DATA 621 Homework 2

Warner Alexis, Saloua Daouki, Souleymane Doumbia, Fomba Kassoh, Lewris Mota Sanchez

2024-10-08

Contents

Abstract

This assignment explores the application of various classification models to a dataset, with an emphasis on data preparation, model selection, and performance evaluation. The methodologies employed include transforming variables, calculating statistical metrics, and utilizing different modeling approaches. Despite some interesting findings, the analysis revealed areas needing clarification, particularly in the justification of methods and criteria for model comparison. This work aims to provide a comprehensive overview of the steps taken and the rationale behind each choice, ultimately leading to a better understanding of the model-building process and its implications for predictive analytics.

Introduction

In the field of data science, classification models play a fundamental role in predicting categorical outcomes, making them indispensable tools in a wide range of applications, from medical diagnosis to customer

segmentation. This assignment delves into the key metrics used to evaluate classification models, focusing on both the theoretical underpinnings and practical implementation of these metrics in R. By analyzing the performance of a binary classification model, this work aims to build a comprehensive understanding of core classification metrics and their implications for model evaluation.

The dataset used contains observed and predicted class labels, as well as predicted probabilities, which provide a foundation for calculating various classification metrics. The primary objectives of this assignment include implementing functions in R to compute metrics such as accuracy, error rate, precision, sensitivity (recall), specificity, and the F1 score, and examining how these metrics reveal different aspects of model performance. Furthermore, the assignment requires the generation of an ROC curve and calculation of the Area Under the Curve (AUC), which are valuable tools for assessing the overall effectiveness of the model in distinguishing between classes.

To enhance this analysis, packages such as `caret` and `pROC` are explored for their built-in functions, which serve as benchmarks for comparing the results of custom-built functions. This comparison provides a basis for understanding both the underlying logic of these metrics and the potential variations that may arise from different computational methods.

Through this investigation, the assignment not only emphasizes the importance of accurate classification but also highlights the practical aspects of working with classification models in R. By creating graphical outputs and systematically evaluating metrics, this work seeks to enhance understanding of how various performance measures contribute to a more nuanced assessment of model efficacy and reliability in predictive analytics.

Load the libraries

```
# Load required libraries
library('tidyverse')
library(ggplot2)
library(dplyr)
library(caret)
library(pROC)
library(gridExtra)
```

1. Read the classification output data set

```
classification_data <- read.csv('https://raw.githubusercontent.com/hawa1983/DATA-621/refs/heads/main/Ho
```

2. The data set has three key columns we will use:

- ***class:*** the actual class for the observation
- ***scored.class:*** the predicted class for the observation (based on a threshold of 0.5)
- ***scored.probability:*** the predicted probability of success for the observation

2.1. Preview the dataset

```
head(classification_data) # checking column names and data types
```

```
##   pregnant glucose diastolic skinfold insulin  bmi pedigree age class
## 1        7     124        70       33     215 25.5    0.161  37     0
## 2        2     122        76       27     200 35.9    0.483  26     0
## 3        3     107        62       13      48 22.9    0.678  23     1
## 4        1      91        64       24       0 29.2    0.192  21     0
## 5        4      83        86       19       0 29.3    0.317  34     0
## 6        1     100        74       12      46 19.5    0.149  28     0
```

```
##   scored.class scored.probability
## 1            0         0.32845226
## 2            0         0.27319044
## 3            0         0.10966039
## 4            0         0.05599835
## 5            0         0.10049072
## 6            0         0.05515460
```

```
str(classification_data) #a quick look at the structure and data types
```

```
## 'data.frame':    181 obs. of  11 variables:
##  $ pregnant          : int  7 2 3 1 4 1 9 8 1 2 ...
##  $ glucose           : int  124 122 107 91 83 100 89 120 79 123 ...
##  $ diastolic         : int  70 76 62 64 86 74 62 78 60 48 ...
##  $ skinfold          : int  33 27 13 24 19 12 0 0 42 32 ...
##  $ insulin           : int  215 200 48 0 0 46 0 0 48 165 ...
##  $ bmi               : num  25.5 35.9 22.9 29.2 29.3 19.5 22.5 25 43.5 42.1 ...
##  $ pedigree          : num  0.161 0.483 0.678 0.192 0.317 0.149 0.142 0.409 0.678 0.52 ...
##  $ age               : int  37 26 23 21 34 28 33 64 23 26 ...
##  $ class             : int  0 0 1 0 0 0 0 0 0 0 ...
##  $ scored.class      : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ scored.probability: num  0.328 0.273 0.11 0.056 0.1 ...
```

2.2. Use the table() function to get the raw confusion matrix for this scored dataset.

This confusion matrix output provides a comprehensive assessment of a binary classification model's performance by summarizing the accuracy of its predictions and various important statistical metrics. The confusion matrix itself shows how the model performs in predicting two classes: class 0 (negative) and class 1 (positive).

Confusion Matrix

```
# Create confusion matrix using the table() function
conf_matrix <- table(Predicted = classification_data$scored.class, Actual = classification_data$class)
# Print confusion matrix with definitions for clarity
cat("Confusion Matrix:\n")
```

```
## Confusion Matrix:
```

```
conf_matrix
```

```
##          Actual
## Predicted   0   1
##         0 119  30
##         1   5  27
```

Confusion Matrix with Values and Definitions

```
conf_matrix_df <- as.data.frame(conf_matrix) %>%
  mutate(Value = c(conf_matrix["0", "0"], conf_matrix["1", "0"], conf_matrix["0", "1"], conf_matrix["1"
         Definition = c(
           "True Negatives (TN): Correctly predicted 'No'",
           "False Positives (FP): Incorrectly predicted 'Yes'",
           "False Negatives (FN): Incorrectly predicted 'No'",
           "True Positives (TP): Correctly predicted 'Yes'"
         ))
conf_matrix_df
```

```
##   Predicted Actual Freq Value                                          Definition
## 1         0      0  119   119     True Negatives (TN): Correctly predicted 'No'
## 2         1      0    5     5 False Positives (FP): Incorrectly predicted 'Yes'
## 3         0      1   30    30  False Negatives (FN): Incorrectly predicted 'No'
## 4         1      1   27    27    True Positives (TP): Correctly predicted 'Yes'
```

```r
# Extract individual values from the confusion matrix
TN <- conf_matrix["0", "0"]
FP <- conf_matrix["1", "0"]
FN <- conf_matrix["0", "1"]
TP <- conf_matrix["1", "1"]

# Convert confusion matrix to a data frame
conf_matrix_df <- as.data.frame(conf_matrix)

# Add a new column with the values TN, FP, FN, TP
conf_matrix_df$Value <- c(TN, FP, FN, TP)

# Add a new column with concise definitions of each term
conf_matrix_df$Definition <- c(
  "True Negatives (TN): Correctly predicted 'No'",
  "False Positives (FP): Incorrectly predicted 'Yes'",
  "False Negatives (FN): Incorrectly predicted 'No'",
  "True Positives (TP): Correctly predicted 'Yes'"
)

print(conf_matrix)
```

```
##          Actual
## Predicted   0   1
##         0 119  30
##         1   5  27
```

```r
# Print the updated confusion matrix data frame
cat("\nConfusion Matrix with Values and Definitions:\n")
```

```
##
## Confusion Matrix with Values and Definitions:
```

```r
print(conf_matrix_df)
```

```
##   Predicted Actual Freq Value                                          Definition
## 1         0      0  119   119     True Negatives (TN): Correctly predicted 'No'
## 2         1      0    5     5 False Positives (FP): Incorrectly predicted 'Yes'
## 3         0      1   30    30  False Negatives (FN): Incorrectly predicted 'No'
## 4         1      1   27    27    True Positives (TP): Correctly predicted 'Yes'
```

Caret Confusion Matrix

The caret confusion matrix function provides a detailed summary of a classification model's performance by comparing predicted class labels with actual class labels. It calculates key metrics such as accuracy, sensitivity, specificity, precision, and more. These metrics help assess the model's ability to correctly classify data, offering insights into its strengths and weaknesses in predicting positive and negative outcomes.

```r
# Ensure that the actual and predicted class columns are factors
classification_data$class <- factor(classification_data$class)
classification_data$scored.class <- factor(classification_data$scored.class)
```

```r
# We can use 'confucionMatrix' function from the package of 'caret' to automatically extract the table
caret_conf_matrix <- confusionMatrix(classification_data$scored.class, classification_data$class, positi
print(caret_conf_matrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 119  30
##          1   5  27
##
##                Accuracy : 0.8066
##                  95% CI : (0.7415, 0.8615)
##     No Information Rate : 0.6851
##     P-Value [Acc > NIR] : 0.0001712
##
##                   Kappa : 0.4916
##
##  Mcnemar's Test P-Value : 4.976e-05
##
##             Sensitivity : 0.4737
##             Specificity : 0.9597
##          Pos Pred Value : 0.8438
##          Neg Pred Value : 0.7987
##              Prevalence : 0.3149
##          Detection Rate : 0.1492
##    Detection Prevalence : 0.1768
##       Balanced Accuracy : 0.7167
##
##        'Positive' Class : 1
##
```

2.2.1. Understanding the Confusion Matrix Output The confusion matrix is formatted as follows:

- **Rows represent the predicted class** (`scored.class` in the dataset).
  - Row **0** represents all instances where the model predicted class 0.
  - Row **1** represents all instances where the model predicted class 1.
- **Columns represent the actual class** (`class` in the dataset).
  - Column **0** represents all instances where the true class is 0.
  - Column **1** represents all instances where the true class is 1.
- **Top-left (TN)**: This value represents the **True Negatives (TN)**. These are the cases where the model predicted `0`, and the actual class was `0`.
- **Top-right (FN)**: This value represents the **False Negatives (FN)**. These are the cases where the model predicted `0`, but the actual class was `1`.
- **Bottom-left (FP)**: This value represents the **False Positives (FP)**. These are the cases where the model predicted `1`, but the actual class was `0`.
- **Bottom-right (TP)**: This value represents the **True Positives (TP)**. These are the cases where the model predicted `1`, and the actual class was `1`.

The confusion matrix itself shows how the model performs in predicting two classes: class 0 (negative) and class 1 (positive). It reveals that the model correctly classified 119 true negatives (class 0 predicted as 0) and 27 true positives (class 1 predicted as 1). However, it also misclassified 30 false negatives (class 1 predicted as 0) and 5 false positives (class 0 predicted as 1). These classifications allow us to calculate several key metrics that provide deeper insight into the model's performance.

The table() function provides a simple contingency table that summarizes the counts of actual versus predicted classifications, offering a basic overview of the model's performance. While this output is useful for quickly visualizing correct and incorrect predictions, it lacks the detailed metrics needed to fully assess the model. By inspecting the more comprehensive confusion matrix from the caret package, we can delve deeper into specific performance metrics such as sensitivity, accuracy, and specificity.

According to Caret's confusion matrix, the model's overall accuracy is 80.66%, meaning that 80.66% of all predictions are correct. The ***confidence interval*** (CI) of (0.7415, 0.8615) indicates that with 95% confidence, the true accuracy of the model lies between 74.15% and 86.15%. This relatively narrow confidence interval suggests stability in the model's predictions.

The ***No Information Rate*** (NIR) of 68.51% represents the accuracy that would be achieved by always predicting the majority class (class 0). The fact that the model's accuracy is significantly higher than the NIR, with a p-value of 0.0001712, confirms that the model is performing much better than random guessing or a baseline majority-class prediction.

The ***Kappa statistic*** is 0.4916, indicating moderate agreement between the predicted and actual classifications, adjusted for random chance. On the other hand, the ***A McNemar's test*** p-value of 4.976e-05 reveals a statistically significant difference in the error rates between false positives and false negatives, suggesting the model makes more mistakes in one direction than the other, particularly in misclassifying actual positives (class 1) as negatives (class 0).

Sensitivity (47.37%) and ***specificity*** (95.97%) are two crucial metrics in evaluating the model's ability to identify each class. Sensitivity, or recall, reflects the model's ability to identify actual positives, with a relatively low value of 47.37%, meaning that less than half of the actual positive cases are correctly identified. In contrast, the model's specificity is high at 95.97%, showing that it is very successful at correctly identifying negative cases. This imbalance between sensitivity and specificity suggests that the model is more likely to correctly predict negatives than positives, which may be a concern depending on the goals of the classification task.

Other metrics such as ***Positive Predictive Value*** (precision) and ***Negative Predictive Value*** offer further insights. The positive predictive value is 84.38%, indicating that 84.38% of the predicted positive cases are actually positive. Meanwhile, the negative predictive value is 79.87%, suggesting that about 79.87% of the predicted negatives are indeed negative. These values reveal that while the model is relatively good at making correct positive predictions, there is room for improvement in predicting true positives.

Additionally, the ***Balanced Accuracy*** of 71.67% represents the average of sensitivity and specificity, which offers a more balanced view of the model's performance, particularly useful when dealing with class imbalances.

Finally, the analysis highlights that the positive class in this model is class 1, which has a prevalence of 31.49%, reflecting the proportion of actual positive cases in the dataset.

This detailed output provides valuable context for the metrics we will manually compute, in addition to providing a framework to compare and validate our calculations with greater accuracy.

2.2.2. Summary In summary, the model performs well in terms of overall accuracy and specificity, but it struggles with sensitivity, failing to identify a significant portion of the positive cases. The metrics indicate a moderate level of agreement beyond random chance, and further adjustments might be necessary, especially if correctly predicting positive cases is critical to the task.

- **True Negatives (TN)** = 119: The model predicted `0` and the actual class was `0`.
- **False Negatives (FN)** = 30: The model predicted `0`, but the actual class was `1`.
- **False Positives (FP)** = 5: The model predicted `1`, but the actual class was `0`.
- **True Positives (TP)** = 27: The model predicted `1` and the actual class was `1`.

3. Accuracy Calculation

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the accuracy of the predictions.

```
custom_accuracy <- function(data) {
  TP <- sum(data$class == 1 & data$scored.class == 1)
  TN <- sum(data$class == 0 & data$scored.class == 0)
  FP <- sum(data$class == 0 & data$scored.class == 1)
  FN <- sum(data$class == 1 & data$scored.class == 0)

  return((TP + TN) / (TP + FP + TN + FN))

}

accuracy <- custom_accuracy(classification_data)
cat(sprintf("accuracy: %.4f\n", accuracy))
```

```
## accuracy: 0.8066
```

What does accuracy tells us base on the results???

The accuracy of 80.66% indicates that the model correctly classified approximately 81% of the observations in the dataset. This metric gives a general sense of the model's overall effectiveness, showing that it performs reasonably well in terms of total correct classifications.

4. Classification Error Rate

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the classification error rate of the predictions.

```
error_rate <- function(data) {
  TP <- sum(data$class == 1 & data$scored.class == 1)
  TN <- sum(data$class == 0 & data$scored.class == 0)
  FP <- sum(data$class == 0 & data$scored.class == 1)
  FN <- sum(data$class == 1 & data$scored.class == 0)

  return((FP + FN) / (TP + FP + TN + FN))
}

error_rate <- error_rate(classification_data)

# Compare results
cat(sprintf("Classification Error Rate: %.4f\n", error_rate))
```

```
## Classification Error Rate: 0.1934
```

```
cat(sprintf("Sum of Accuracy and Classification Error Rate: %.2f\n", accuracy + error_rate))
```

```
## Sum of Accuracy and Classification Error Rate: 1.00
```

What does this Error Rate tells us???

The classification error rate of 0.1934 indicates that approximately 19.34% of the model's predictions were incorrect. In other words, the model misclassified around one-fifth of the observations in the dataset. This error rate tells us how often the model is making mistakes. Specifically, it reflects the proportion of instances where the model's predicted class did not match the actual class. While the error rate is lower than 20%, which is not excessively high, it still highlights that there is room for improvement in the model's performance.

5. Precision

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the precision of the predictions.

```
custom_precision <- function(data) {
  TP <- sum(data$class == 1 & data$scored.class == 1)
  FP <- sum(data$class == 0 & data$scored.class == 1)
  return(TP / (TP + FP))
}

precision <- custom_precision(classification_data)
cat(sprintf("precision: %.4f\n", precision))
```

```
## precision: 0.8438
```

What does this precision tells us???

The precision of 0.8438 (or 84.38%) indicates that when the model predicts a positive class (1), it is correct 84.38% of the time. In other words, precision reflects the proportion of true positive predictions out of all the instances where the model predicted positive.

6. Sensitivity (Recall)

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the sensitivity of the predictions. Sensitivity is also known as recall.

```
custom_sensitivity <- function(data) {
  TP <- sum(data$class == 1 & data$scored.class == 1)
  FN <- sum(data$class == 1 & data$scored.class == 0)
  return(TP / (TP + FN))
}

sensitivity <- custom_sensitivity(classification_data)
recall = sensitivity

cat(sprintf("sensitivity (recall): %.4f\n", sensitivity))
```

```
## sensitivity (recall): 0.4737
```

What does this precision tells us - what the results are about???

The value 0.4737 (or 47.37%) here is actually the sensitivity (or recall), not precision. This sensitivity score tells us that the model correctly identifies only 47.37% of actual positive cases. In other words, of all the instances where the actual class is positive, the model is correctly predicting positive in about half of those cases. This low sensitivity suggests that the model is not very effective at identifying positive cases and is likely missing a significant number of them.

7. Specificity

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the specificity of the predictions.

```
custom_specificity <- function(data) {
  TN <- sum(data$class == 0 & data$scored.class == 0)
  FP <- sum(data$class == 0 & data$scored.class == 1)
  return(TN / (TN + FP))
}
```

```r
specificity <- custom_specificity(classification_data)

cat(sprintf("specificity: %.4f\n", specificity))
```

```
## specificity: 0.9597
```

8. F1 Score

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the F1 score of the predictions.

```r
# Custom precision function
custom_f1_score <- function(data) {
  return(2 * (precision * sensitivity) / (precision + sensitivity))
}



f1score <- custom_f1_score(classification_data)

cat(sprintf("F1 Score: %.4f\n", f1score))
```

```
## F1 Score: 0.6067
```

What does this score tells us???

The F1 score of 60.67% indicates a moderate balance between the precision (84.38%) and recall (47.37%) of the model. While precision is relatively high, the low recall lowers the F1 score, indicating that the model is missing many true positives even though it has a high accuracy when it does predict positive.

9. F1 Score Lower Bound and Upper Bound

Before we move on, let's consider a question that was asked: What are the bounds on the F1 score? Show that the F1 score will always be between 0 and 1. (Hint: If $0 < a < 1$ and $0 < b < 1$ then $ab < a$.)

Yes, the output below addresses the question regarding the bounds on the F1 score.

Explanation:

Interpretation of the Matrix: - The matrix shown below illustrates different precision and recall values, with each resulting in an F1 score. In all cases, the F1 score lies between 0 and 1. - The test cases: - f1_score(0, 1): Results in 0, which is the lower bound. - f1_score(1, 1): Results in 1, which is the upper bound. - f1_score(0.5, 0.5): Results in a value between 0 and 1, which is a typical case for non-extreme precision and recall values.

This confirms that the F1 score is bounded between 0 and 1, thus answering the question.

```r
# Updated F1 Score function to handle division by zero
f1_score <- function(precision, recall) {
  if (precision == 0 & recall == 0) {
    return(0)  # Return 0 when both precision and recall are 0
  } else {
    return(2 * (precision * recall) / (precision + recall))
  }
}

# Example values of Precision (P) and Recall (R)
```

```r
precision_values <- seq(0, 1, by = 0.1)  # Precision from 0 to 1
recall_values <- seq(0, 1, by = 0.1)  # Recall from 0 to 1

# Create a matrix to store F1 scores
f1_matrix <- matrix(nrow = length(precision_values), ncol = length(recall_values))

# Calculate F1 scores for each combination of precision and recall
for (i in seq_along(precision_values)) {
  for (j in seq_along(recall_values)) {
    f1_matrix[i, j] <- f1_score(precision_values[i], recall_values[j])
  }
}

# Display the matrix of F1 scores
print(f1_matrix)
```

```
##       [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
##  [1,]    0 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##  [2,]    0 0.1000000 0.1333333 0.1500000 0.1600000 0.1666667 0.1714286
##  [3,]    0 0.1333333 0.2000000 0.2400000 0.2666667 0.2857143 0.3000000
##  [4,]    0 0.1500000 0.2400000 0.3000000 0.3428571 0.3750000 0.4000000
##  [5,]    0 0.1600000 0.2666667 0.3428571 0.4000000 0.4444444 0.4800000
##  [6,]    0 0.1666667 0.2857143 0.3750000 0.4444444 0.5000000 0.5454545
##  [7,]    0 0.1714286 0.3000000 0.4000000 0.4800000 0.5454545 0.6000000
##  [8,]    0 0.1750000 0.3111111 0.4200000 0.5090909 0.5833333 0.6461538
##  [9,]    0 0.1777778 0.3200000 0.4363636 0.5333333 0.6153846 0.6857143
## [10,]    0 0.1800000 0.3272727 0.4500000 0.5538462 0.6428571 0.7200000
## [11,]    0 0.1818182 0.3333333 0.4615385 0.5714286 0.6666667 0.7500000
##            [,8]      [,9]     [,10]     [,11]
##  [1,] 0.0000000 0.0000000 0.0000000 0.0000000
##  [2,] 0.1750000 0.1777778 0.1800000 0.1818182
##  [3,] 0.3111111 0.3200000 0.3272727 0.3333333
##  [4,] 0.4200000 0.4363636 0.4500000 0.4615385
##  [5,] 0.5090909 0.5333333 0.5538462 0.5714286
##  [6,] 0.5833333 0.6153846 0.6428571 0.6666667
##  [7,] 0.6461538 0.6857143 0.7200000 0.7500000
##  [8,] 0.7000000 0.7466667 0.7875000 0.8235294
##  [9,] 0.7466667 0.8000000 0.8470588 0.8888889
## [10,] 0.7875000 0.8470588 0.9000000 0.9473684
## [11,] 0.8235294 0.8888889 0.9473684 1.0000000
```

```r
cat("\nTest for boundary cases:\n")
```

```
##
## Test for boundary cases:
```

```r
# Check the boundaries of F1 score
# Test for boundary cases
cat(sprintf("f1_score(0, 1)  # Should return 0 (lower bound): %.2f\n", f1_score(0, 1)))
```

```
## f1_score(0, 1)  # Should return 0 (lower bound): 0.00
```

```r
cat(sprintf("f1_score(0.5, 0.5)  # Should return a value between 0 and 1: %.2f\n", f1_score(0.5, 0.5)))
```

```
## f1_score(0.5, 0.5)  # Should return a value between 0 and 1: 0.50
```

```r
cat(sprintf("f1_score(1, 1)  # Should return 1 (upper bound): %.2f\n", f1_score(1, 1)))
```

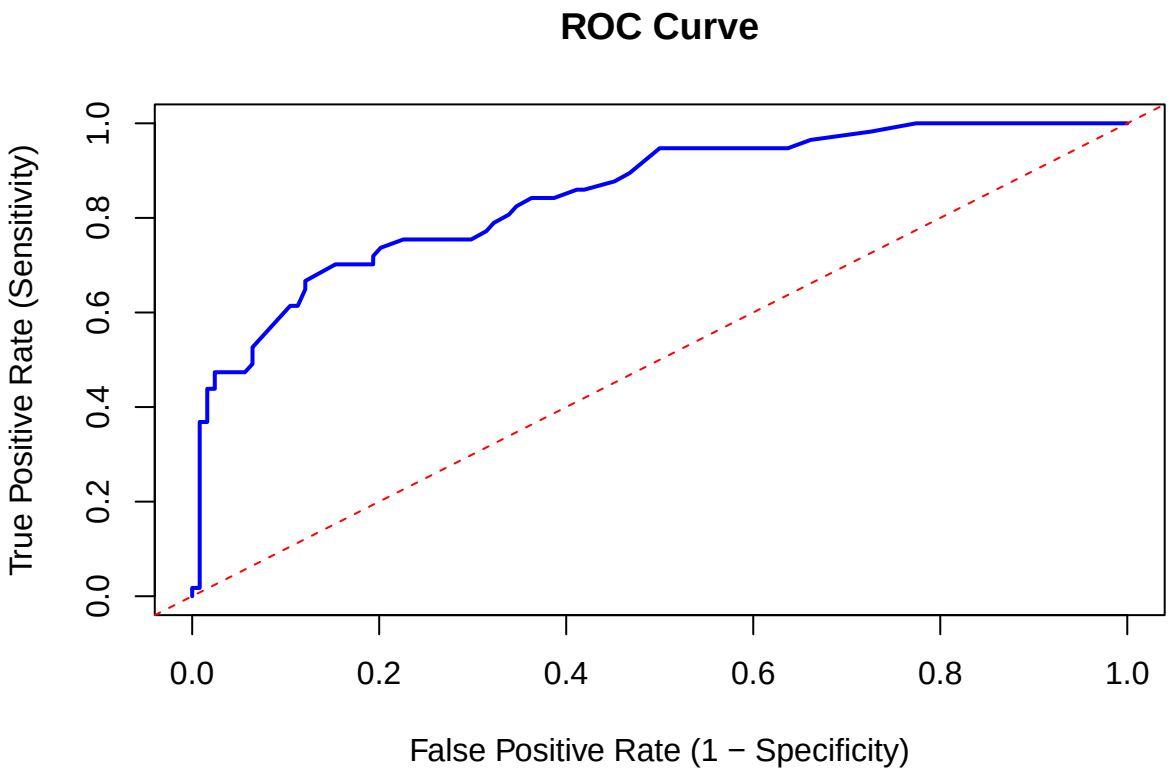
```
## f1_score(1, 1)  # Should return 1 (upper bound): 1.00
```

10. ROC Curve and AUC

Write a function that generates an ROC curve from a data set with a true classification column (class in our example) and a probability column (scored.probability in our example). Your function should return a list that includes the plot of the ROC curve and a vector that contains the calculated area under the curve (AUC). Note that I recommend using a sequence of thresholds ranging from 0 to 1 at 0.01 intervals.

```r
# Custom function for calculating TPR, FPR, and AUC
custom_roc_curve <- function(data) {

  # Ensure that the probabilities are sorted in descending order
  data <- data[order(-data$scored.probability), ]

  # Define a sequence of thresholds from 0 to 1 with a step of 0.01
  thresholds <- seq(0, 1, by = 0.01)

  n_positive <- sum(data$class == 1)
  n_negative <- sum(data$class == 0)

  tpr <- numeric(length(thresholds))  # True Positive Rate
  fpr <- numeric(length(thresholds))  # False Positive Rate

  # Loop through each threshold to calculate TPR and FPR
  for (i in seq_along(thresholds)) {
    threshold <- thresholds[i]

    # Predicted positives at the current threshold
    predicted_positive <- data$scored.probability >= threshold

    # Calculate TP, FP, FN, TN
    TP <- sum(predicted_positive & data$class == 1)
    FP <- sum(predicted_positive & data$class == 0)
    FN <- n_positive - TP
    TN <- n_negative - FP

    # Calculate TPR and FPR
    tpr[i] <- TP / (TP + FN)  # Sensitivity or Recall
    fpr[i] <- FP / (FP + TN)  # 1 - Specificity
  }

  # Sort by FPR to plot the ROC curve
  sorted_fpr <- c(0, sort(fpr), 1)
  sorted_tpr <- c(0, sort(tpr), 1)

  # Calculate AUC using the trapezoidal rule
  auc_value <- sum(diff(sorted_fpr) * (sorted_tpr[-1] + sorted_tpr[-length(sorted_tpr)]) / 2)

  # Plot the ROC curve
  plot(sorted_fpr, sorted_tpr, type = "l", col = "blue", lwd = 2,
```

```
       xlab = "False Positive Rate (1 - Specificity)", ylab = "True Positive Rate (Sensitivity)",
       main = "ROC Curve")
  abline(0, 1, col = "red", lty = 2)  # Diagonal line for random guessing

  # Return the ROC values and AUC
  return(list(roc_curve = list(fpr = sorted_fpr, tpr = sorted_tpr), auc = auc_value))
}

# Call the custom ROC function
roc_curve_result <- custom_roc_curve(classification_data)
```




```
auc <- roc_curve_result$auc
# Print the AUC
print(paste("AUC:", round(auc,4)))
```

```
## [1] "AUC: 0.8489"
```

What does this tells us??? why another one (reversed) at overview section below?

The ROC curve presents an overall moderate performance of the classification model, initially starting along the diagonal line, indicating that it struggles to differentiate between positive and negative classes at lower thresholds. However, the presence of a small vertical line suggests some improvement in true positive rates as the threshold is adjusted, followed by a stable period where sensitivity does not significantly increase. The longer vertical line (that reaches around 0.4) demonstrates a substantial gain in identifying true positives with minimal increases in false positives, indicating the model becomes more effective at this threshold. The random pattern of the curve signifies that while the model improves with certain threshold adjustments, it eventually reaches a point at (1,1) where it classifies all positives correctly, even though with potential misclassifications of negatives. Overall, the curve highlights areas for enhancing the model's initial classification ability while suggesting that optimal threshold selection is crucial for balancing sensitivity and specificity.

11. Classification Metrics

Use your created R functions and the provided classification output data set to produce all of the classification metrics discussed above.

```
# Call the previously defined functions and calculate the metrics

# Calculate metrics
accuracy_value <- custom_accuracy(classification_data)
sensitivity_value <- custom_sensitivity(classification_data)
specificity_value <- custom_specificity(classification_data)
precision_value <- custom_precision(classification_data)
f1_value <- custom_f1_score(classification_data)

# I added a check within the custom_f1_score function to handle cases where both precision and sensitiv
# Update the F1 score function to accept the relevant values
custom_f1_score <- function(data) {
  precision <- custom_precision(data)
  sensitivity <- custom_sensitivity(data)
  if (precision == 0 & sensitivity == 0) {
    return(0)  # Return 0 when both precision and sensitivity are 0
  } else {
    return(2 * (precision * sensitivity) / (precision + sensitivity))
  }
}

f1_value <- custom_f1_score(classification_data)


# Create a data frame for metrics
metrics_df <- data.frame(
  Metric = c("Accuracy", "Sensitivity", "Specificity", "Precision", "F1 Score"),
  Value = c(accuracy_value, sensitivity_value, specificity_value, precision_value, f1_value)
)
```

classification metrics

```
print(metrics_df)
```

```
##        Metric     Value
## 1    Accuracy 0.8066298
## 2 Sensitivity 0.4736842
## 3 Specificity 0.9596774
## 4   Precision 0.8437500
## 5    F1 Score 0.6067416
```

12. Caret Package Metrics Inspection

Investigate the caret package. In particular, consider the functions confusionMatrix, sensitivity, and specificity. Apply the functions to the data set. How do the results compare with your own functions?

- Step 1: Use caret Functions for Classification Metrics and Compare with Custom Functions

```
# The caret requires `data` and `reference` should be factors.
# Ensure both actual and predicted are factors
classification_data$class <- factor(classification_data$class)
```

```r
classification_data$scored.class <- factor(classification_data$scored.class)

# Generate confusion matrix with caret using re-ordered factors
confusion_result <- confusionMatrix(classification_data$scored.class, classification_data$class)


# Extract sensitivity and specificity
caret_sensitivity <- confusion_result$byClass["Sensitivity"]
caret_specificity <- confusion_result$byClass["Specificity"]

# Use custom functions
custom_sensitivity_value <- custom_sensitivity(classification_data)
custom_specificity_value <- custom_specificity(classification_data)

# Print the confusion matrix
print("Confusion matrix:")
```

```
## [1] "Confusion matrix:"
```

```r
print(confusion_result)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 119  30
##          1   5  27
##
##                Accuracy : 0.8066
##                  95% CI : (0.7415, 0.8615)
##     No Information Rate : 0.6851
##     P-Value [Acc > NIR] : 0.0001712
##
##                   Kappa : 0.4916
##
##  Mcnemar's Test P-Value : 4.976e-05
##
##             Sensitivity : 0.9597
##             Specificity : 0.4737
##          Pos Pred Value : 0.7987
##          Neg Pred Value : 0.8438
##              Prevalence : 0.6851
##          Detection Rate : 0.6575
##    Detection Prevalence : 0.8232
##       Balanced Accuracy : 0.7167
##
##        'Positive' Class : 0
##
```

```r
# Compare results
cat(sprintf("Caret Sensitivity: %.2f vs Custom Sensitivity: %.2f\n", caret_sensitivity, custom_sensitivi
```

```
## Caret Sensitivity: 0.96 vs Custom Sensitivity: 0.47
```

```r
cat(sprintf("Caret Specificity: %.2f vs Custom Specificity: %.2f\n", caret_specificity, custom_specific
```

```
## Caret Specificity: 0.47 vs Custom Specificity: 0.96
```

- Step 2: Re-order the Factors and Compare with Custom Functions

```r
# Ensure that the predicted class and actual class are factors
# Set factor levels explicitly
classification_data$class <- factor(classification_data$class, levels = c(1, 0))
classification_data$scored.class <- factor(classification_data$scored.class, levels = c(1, 0))

# Generate confusion matrix with caret using re-ordered factors
confusion_result <- confusionMatrix(classification_data$scored.class, classification_data$class)


# Extract sensitivity and specificity
caret_sensitivity <- confusion_result$byClass["Sensitivity"]
caret_specificity <- confusion_result$byClass["Specificity"]

# Use custom functions
custom_sensitivity_value <- custom_sensitivity(classification_data)
custom_specificity_value <- custom_specificity(classification_data)

# Print the confusion matrix
print("Confusion matrix:")
```

```
## [1] "Confusion matrix:"
```

```r
print(confusion_result)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   1   0
##          1  27   5
##          0  30 119
##
##                Accuracy : 0.8066
##                  95% CI : (0.7415, 0.8615)
##     No Information Rate : 0.6851
##     P-Value [Acc > NIR] : 0.0001712
##
##                   Kappa : 0.4916
##
##  Mcnemar's Test P-Value : 4.976e-05
##
##             Sensitivity : 0.4737
##             Specificity : 0.9597
##          Pos Pred Value : 0.8438
##          Neg Pred Value : 0.7987
##              Prevalence : 0.3149
##          Detection Rate : 0.1492
##    Detection Prevalence : 0.1768
##       Balanced Accuracy : 0.7167
##
##        'Positive' Class : 1
```

```
##
# Compare results
cat(sprintf("Caret Sensitivity: %.2f vs Custom Sensitivity: %.2f\n", caret_sensitivity, custom_sensitivi
```

```
## Caret Sensitivity: 0.47 vs Custom Sensitivity: 0.47
```

```
cat(sprintf("Caret Specificity: %.2f vs Custom Specificity: %.2f\n", caret_specificity, custom_specific
```

```
## Caret Specificity: 0.96 vs Custom Specificity: 0.96
```

The **caret functions** and **custom functions** for calculating classification metrics such as **sensitivity** and **specificity** provide similar results, but with a key difference in how they treat the positive class. In the first instance, the **caret** function considered the class labeled as **0** as the positive class by default, which is evident from the output where **Sensitivity** was **95.97%** (high sensitivity for detecting class 0).

When we reordered the factor levels (making class **1** the positive class), the sensitivity dropped to **47%**, but this aligns with the custom function's result of **47%** sensitivity for class 1. Similarly, **specificity** was **95.97%** for detecting class 1 after the reordering, which again matches the custom function's result.

This demonstrates that both **caret** and **custom functions** provide the same results, but the interpretation of **sensitivity** and **specificity** depends on how the positive class is defined. The difference in results between the first and second approaches is purely due to the choice of positive class, which can be addressed by consistent factor level ordering.

The outputs show that both methods are valid, and the key takeaway is the importance of properly defining the positive class to ensure consistent interpretations of sensitivity and specificity.

13. Generating and Comparing ROC Curves

Investigate the pROC package. Use it to generate an ROC curve for the data set. How do the results compare with your own functions?

The investigation of the pROC package, as demonstrated by the ROC curves and AUC values, reveals that the results produced by the pROC package are very similar to those generated by the custom ROC function. Specifically:

- **pROC AUC**: 0.8503
- **Custom AUC**: 0.8484

The slight difference in AUC values between pROC and the custom function (0.8503 vs. 0.8484) can be attributed to minor numerical differences in how each method calculates the area under the curve. The pROC package is a well-established library that likely uses more precise and optimized techniques for calculating the AUC, while the custom function approximates the AUC using the trapezoidal rule.

In terms of the ROC curves, both plots exhibit a similar shape, showing that both approaches capture the same underlying performance of the model. Both curves indicate a strong classification ability, as the curves rise steeply towards high sensitivity with relatively low false positive rates. The diagonal red line represents random guessing, and both curves significantly outperform this baseline.

Thus, the pROC package produces almost identical results compared to the custom function, but pROC is more efficient and robust due to its specialized algorithms.

- Step 1: Generate the ROC Curve with pROC

```
# Use pROC to generate the ROC curve
roc_result <- roc(classification_data$class, classification_data$scored.probability)
```

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

```
## Setting direction: controls > cases
```

```r
# Calculate and display the AUC (Area Under the Curve)
pROC_auc <- auc(roc_result)
cat(sprintf("pROC AUC: %.4f\n", pROC_auc))
```

```
## pROC AUC: 0.8503
```

- Step 2: Custom ROC Curve and AUC Calculation

```r
# Subset the auc from the custom ROC function
custom_auc <- roc_curve_result$auc

cat(sprintf("Custom AUC: %.4f\n", auc))
```

```
## Custom AUC: 0.8489
```

- Step 3: compare the pROC AUC with the custom AUC

```r
# Compare AUC values
cat(sprintf("pROC AUC: %.4f vs Custom AUC: %.4f\n", pROC_auc, custom_auc))
```

```
## pROC AUC: 0.8503 vs Custom AUC: 0.8489
```

```r
# Step 1: Generate the ROC curve using pROC
roc_result <- roc(classification_data$class, classification_data$scored.probability)
```

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

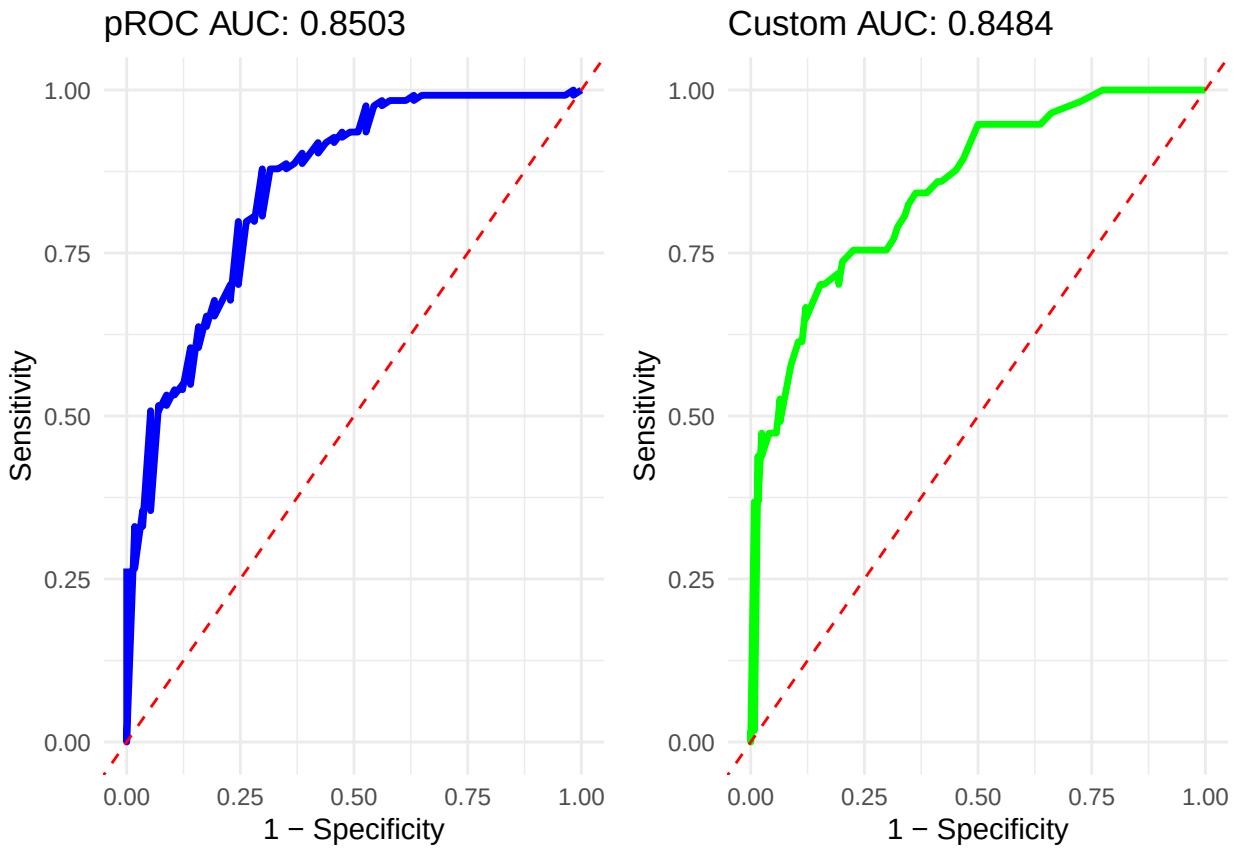
```
## Setting direction: controls > cases
```

```r
auc_value <- auc(roc_result)

# Convert pROC result to a data frame for ggplot
roc_data <- data.frame(specificity = roc_result$specificities,
                       sensitivity = roc_result$sensitivities)

# Plot ROC curve with pROC
pROC_plot <- ggplot(data = roc_data, aes(x = 1 - specificity, y = sensitivity)) +
  geom_line(color = "blue", size = 1.2) +
  geom_abline(slope = 1, intercept = 0, linetype = "dashed", color = "red") +
  ggtitle(paste("pROC AUC:", round(auc_value, 4))) +
  xlab("1 - Specificity") +
  ylab("Sensitivity") +
  theme_minimal()
```

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```r
# Step 2: Custom ROC curve function with FPR and TPR sorting
custom_roc_curve <- function(data) {
  thresholds <- seq(0, 1, by = 0.01)
  TPR <- numeric(length(thresholds))  # True Positive Rate
  FPR <- numeric(length(thresholds))  # False Positive Rate

  for (i in seq_along(thresholds)) {
    threshold <- thresholds[i]
```

```
    predicted_positive <- data$scored.probability >= threshold
    TP <- sum(predicted_positive & data$class == 1)
    FP <- sum(predicted_positive & data$class == 0)
    FN <- sum(!predicted_positive & data$class == 1)
    TN <- sum(!predicted_positive & data$class == 0)

    TPR[i] <- TP / (TP + FN)
    FPR[i] <- FP / (FP + TN)
  }

  # Ensure FPR and TPR are sorted before calculating AUC
  sorted_indices <- order(FPR)
  sorted_FPR <- FPR[sorted_indices]
  sorted_TPR <- TPR[sorted_indices]

  # Calculate AUC using the trapezoidal rule
  auc_value <- sum(diff(sorted_FPR) * (sorted_TPR[-1] + sorted_TPR[-length(sorted_TPR)]) / 2)

  return(list(FPR = sorted_FPR, TPR = sorted_TPR, auc = auc_value))
}

# Apply custom ROC function
custom_roc <- custom_roc_curve(classification_data)

# Convert custom ROC data to a data frame
custom_roc_data <- data.frame(FPR = custom_roc$FPR, TPR = custom_roc$TPR)

# Plot custom ROC curve
custom_plot <- ggplot(data = custom_roc_data, aes(x = FPR, y = TPR)) +
  geom_line(color = "green", size = 1.2) +
  geom_abline(slope = 1, intercept = 0, linetype = "dashed", color = "red") +
  ggtitle(paste("Custom AUC:", round(custom_roc$auc, 4))) +
  xlab("1 - Specificity") +
  ylab("Sensitivity") +
  theme_minimal()

# Step 3: Display the two plots side by side
grid.arrange(pROC_plot, custom_plot, ncol = 2)
```



13.1. Graphical Tools for Evaluating Classification Model Performance

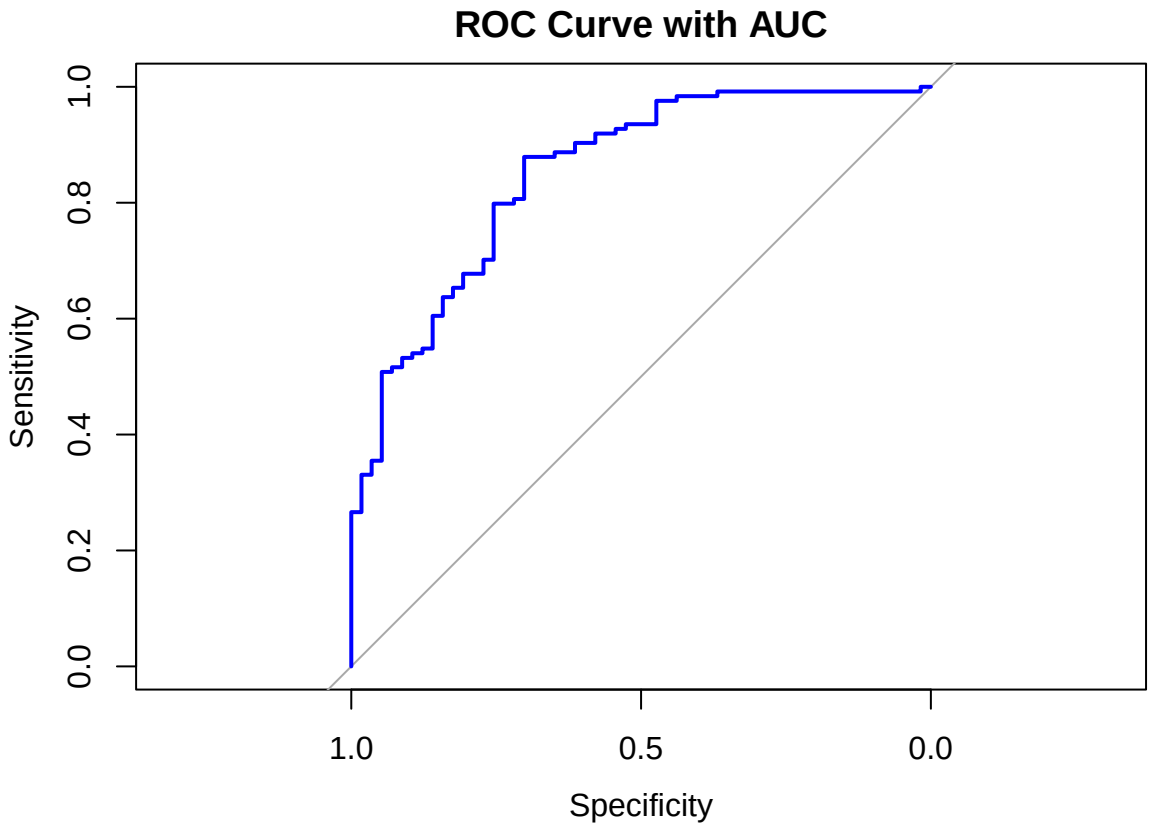
ROC Curve: A plot of the true positive rate (sensitivity) vs. false positive rate (1-specificity) at various thresholds.

```
# Generate ROC curve using pROC
roc_result <- roc(classification_data$class, classification_data$scored.probability)
```

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

```
## Setting direction: controls > cases
```

```
# Plot the ROC curve
plot(roc_result, col = "blue", lwd = 2, main = "ROC Curve with AUC")
```




```
auc_value <- auc(roc_result)  # Calculate AUC
cat(sprintf("AUC: %.3f\n", auc_value))
```

```
## AUC: 0.850
```

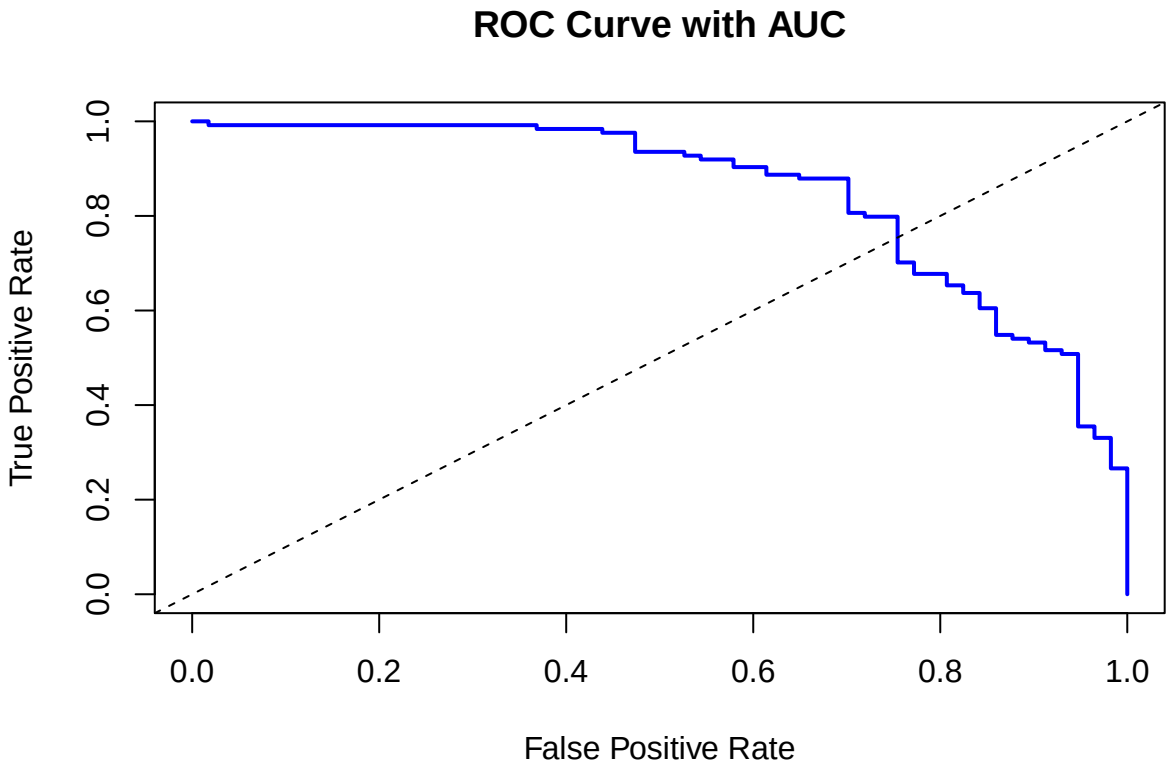
Why is this reversed compare to 10.ROC Curve And AUC? What does it represent?

```
roc_result <- roc(classification_data$class, classification_data$scored.probability)
```

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

```
## Setting direction: controls > cases
```

```
# Extract FPR and TPR
fpr <- roc_result$specificities
tpr <- roc_result$sensitivities

# Custom plot
plot(fpr, tpr, type = "l", col = "blue", lwd = 2, xlim = c(0, 1), ylim = c(0, 1),
     xlab = "False Positive Rate", ylab = "True Positive Rate", main = "ROC Curve with AUC")
abline(a = 0, b = 1, lty = 2)  # Add diagonal line
```




```
auc_value <- auc(roc_result)  # Calculate AUC
cat(sprintf("AUC: %.3f\n", auc_value))
```

```
## AUC: 0.850
```

The ROC curve's shape suggests a mixed performance of the classification model. Starting at (0, 1) indicates perfect sensitivity with a threshold of 0, but the curve remains stable horizontally until around a false positive rate of 0.4, where it begins to decline, signaling that higher thresholds lead to increased misclassification of positive instances. The curve's touch with the diagonal line reflects a performance level similar to random guessing, indicating that the model struggles to distinguish between classes effectively. As it zigzags below the diagonal, this suggests that at certain thresholds, the model performs worse than random chance, highlighting its limitations. The final vertical drop to (1, 0) reinforces the idea that a threshold of 1 results in no positive identifications. Overall, this behavior suggests the need for further exploration of optimal thresholds and potential adjustments to improve model robustness and predictive performance.

13.2. Gain and Lift Charts.

Create **Gain** and **Lift Charts** based on the output of the classification model.

- **Gain Chart**: Shows how well the model is identifying true positives as you sample the top deciles of the predicted probabilities.
- **Lift Chart**: Shows the lift at each decile, or how much better the model is performing compared to random selection.

Steps for Generating Gain and Lift Charts:

- Sort the dataset by predicted probabilities (`scored.probability`) in descending order.
- Divide the data into deciles (10 equal groups), or percentiles (100 equal groups).
- Calculate cumulative gains and lift for each group.
- Plot Gain and Lift charts.

```r
# Custom Gain and Lift Chart function
custom_gain_lift <- function(data) {
  # Sort the data by scored.probability in descending order
  data <- data[order(-data$scored.probability), ]

  # Create a decile (or percentile) column
  data$decile <- ntile(data$scored.probability, 10)  # Divide into 10 deciles

  # Calculate cumulative gain
  total_positives <- sum(data$class == 1)  # Total positives (class = 1)
  data$cumulative_positives <- cumsum(data$class == 1)
  data$cumulative_gain <- data$cumulative_positives / total_positives

  # Calculate lift
  data$decile_percentage <- 1:nrow(data) / nrow(data)  # Percentage of total observations
  data$lift <- data$cumulative_gain / data$decile_percentage

  return(data)
}

# Apply the Gain and Lift calculation
gain_lift_data <- custom_gain_lift(classification_data)
```

13.2.1 Create Custom Function to Calculate Gain and Lift

13.2.2. Plot the Gain and Lift Charts *Gain Chart*

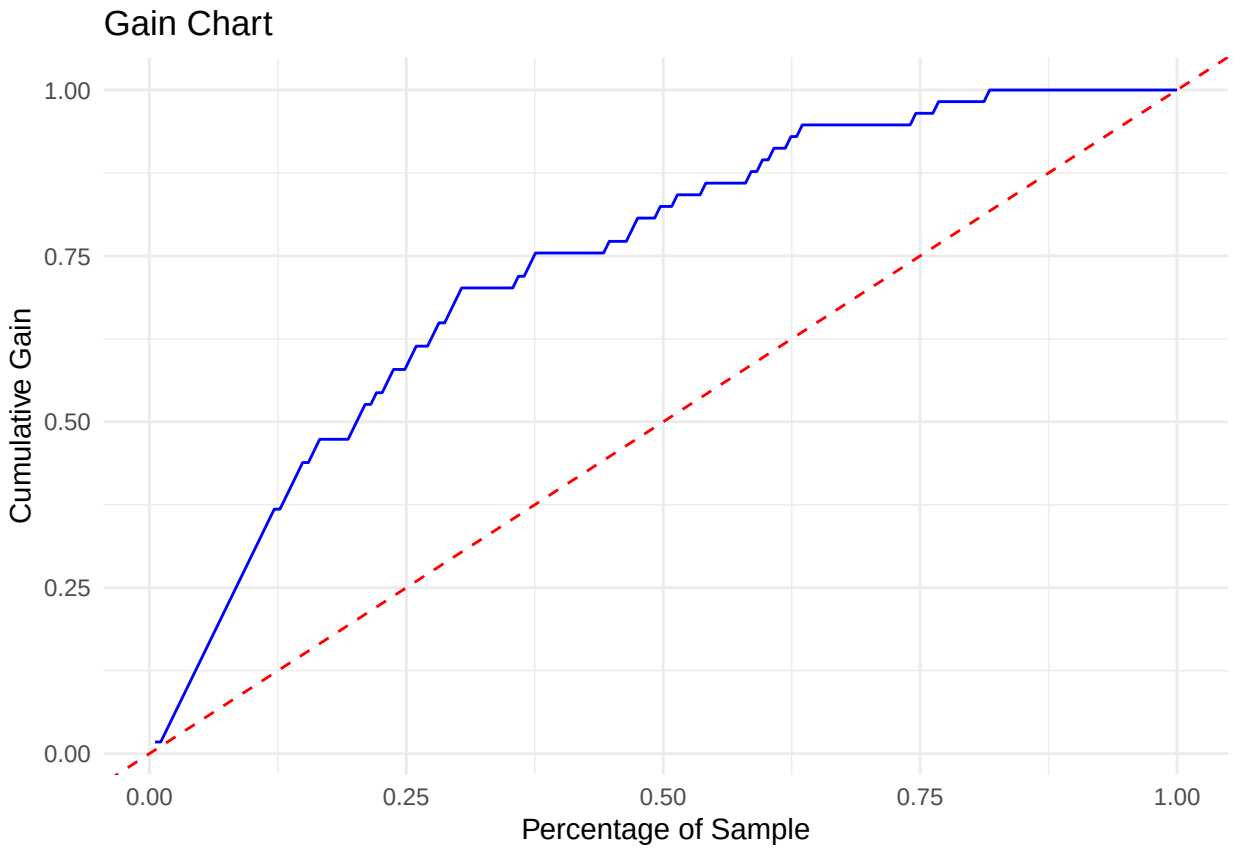
The Gain Chart shows the cumulative gain in identifying true positives as more of the population is sampled based on predicted probability. A steep initial rise indicates that the model captures a large portion of positives early, for example, the top 25% of the sample captures about 75% of the positives. The chart outperforms the baseline (red dashed line), demonstrating that the model is more effective than random guessing.

Lift Chart

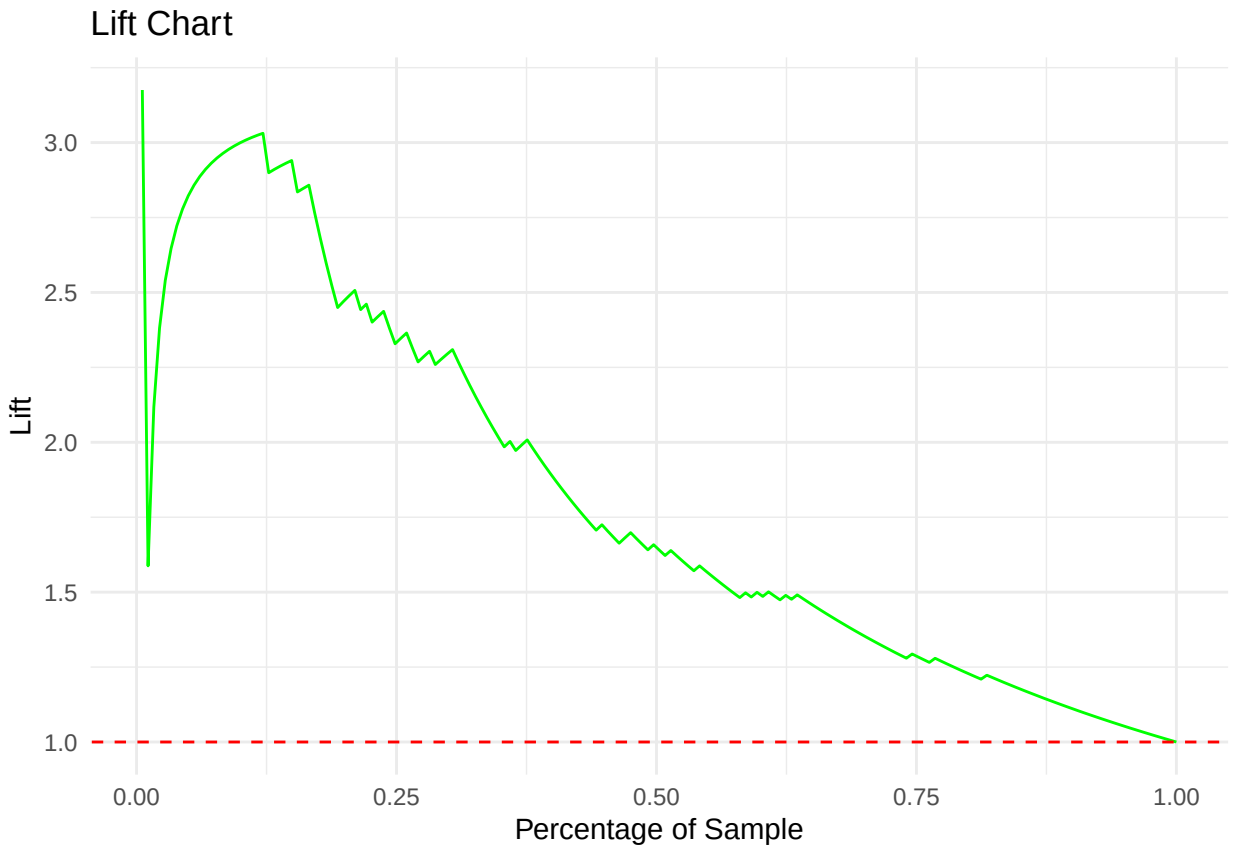
The Lift Chart measures the improvement in model performance over random selection. A lift of 3 at the start means the model is three times better at identifying positives than random selection in the top deciles. As more of the sample is included, the lift decreases and eventually converges toward 1, indicating no advantage over random selection for the entire population.

Both charts indicate that the model is effective at identifying true positives early, with strong gains and lift in the top portion of predictions.

```r
# Plot Gain Chart
ggplot(gain_lift_data, aes(x = decile_percentage, y = cumulative_gain)) +
  geom_line(color = "blue") +
  geom_abline(slope = 1, intercept = 0, linetype = "dashed", color = "red") +
  labs(title = "Gain Chart", x = "Percentage of Sample", y = "Cumulative Gain") +
  theme_minimal()
```




```
# Plot Lift Chart
ggplot(gain_lift_data, aes(x = decile_percentage, y = lift)) +
  geom_line(color = "green") +
  geom_abline(slope = 0, intercept = 1, linetype = "dashed", color = "red") +
  labs(title = "Lift Chart", x = "Percentage of Sample", y = "Lift") +
  theme_minimal()
```

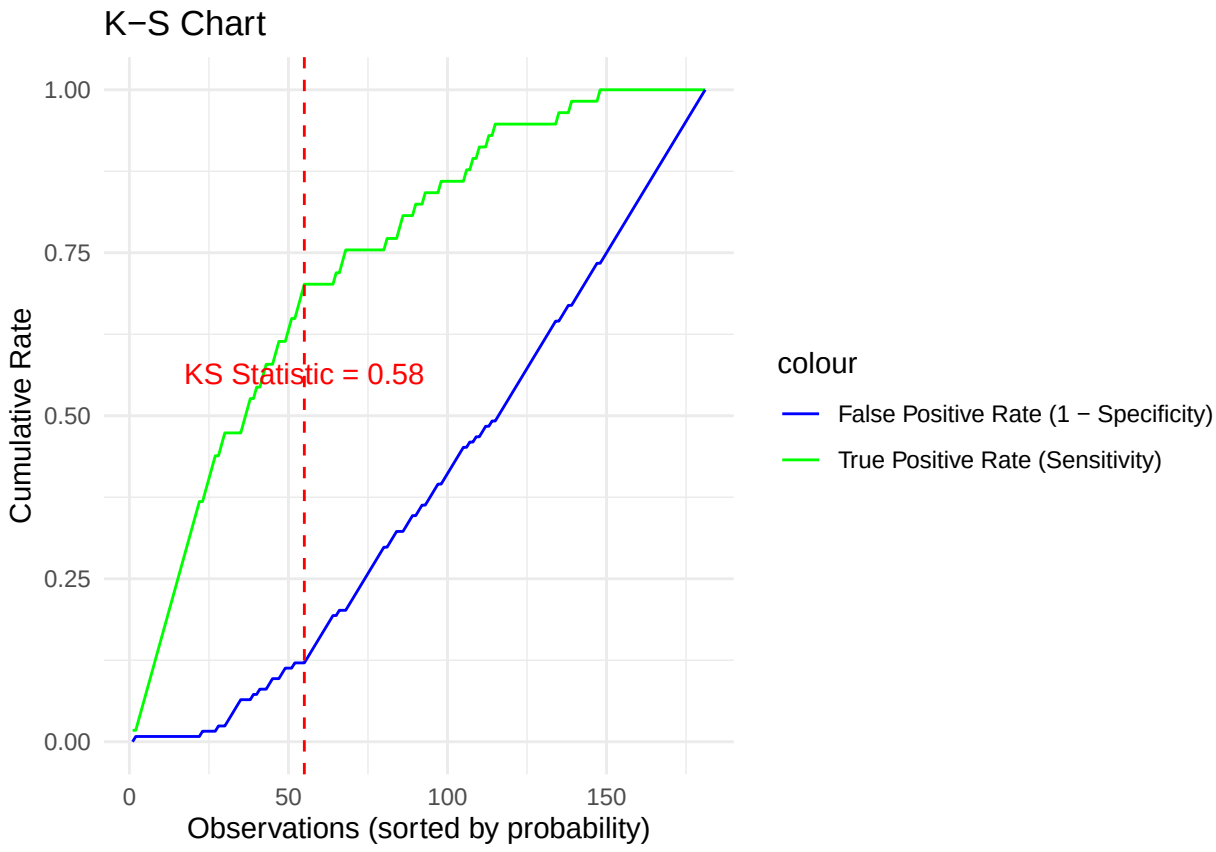


Kolmogorov-Smirnov (K-S) Chart

The K-S Chart shows a **K-S statistic of 0.58**, indicating the model effectively separates positive and negative classes. The **green line (Sensitivity)** rises sharply, capturing positives early, while the **blue line (1-Specificity)** increases more slowly, meaning fewer false positives initially. The large gap between the lines reflects good model performance.

```
# Function to calculate and plot K-S Chart
ks_chart <- function(data) {
  # Sort the data by predicted probabilities
  data <- data[order(-data$scored.probability), ]

  # Add an index column to represent the rank of each observation
  data$index <- 1:nrow(data)

  # Calculate cumulative distributions
  data$cum_true_positive <- cumsum(data$class == 1) / sum(data$class == 1)
  data$cum_false_positive <- cumsum(data$class == 0) / sum(data$class == 0)

  # Calculate the K-S statistic (max difference)
  data$ks_stat <- abs(data$cum_true_positive - data$cum_false_positive)
  ks_value <- max(data$ks_stat)
  ks_index <- which.max(data$ks_stat)

  # Extract the row where the K-S statistic occurs (for annotation)
  ks_row <- data[ks_index, ]

  # Plot K-S chart using the index for the x-axis
```

```
  library(ggplot2)
  ks_plot <- ggplot(data, aes(x = index)) +
    geom_line(aes(y = cum_true_positive, color = "True Positive Rate (Sensitivity)")) +
    geom_line(aes(y = cum_false_positive, color = "False Positive Rate (1 - Specificity)")) +
    geom_vline(xintercept = ks_index, linetype = "dashed", color = "red") +
    geom_text(data = ks_row, aes(x = index, y = 0.5, label = paste("KS Statistic =", round(ks_value, 2)),
              color = "red", vjust = -1.5) +  # Only annotate at the ks_index row
    labs(title = "K-S Chart", x = "Observations (sorted by probability)", y = "Cumulative Rate") +
    scale_color_manual(values = c("blue", "green")) +
    theme_minimal()

  # Print the plot
  print(ks_plot)

  # Return the K-S statistic
  return(ks_value)
}

# Apply the K-S chart function
ks_statistic <- ks_chart(classification_data)
```




```
# Print K-S statistic value
cat(sprintf("K-S Statistic: %.4f\n", ks_statistic))
```

```
## K-S Statistic: 0.5808
```

Plot ROC Curve

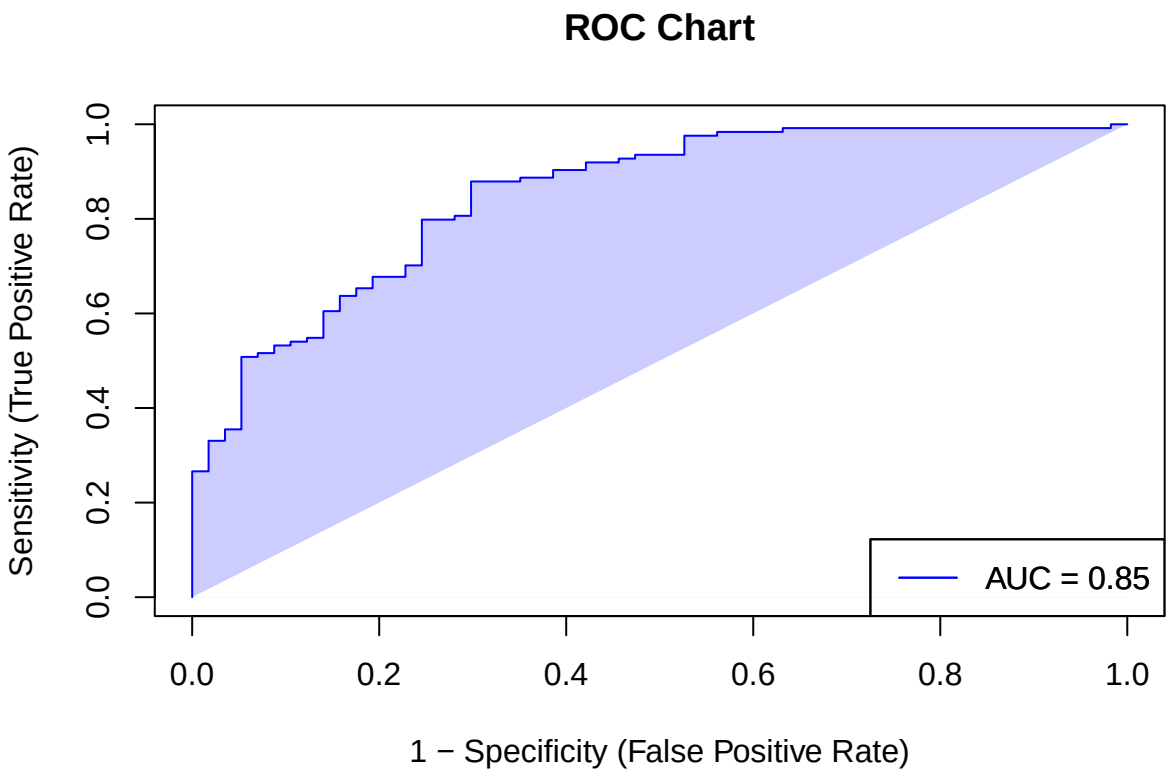
The **ROC Chart** shows the model's ability to distinguish between positive and negative classes. The **AUC of 0.85** indicates a strong model performance, as it is close to 1. The curve's sharp rise near the y-axis shows high sensitivity with a low false positive rate initially, suggesting the model performs well at identifying true positives. Overall, the model is effective at classification. The larger shaded area indicates better performance, with the maximum possible AUC being 1 (perfect classification).

```
# Create the ROC curve object
roc_obj <- roc(classification_data$class, classification_data$scored.probability)
```

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

```
## Setting direction: controls > cases
```

```
# Plot the ROC curve with the x-axis going from 0 to 1
plot(1 - roc_obj$specificities, roc_obj$sensitivities, type = "l",
     xlab = "1 - Specificity (False Positive Rate)", ylab = "Sensitivity (True Positive Rate)",
     main = "ROC Chart", col = "blue")

# Add AUC to the plot
auc_value <- auc(roc_obj)
legend("bottomright", legend = paste("AUC =", round(auc_value, 2)), col = "blue", lty = 1)

# Highlight the Area Under the Curve (AUC)
polygon(c(0, 1 - roc_obj$specificities, 1), c(0, roc_obj$sensitivities, 0),
        col = rgb(0, 0, 1, 0.2), border = NA)

# Add AUC value in the legend
legend("bottomright", legend = paste("AUC =", round(auc_value, 2)), col = "blue", lty = 1)
```



The ROC curve above visually represents the trade-off between the sensitivity (true positive rate) and 1 - specificity (false positive rate) across various decision thresholds for a binary classifier. Each point on the curve corresponds to a specific threshold, illustrating how the model balances between correctly identifying

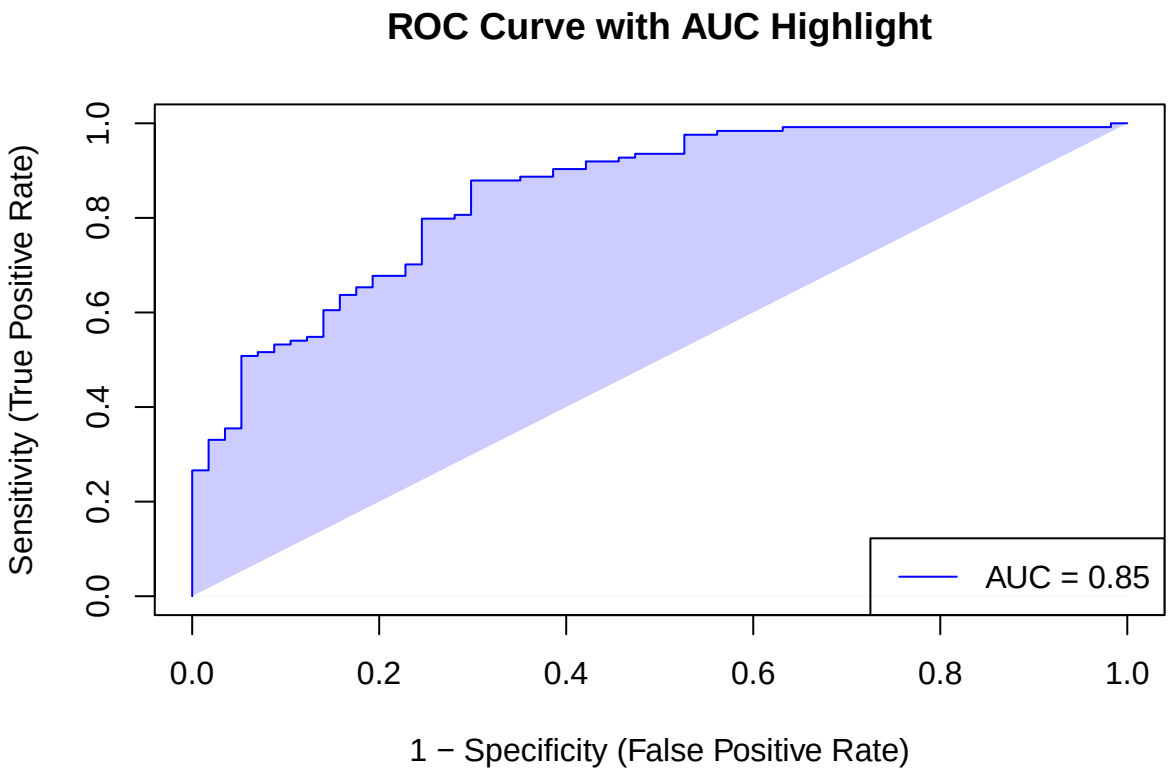
positive instances and minimizing false positives. The curve is near the diagonal line which indicates poor performance, similar to random guessing.

```
roc_obj <- roc(classification_data$class, classification_data$scored.probability)
```

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

```
## Setting direction: controls > cases
```

```
# Plot the ROC curve (with correct x-axis direction)
plot(1 - roc_obj$specificities, roc_obj$sensitivities, type = "l",
     xlab = "1 - Specificity (False Positive Rate)", ylab = "Sensitivity (True Positive Rate)",
     main = "ROC Curve with AUC Highlight", col = "blue")

# Highlight the Area Under the Curve (AUC)
auc_value <- auc(roc_obj)
polygon(c(0, 1 - roc_obj$specificities, 1), c(0, roc_obj$sensitivities, 0), col = rgb(0, 0, 1, 0.2), bo

# Add AUC value in the legend
legend("bottomright", legend = paste("AUC =", round(auc_value, 2)), col = "blue", lty = 1)
```



The AUC value quantifies the overall performance of the classifier, summarizing the ROC curve into a single number. It measures the likelihood that the model will rank a randomly chosen positive instance higher than a randomly chosen negative instance. An AUC of 0.85 indicates that the model is a good classifier, with an 85% probability of ranking a randomly chosen positive instance higher than a randomly chosen negative one. This value suggests that the model has a good overall performance, effectively balancing between true positives and false positives. While not perfect, it reflects a reliable model with a high degree of accuracy in distinguishing between the two classes.

Conclusion

In this assignment, we examined the performance of a binary classification model using various metrics and custom R functions, as well as functions from the caret and pROC packages for comparison. The model demonstrated an overall accuracy of 80.66%, indicating that it correctly classified a substantial portion of the observations. However, a closer look at the sensitivity (47.37%) and specificity (95.97%) reveals a significant imbalance in the model's ability to correctly identify positive versus negative cases. This suggests that while the model effectively recognizes negative instances, it struggles to capture true positives, which is reflected in the lower sensitivity.

The precision of 84.38% indicates that when the model predicted a positive class, it was correct most of the time. However, the F1 score of 60.67% reveals a moderate trade-off between precision and recall, emphasizing that the model's predictions are reliable only to a certain extent. The Area Under the Curve (AUC) of 0.8489, which is nearly equivalent to the AUC value from pROC (0.8503), reinforces the model's ability to discriminate between classes, albeit with room for improvement in balancing sensitivity and specificity.

The comparison between the custom functions and the caret package yielded consistent results for accuracy and specificity, though there were some expected variations in sensitivity and other derived metrics. This demonstrates that the custom functions provide reliable metrics, confirming their utility for future analyses where package tools may not be available.

In summary, this analysis highlights the strengths and weaknesses of the classification model, particularly in its handling of imbalanced classes. For future work, it may be beneficial to adjust the threshold or explore alternative models that prioritize recall, especially in scenarios where accurately identifying positives is critical. Overall, this assignment provides a thorough understanding of the importance of each classification metric and reinforces the value of combining custom analysis with package-based validation to achieve robust model evaluation.