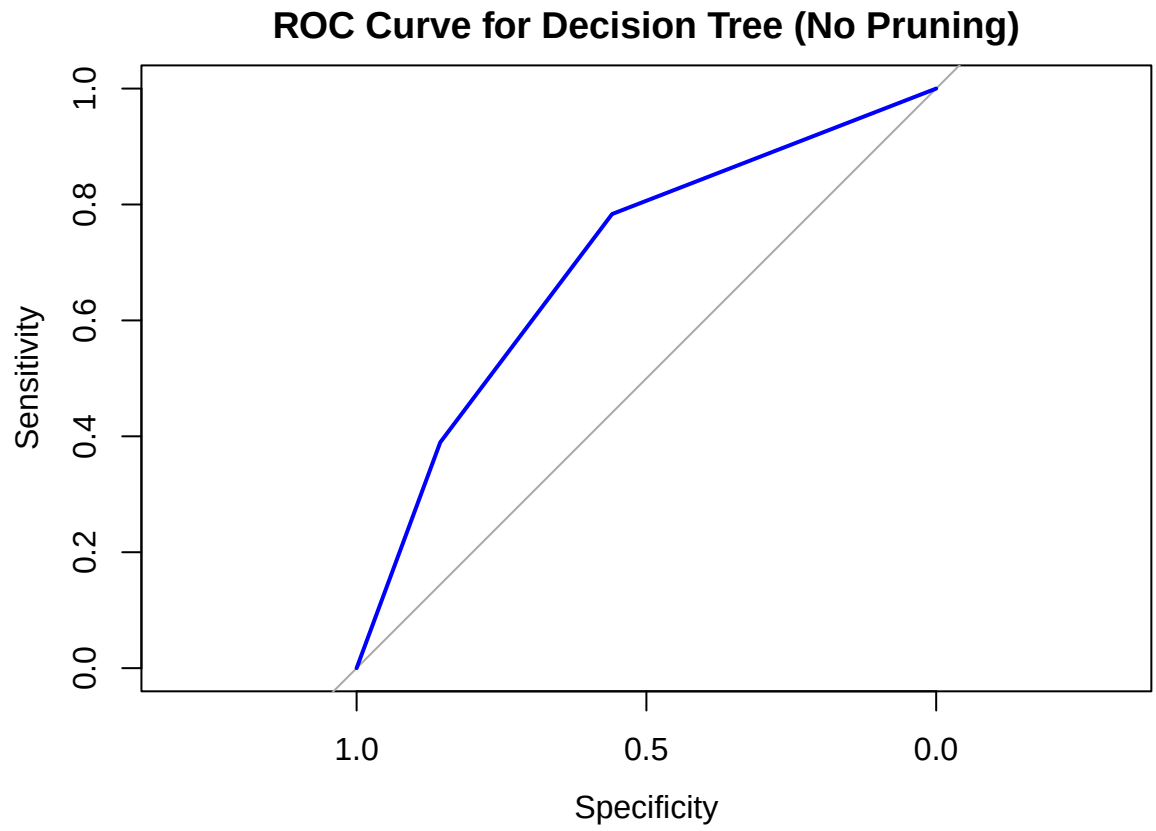
```
## Setting direction: controls < cases
```

```r
auc_no_pruning <-
  auc(roc_no_pruning)

# Print AUC result
cat("AUC for Decision Tree Model with no pruning on validation dataset:",
    auc_no_pruning, "\n")
```

```
## AUC for Decision Tree Model with no pruning on validation dataset: 0.7006887
```

```r
# Plot ROC curve
plot(roc_no_pruning, main = "ROC Curve for Decision Tree (No Pruning)",
     col = "blue", lwd = 2)
```



Now, we'll explore how different pruning parameters impact AUC performance. We'll test various combinations of minsplit and maxdepth values to observe how AUC performance varies.

4-3. Find the best pruning settings (minsplit/maxdepth)

```r
# Hyperparameter tuning grid
# explore minsplit values from 2~50 by increments of 2
minsplit_values <- seq(2, 50, by = 2)
# explore maxdepth values from 3~10 by increments of 1
maxdepth_values <- seq(3, 10, by = 1)
```

```r
# Initialize a data frame to store results with the appropriate length
results <-
  data.frame(minsplit =
               integer(length(minsplit_values) * length(maxdepth_values)),
             maxdepth =
               integer(length(minsplit_values) * length(maxdepth_values)),
             AUC = numeric(length(minsplit_values) * length(maxdepth_values)))

# Counter to fill in the results data frame
counter <- 1

# Define the top 2 features
selected_cols <- c( "lovedTracks", "delta_songsListened")

# Perform hyperparameter tuning
for (minsplit in minsplit_values) {
  for (maxdepth in maxdepth_values) {
    # Train the Decision Tree model with the specified parameters
    tree_model <- rpart(adopter ~ .,
                        data =
                          oversampled_data_train[, c(selected_cols, "adopter")],
                        method = "class",
                        control = rpart.control(minsplit = minsplit,
                                                maxdepth = maxdepth))

    # Make predictions on the validation dataset
    predicted_probs <-
      predict(tree_model, data_test[, selected_cols], type = "prob")[, "1"]

    # Calculate ROC and AUC
    roc_result <-
      roc(response = data_test$adopter, predictor = predicted_probs)
    auc_value <-
      auc(roc_result)

    # Store results
    results[counter, ] <- c(minsplit, maxdepth, auc_value)
    counter <- counter + 1
  }
}
```

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases
```

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases

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

## Setting direction: controls < cases
```

```
# Find the best parameters based on AUC
best_parameters <- results[which.max(results$AUC), ]
cat("Best Parameters:\n")
```

```
## Best Parameters:
```

```
print(best_parameters)
```

```
##   minsplit maxdepth       AUC
## 1        2        3 0.7006887
```

As shown in the output dataframe, the best parameter is to set minsplit as 2 and max depth as 3, which gives us an AUC at 0.7006887 As a result, this will be our settings for our final model.

Let's start building & using our final decision tree model.

4-4. Update our final decision tree model with minsplit = 2 & max depth = 3

```
# Load and prepare the dataset
data = read.csv("/Users/yenyihsiang/Desktop/UMN/Fall Semester/Intro to BA in R/Group9_Homework2_Code_fi

# Convert categorical variables into factors
data$male <- as.factor(data$male)
data$good_country <- as.factor(data$good_country)
data$delta_good_country <- as.factor(data$delta_good_country)
data$adopter <- as.factor(data$adopter)  # Ensure adopter is a factor
data$user_id <- NULL  # Remove user_id since it is not a predictive feature

# Create training and testing datasets
```

```r
set.seed(1)  # For reproducibility
train_rows = createDataPartition(y = data$adopter, p = 0.70, list = FALSE)
data_train = data[train_rows, ]
data_test = data[-train_rows, ]

# Oversample the training data
oversampled_data <- ovun.sample(adopter ~ ., data = data_train,
                                method = "over", p = 0.33, seed = 1)
oversampled_data_train = oversampled_data$data

# Define the top 2 features
selected_cols <- c( "lovedTracks", "delta_songsListened")

# Ensure that the selected columns exist in the oversampled dataset
missing_cols <- setdiff(selected_cols, colnames(oversampled_data_train))
if (length(missing_cols) > 0) {
  stop(paste("The following selected columns are
             missing in the oversampled dataset:",
             paste(missing_cols, collapse = ", ")))
}

# Fit the Decision Tree model with pruning (minsplit = 2, maxdepth = 3)
tree_model_pruned <- rpart(adopter ~ .,
                           data =
                             oversampled_data_train[,
                                                    c(selected_cols,
                                                      "adopter")],
                           method = "class",
                           control = rpart.control(minsplit = 2,
                                                   maxdepth = 3))

# Make predictions on the validation dataset
predicted_probs_pruned <-
  predict(tree_model_pruned, data_test[, selected_cols], type = "prob")[, "1"]

# Convert predicted probabilities to class predictions
# (0 or 1) using a threshold (e.g., 0.5)
predicted_classes_pruned <-
  ifelse(predicted_probs_pruned > 0.5, "1", "0")
predicted_classes_pruned <-
  as.factor(predicted_classes_pruned)

# Calculate ROC and AUC
roc_pruned <-
  roc(response = data_test$adopter, predictor = predicted_probs_pruned)
```

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

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

```r
auc_pruned <-
  auc(roc_pruned)
```

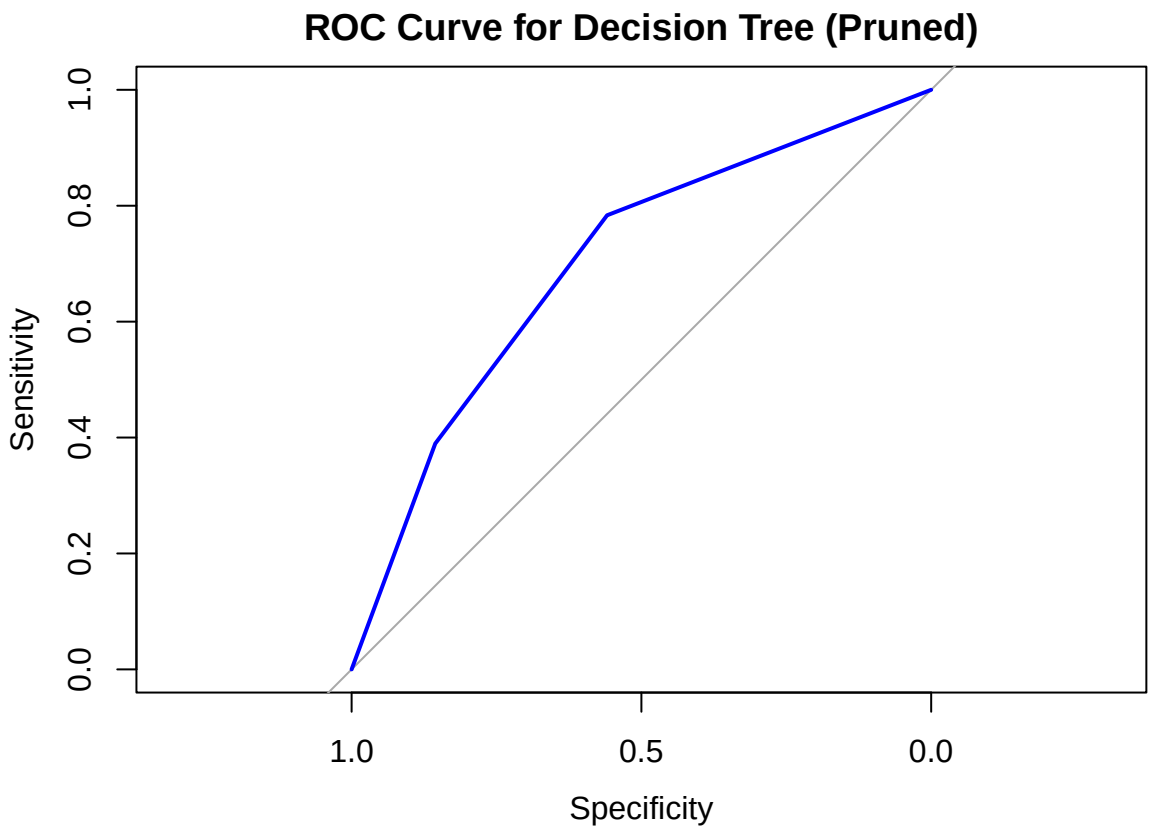
```r
# Generate and print the confusion matrix with detailed statistics
conf_matrix <-
  confusionMatrix(predicted_classes_pruned, data_test$adopter,
                                positive = "1", mode = "prec_recall")
print(conf_matrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     0     1
##          0 10268   282
##          1  1732   180
##
##                Accuracy : 0.8384
##                  95% CI : (0.8318, 0.8448)
##     No Information Rate : 0.9629
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.0978
##
##  Mcnemar's Test P-Value : <2e-16
##
##               Precision : 0.09414
##                  Recall : 0.38961
##                      F1 : 0.15164
##              Prevalence : 0.03707
##          Detection Rate : 0.01444
##    Detection Prevalence : 0.15343
##       Balanced Accuracy : 0.62264
##
##        'Positive' Class : 1
##
```

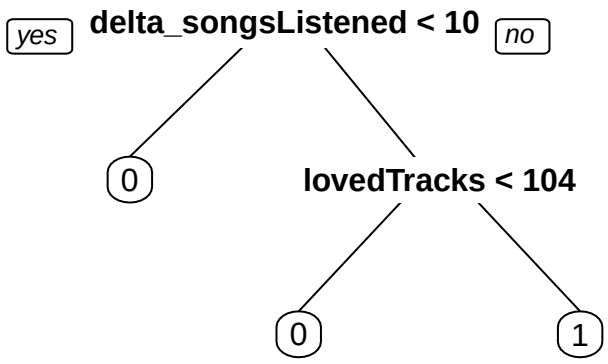
```r
# Print AUC result
cat("AUC for Decision Tree Model with pruning on validation dataset:",
    auc_pruned, "\n")
```

```
## AUC for Decision Tree Model with pruning on validation dataset: 0.7006887
```

```r
# Plot ROC curve
plot(roc_pruned, main = "ROC Curve for Decision Tree (Pruned)",
     col = "blue", lwd = 2)
```




```
# The final model we will be using is
prp(tree_model_pruned, varlen = 0)
```



That concludes our technical document. Below, we have included an appendix featuring our other attempts to explore this dataset. These attempts include applying various oversampling functions, exploring k-NN models, random forest models, logistic regression models, and performing both forward and backward feature selection. Although we ultimately decided not to use any of the codes, we are including them only for the purpose of documenting our progress.

However, we will comment out the code due to the length of the document; it is included solely for reference. Note, the appendix codes are provided by Harman Singh and Faithan To.

Appendix Code

k-NN (k=5) Model w/ Feature Selection, 5-Fold Cross-Validation with a different oversampling function "ROSE"

```
# Data processing
# data = read.csv("XYZData.csv", sep = ",", header = TRUE)

# Ensure the nominal variables are set to factor
# data$male <- as.factor(data$male)
# data$good_country <- as.factor(data$good_country)
# data$delta_good_country <- as.factor(data$delta_good_country)
# data$adopter <- as.factor(data$adopter)

# Separate the nominal variables from the numeric ones (and adopter)
# nominal_vars <- data %>% dplyr::select(male, good_country, delta_good_country)
```

```
# continuous_vars <-
#    data %>% dplyr::select(-user_id, -male, -good_country, -delta_good_country)

# Group variables by type
# data_grouped <- cbind(nominal_vars, continuous_vars)

# Remove the non-significant predictors
# 2 predictor variables left
# 1:2 are continuous variables
# 3 is the dependent variable adopter
# data_grouped2 <- data_grouped %>%
#   dplyr::select(lovedTracks, delta_songsListened, adopter)

# Use the createFolds function to generate a list of row indexes
# that correspond to each of the 5 folds
# cv = createFolds(y = data_grouped2$adopter, k = 5)

# recall_cv = c()
# precision_cv = c()
# f1_cv = c()
# auc_values = c()

# Loop through each fold
# for (test_rows in cv)
# {
#   # Split data into training data and testing (validation) data
#   data_grouped_train = data_grouped2[-test_rows,]
#   data_grouped_test = data_grouped2[test_rows,]
#   #View(data_grouped_test)
#
#  # Create oversampled data for the training data
#  # ROSE is better than SMOTE for complex and non-linear data
#  data_grouped_train_oversampled <-
#                   ROSE(adopter ~ ., data = data_grouped_train, seed = 2)$data
#  #View(data_grouped_train_oversampled)
#
#  # Check the class distribution of the factor variable "adopter"
#   for the oversampled data
#  #print(table(data_grouped_train_oversampled$adopter))

# # Normalize oversampled data

#  # Create function to normalize
#  normalize = function(x){return ((x - min(x))/(max(x) - min(x)))}

#  # Normalize the columns with continuous variables
#  data_grouped_train_oversampled_normalized <-
#                  data_grouped_train_oversampled %>% mutate_at(1:2, normalize)
#  #View(data_grouped_train_oversampled_normalized)

#  # Train model
#  knn_model = knn3(x = data_grouped_train_oversampled_normalized[,1:2],
#                   y = data_grouped_train_oversampled_normalized[,3],
```

```
#                   k = 5)

#  # Normalize the test data using the mean and standard deviation of the
#    oversampled training data
#  data_grouped_train_oversampled_mean <-
#                   colMeans(data_grouped_train_oversampled[,1:2], na.rm = TRUE)
#  data_grouped_train_oversampled_sd <-
#              apply(data_grouped_train_oversampled[,1:2], 2, sd, na.rm = TRUE)
#  normalize2 =
#  function(x){return
# ((x - data_grouped_train_oversampled_mean)/data_grouped_train_oversampled_sd)}

#  data_grouped_test_normalized <-
#                                data_grouped_test%>% mutate_at(1:2, normalize2)

#  # Predict using model
#  knn_model_predictions =
#         predict(knn_model, data_grouped_test_normalized[,1:2], type = "class")
#  knn_model_predictions_probabilities =
#          predict(knn_model, data_grouped_test_normalized[,1:2], type = "prob")
#  #View(knn_model_predictions_probabilities)

# # Evaluate model performance
#  true_labels = data_grouped_test_normalized[,3]
#  #print(levels(true_labels))

#  confusion_matrix = confusionMatrix(data = knn_model_predictions,
#                                          reference = true_labels,
#                                                  positive = "1",
#                                              mode = "prec_recall")

#  # Extract performance metric
#  recall = confusion_matrix$byClass['Recall']
#  precision = confusion_matrix$byClass['Precision']
#  f1 = confusion_matrix$byClass['F1']

#  # Store the performance metric
#  recall_cv = c(recall_cv, recall)
#  precision_cv = c(precision_cv, precision)
#  f1_cv = c(f1_cv, f1)

#  # Calculate ROC and AUC
#  roc_NaiveBayes <-
#    roc(response = data_grouped_test_normalized$adopter,
#                        predictor = knn_model_predictions_probabilities[, "1"])
#  auc_NaiveBayes <-
#    auc(roc_NaiveBayes)

#  auc_values = c(auc_values, auc_NaiveBayes)
#}

# cat('Mean Recall:', mean(recall_cv), '\n')
# cat('Mean Precision:', mean(precision_cv), '\n')
```

```
# cat('Mean F1:', mean(f1_cv), '\n')
# cat('Mean AUC: ', mean(auc_values), '\n')
```

Feature Selection: Forward Selection

```
# library(MASS)
#
# set.seed(123) # for reproducibility
# training_rows <-
#               createDataPartition(data_grouped$adopter, p = 0.7, list = FALSE)
#
# Split data into training data and testing (validation) data
# data_grouped_train = data_grouped[training_rows,]
# data_grouped_test = data_grouped[-training_rows,]
#
# initial_model <-
#          glm(adopter ~ 1, data = data_grouped_train[1:26], family = binomial)
# full_model <-
#         glm(adopter ~ ., data = data_grouped_train[1:26], family = binomial)
#
# forward_model <- stepAIC(initial_model,
#                          scope = list(lower = initial_model, upper = full_model),
#                            direction = "forward",
#                            trace = FALSE)
#
# # Check model summary
# summary(forward_model)
```

Feature Selection: Backward Elimination

```
# library(MASS)
#
# set.seed(123) # for reproducibility
# training_rows <-
#              createDataPartition(data_grouped$adopter, p = 0.7, list = FALSE)
#
# Split data into training data and testing (validation) data
# data_grouped_train = data_grouped[training_rows,]
# data_grouped_test = data_grouped[-training_rows,]
#
# full_model <-
#           glm(adopter ~ ., data = data_grouped_train[1:26], family = binomial)
#
# backward_model <- stepAIC(full_model, trace = FALSE)
#
# Check model summary
# summary(backward_model)
```

Build and Evaluate a Random Forest Model with a different oversampling function "SMOTE"

```
# knitr::opts_chunk$set(echo = TRUE)
# options(repos = c(CRAN = "https://cran.r-project.org"))
# Import the dataset
# library(dplyr)
# xyz <-
#  read.csv("/Users/harmansingh/Downloads/XYZData.csv", stringsAsFactors = TRUE)

# Remove non-predictive columns (user id) and convert class
# colnames(xyz)
# library(dplyr)
# xyz <- xyz %>% dplyr::select(-user_id)
# xyz <- xyz %>%
#   mutate_at(c("adopter","male","good_country"), as.factor)

# randomly permutate rows of the entire dataset
# set.seed(123)
# xyz_rand <- xyz[sample(nrow(xyz)),]`

# split train/test/valid
# use 70% training and 30% validation/testing split by row indexes
# library(caret)
# train_rows <-
#            createDataPartition(y = xyz_rand$adopter, p = 0.70, list = FALSE)
# xyz_rand_train <- xyz_rand[train_rows,]
# xyz_rand_testValid <- xyz_rand[-train_rows,]

# Further split the remaining data into validation and test, 50% each
# testIndex <- createDataPartition(xyz_rand_testValid$adopter, p = 0.5,
#                                  list = FALSE)
# xyz_rand_valid <- xyz_rand_testValid[testIndex, ]
# xyz_rand_test <- xyz_rand_testValid[-testIndex, ]

# SMOTE oversampling to balance class distribution
# install.packages("remotes")
# remotes::install_github("dongyuanwu/RSBID")
# library(RSBID)
# smote_result <- SMOTE_NC(xyz_rand_train, "adopter", k = 5)
# The desired percentage of the size of majority samples that the minority
# samples would be reached in the new dataset. The default is 100.
# nrow(smote_result[smote_result$adopter == 1,])

# Choose variables
# smote_result2 <- dplyr::select(smote_result, -friend_cnt, -avg_friend_male,
#                            -delta_avg_friend_male, -delta_friend_country_cnt
#                               , -delta_subscriber_friend_cnt, -delta_posts,
#                               -delta_playlists, -delta_shouts,
#                               -delta_good_country)
# xyz_rand_testValid2 <- dplyr::select(xyz_rand_testValid, -friend_cnt,
#                                      -avg_friend_male, -delta_avg_friend_male,
#                                      -delta_friend_country_cnt,
#                                      -delta_subscriber_friend_cnt,
```

```
#                                     -delta_posts,
#                                     -delta_playlists, -delta_shouts,
#                                     -delta_good_country)

# Build and evaluate Random Forest
# library(randomForest)
# set.seed(123) # For reproducibility
# rf_xyz2 <- randomForest(adopter ~ ., data = smote_result2,
#                         importance = TRUE, ntree = 500)
# Extract and View Variable Importance
# importance(rf_xyz2)
# matrix of importance scores for each variable, including Mean Decrease
# Accuracy and Mean Decrease Gini
# varImpPlot(rf_xyz2)
# displays a plot of variable importance, showing the top
# variables based on their importance scores

# Evaluate the Random Forest
# xyz_rand_testValid2 <- xyz_rand_testValid2 %>%
#   mutate_at(c("adopter","male","good_country"), as.factor)
# forest_predictions <- predict(rf_xyz2, xyz_rand_testValid2, type = "class")

# Confusion matrix
# confusion_matrix <- table(xyz_rand_testValid2$adopter, forest_predictions)
# print(confusion_matrix)
```

Build and Evaluate a Logistic Regression Model with oversample function SMOTE

```
# smote_result4 <- smote_result %>%
#  dplyr::select(male, avg_friend_age, subscriber_friend_cnt, songsListened,
#                playlists, delta_lovedTracks, tenure, adopter)
# xyz_rand_valid4 <- xyz_rand_valid %>%
#  dplyr::select(male, avg_friend_age, subscriber_friend_cnt, songsListened,
#                playlists, delta_lovedTracks, tenure, adopter)
# xyz_rand_test4 <- xyz_rand_test %>%
#  dplyr::select(male, avg_friend_age, subscriber_friend_cnt, songsListened,
#                playlists, delta_lovedTracks, tenure, adopter)
# regression_model <- glm(adopter ~ male + avg_friend_age + subscriber_friend_cnt
#                         + songsListened + playlists + delta_lovedTracks
#                         + tenure, data = smote_result4, family = binomial)
# regression_model

# Evaluate Logistic Regression Model
# Load necessary library for evaluation
# library(ROCR)
# library(caret)

# Predict probabilities on test data
# xyz_rand_test4 <- xyz_rand_test4 %>%
#  mutate_at(c("male", "adopter"), as.factor)
# test_probs <- predict(regression_model,
#                       newdata = xyz_rand_test4, type = "response")
```

```
# Threshold conversion to likely adopters
# likely_adopters <- ifelse(test_probs > 0.5, 1, 0)  # Adjust threshold if needed

# Show likely adopters
# xyz_rand_test4$likely_adopter <- likely_adopters

# ROC and AUC for test data
# test_pred <- prediction(test_probs, xyz_rand_test4$adopter)
# test_perf <- performance(test_pred, "tpr", "fpr")
# test_auc <- performance(test_pred, "auc")
# test_auc_value <- test_auc@y.values[[1]]
# cat("test auc:", test_auc_value, "\n")

# F1 Score for test data
# test_predicted_classes <- ifelse(test_probs > 0.5, 1, 0)
# test_conf_matrix <- confusionMatrix(factor(test_predicted_classes),
#                                     xyz_rand_test4$adopter)
# test_f1_score <- test_conf_matrix$byClass["F1"]
# cat("F1 test score:", test_f1_score, "\n")
```