DATA 621 Homework 2

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

2024-10-08

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:

Load the libraries

```r
# Load required libraries
library('tidyverse')
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

```r
library(ggplot2)
library(dplyr)
library(caret)
```

```
## Loading required package: lattice
##
## Attaching package: 'caret'
##
## The following object is masked from 'package:purrr':
##
##     lift
```

```r
library(pROC)
```

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

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

```r
library(gridExtra)
```

```
##
## Attaching package: 'gridExtra'
##
## The following object is masked from 'package:dplyr':
##
##     combine
```

1. Read the classification output data set.

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

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

i. Preview the dataset

```r
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
```

```r
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 ...
```

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

- **Rows** represent the **predicted classes** (scored.class), and
- **Columns** represent the **actual classes** (class).

```
# 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)

# 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, posit
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
##
```

```
# 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:
```

```
print(conf_matrix)
```

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

```
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'"
         ))
cat("\nConfusion Matrix with Values and Definitions:\n")
```

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

```
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'
```

```
# 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
```

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

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

```
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'
```

a. Understanding the Confusion Matrix Output

- **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.

b. Interpreting the Values

- **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`.

c. In Summary

- **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
```

Instead, we can do:

```
# Custom accuracy function with additional print statements for TP, TN, FP, FN
custom_accuracy <- function(data) {
  # Calculate TP, TN, FP, and FN
  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)

  # Print individual components for clarity
  cat("True Positives (TP):", TP, "\n")
  cat("True Negatives (TN):", TN, "\n")
  cat("False Positives (FP):", FP, "\n")
  cat("False Negatives (FN):", FN, "\n")

  # Calculate accuracy
  accuracy <- (TP + TN) / (TP + FP + TN + FN)
  return(accuracy)
}

# Calculate and print the accuracy
accuracy <- custom_accuracy(classification_data)
```

```
## True Positives (TP): 27
## True Negatives (TN): 119
## False Positives (FP): 5
## False Negatives (FN): 30
```

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

```
## Accuracy: 0.8066
```

```
# Verify with caret's confusionMatrix for comparison
caret_conf_matrix <- confusionMatrix(classification_data$scored.class, classification_data$class, posit
cat("\nCaret Accuracy:", caret_conf_matrix$overall['Accuracy'], "\n")
```

```
##
## Caret Accuracy: 0.8066298
```

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
```

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

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

INSTEAD, we may do

```r
# Classification Error Rate function with additional print statements for FP and FN
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)

  # Print individual components for clarity
  cat("False Positives (FP):", FP, "\n")
  cat("False Negatives (FN):", FN, "\n")

  # Calculate error rate
  error_rate <- (FP + FN) / (TP + FP + TN + FN)
  return(error_rate)
}

# Calculate and print the classification error rate
error_rate <- error_rate(classification_data)
```

```
## False Positives (FP): 5
## False Negatives (FN): 30
```

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

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

```r
# Compare the sum of accuracy and error rate
cat(sprintf("Sum of Accuracy and Classification Error Rate: %.2f\n", accuracy + error_rate))
```

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

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.

```r
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
```

INSTEAD, we can:

```r
# Precision Function
custom_precision <- function(data) {
  TP <- sum(data$class == 1 & data$scored.class == 1)
  FP <- sum(data$class == 0 & data$scored.class == 1)
  precision <- TP / (TP + FP)

  # Print statement for clarity
  cat(sprintf("Precision: %.4f - Proportion of positive predictions that are correct.\n", precision))

  return(precision)
}
precision <- custom_precision(classification_data)
```

```
## Precision: 0.8438 - Proportion of positive predictions that are correct.
```

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

```
## Precision: 0.8438
```

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

```
## precision (recall): 0.4737
```

INSTEAD,

```
# Sensitivity (Recall) Function
custom_sensitivity <- function(data) {
  TP <- sum(data$class == 1 & data$scored.class == 1)
  FN <- sum(data$class == 1 & data$scored.class == 0)
  sensitivity <- TP / (TP + FN)

  # Print statement for clarity
  cat(sprintf("Sensitivity (Recall): %.4f - Proportion of actual positives that are correctly identifie

  return(sensitivity)
}

sensitivity <- custom_sensitivity(classification_data)
```

```
## Sensitivity (Recall): 0.4737 - Proportion of actual positives that are correctly identified.
```

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

```
## Sensitivity (Recall): 0.4737
```

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.

```r
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))
}

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
```

INSTEAD of passing the precision and sensitivty to the function as parameters, we can have them to be calculated wihtin the function:

```r
# F1 Score Function
custom_f1_score <- function(data) {
  TP <- sum(data$class == 1 & data$scored.class == 1)
  FP <- sum(data$class == 0 & data$scored.class == 1)
  FN <- sum(data$class == 1 & data$scored.class == 0)

  # Calculate precision and sensitivity
  precision <- TP / (TP + FP)
  sensitivity <- TP / (TP + FN)

  # Calculate F1 Score
  if (precision + sensitivity == 0) {
    return(0)  # Avoid division by zero
  }

  f1score <- 2 * (precision * sensitivity) / (precision + sensitivity)

  return(f1score)
}
```

```r
f1score <- custom_f1_score(classification_data)
cat(sprintf("F1 Score: %.4f\n", f1score))
```

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

9 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 < < 1 and 0 < < 1 then < .)

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)
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
```

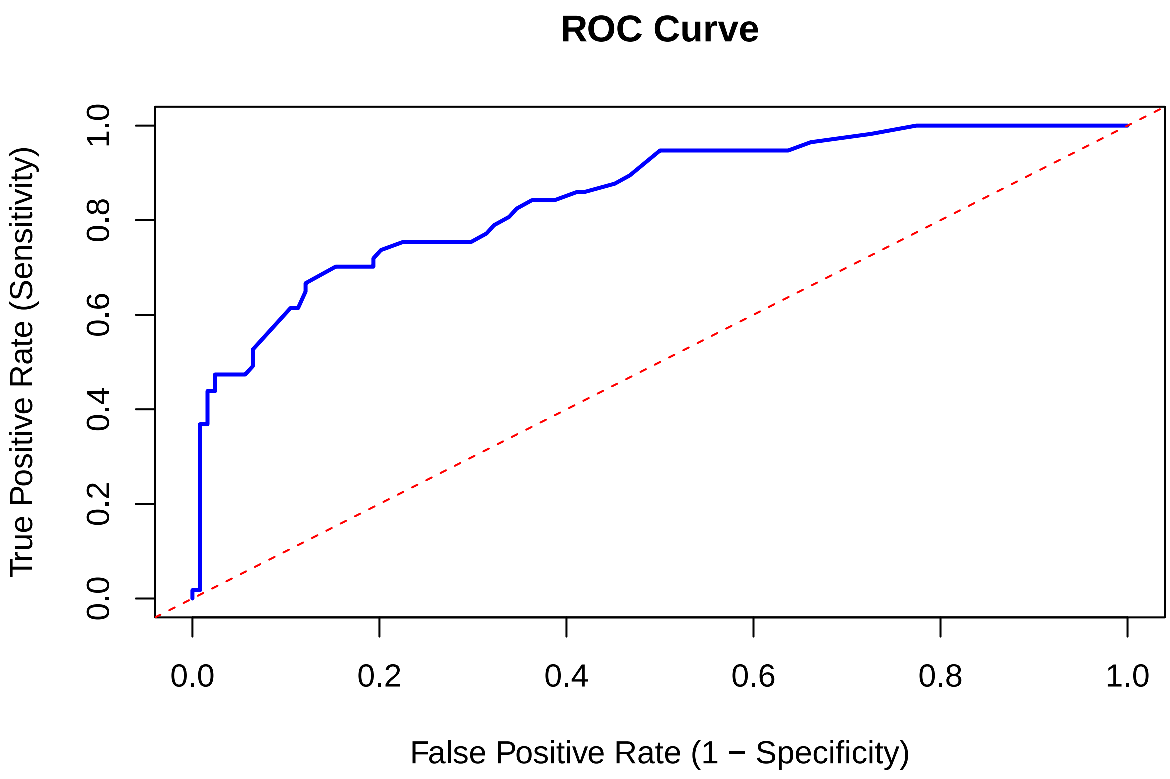
```r
cat(sprintf("f1_score(1, 1)  # Should return 1 (upper bound): %.2f\n", f1_score(1, 1)))
```

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

10. ROC Curve and AUC

10. 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"
```

11. 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)
```

```
## True Positives (TP): 27
## True Negatives (TN): 119
## False Positives (FP): 5
## False Negatives (FN): 30
```

```
sensitivity_value <- custom_sensitivity(classification_data)
```

```
## Sensitivity (Recall): 0.4737 - Proportion of actual positives that are correctly identified.
```

```
specificity_value <- custom_specificity(classification_data)
precision_value <- custom_precision(classification_data)
```

```
## Precision: 0.8438 - Proportion of positive predictions that are correct.
```

```
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
```

```r
# 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)
```

```
## Precision: 0.8438 - Proportion of positive predictions that are correct.
## Sensitivity (Recall): 0.4737 - Proportion of actual positives that are correctly identified.
```

```r
# 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)
)

# Print the metrics data frame
print("classification metrics:")
```

```
## [1] "classification metrics:"
```

```r
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. 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 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.

In conclusion, 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.

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

```
## Sensitivity (Recall): 0.4737 - Proportion of actual positives that are correctly identified.
```

```r
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_specifi
```

```
## 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)
```

```
## Sensitivity (Recall): 0.4737 - Proportion of actual positives that are correctly identified.
```

```r
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_specifici
```

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

13. 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
```

```
# 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
```

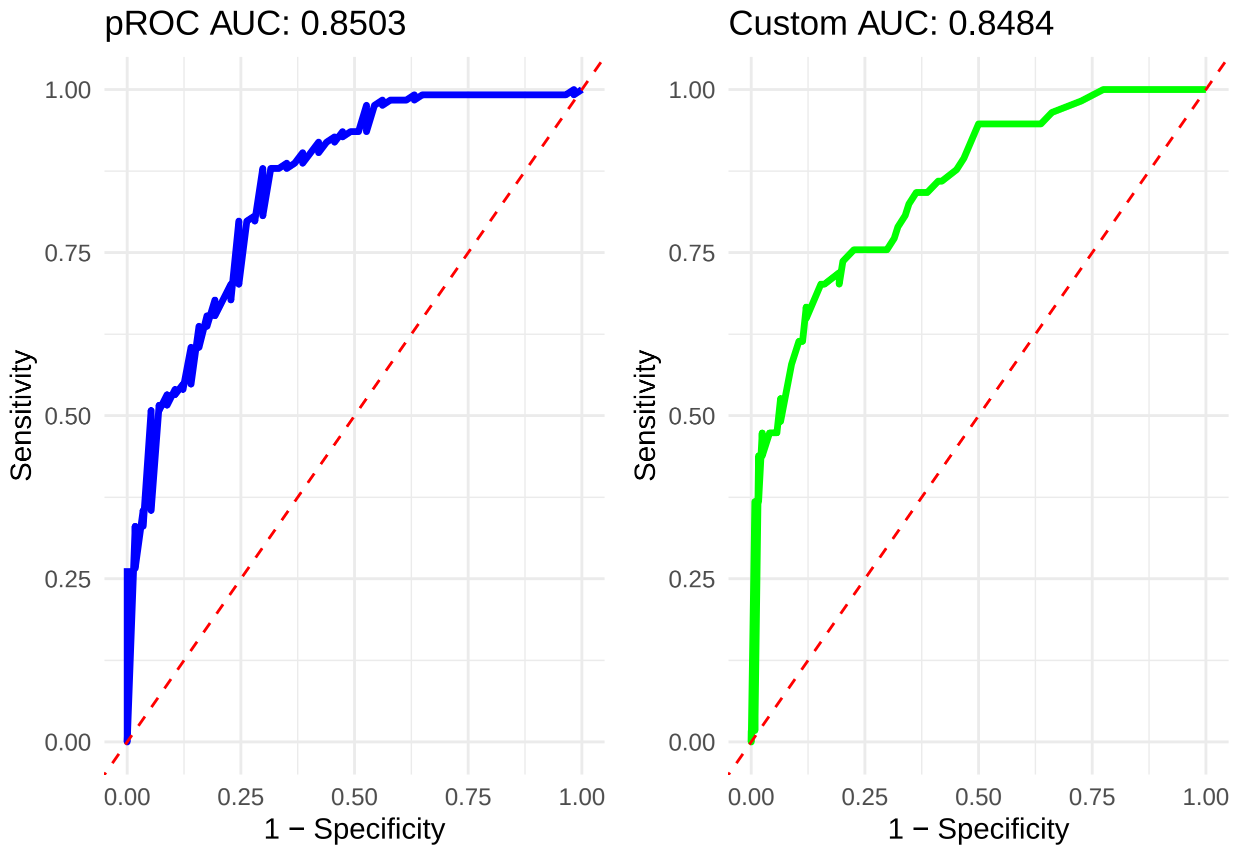
```r
# Step 1: Generate the ROC curve using pROC
roc_result <- roc(classification_data$class, classification_data$scored.probability)
```

```
## 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)
```



Overview: Plotting graphical outputs that help evaluate the performance of classification models (such as binary logistic regression)

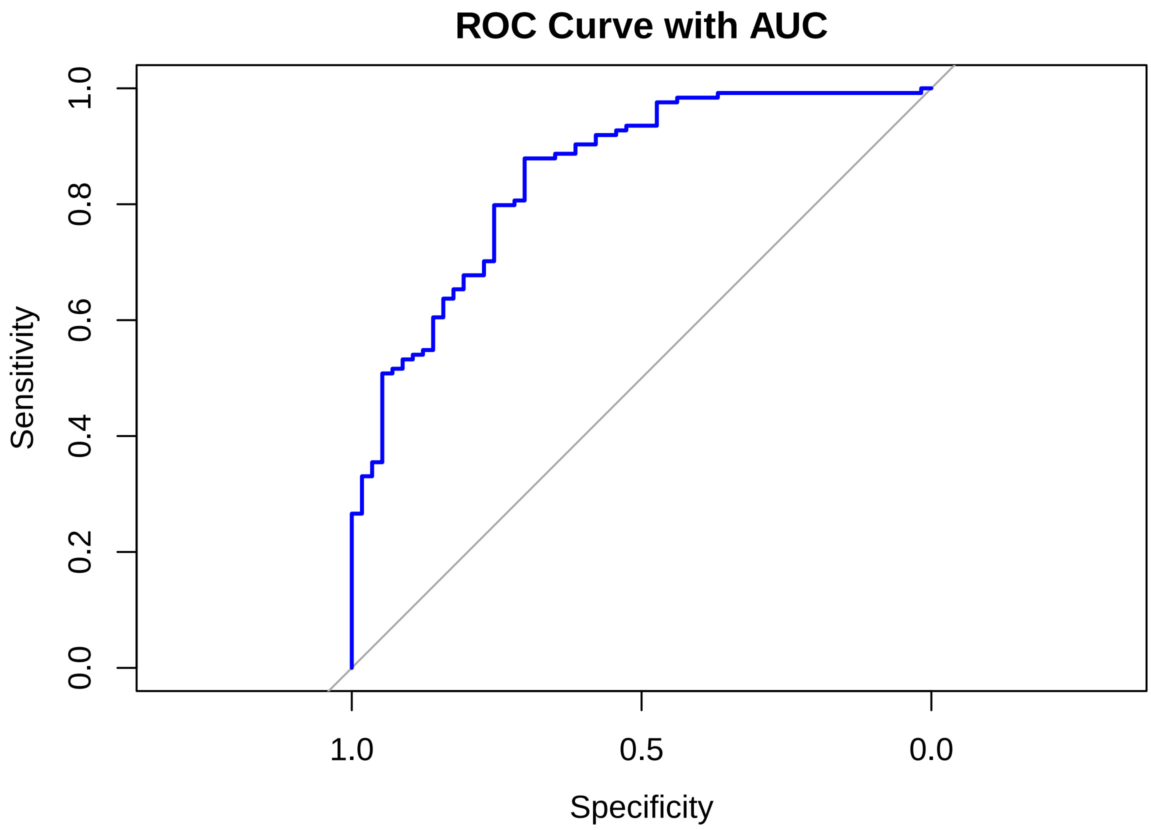
a. ROC Curve: A plot of the true positive rate (sensitivity) vs. false positive rate (1-specificity) at various thresholds.

```r
# 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
```

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



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

```
## AUC: 0.850
```

b. 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)
```

i. Create Custom Function to Calculate Gain and Lift

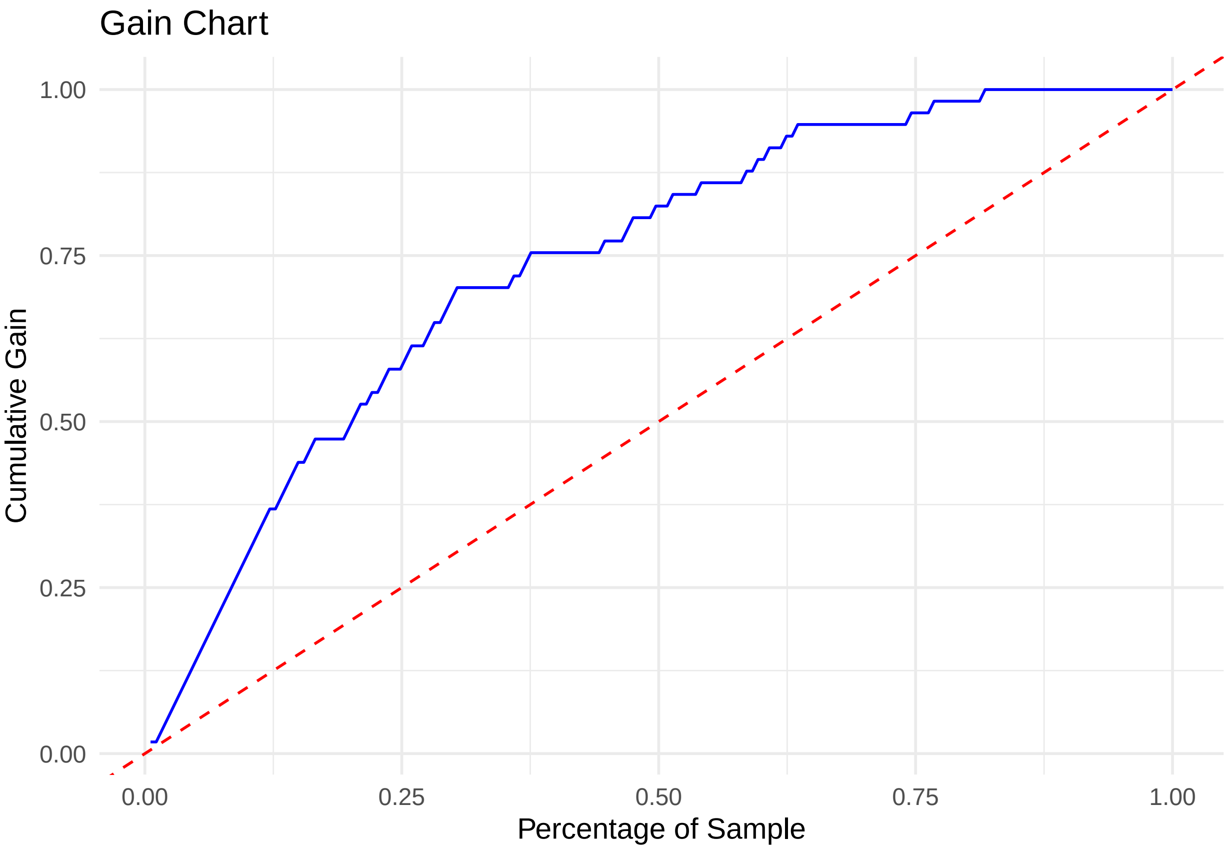
ii. Plot the Gain and Lift Charts

a. 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.

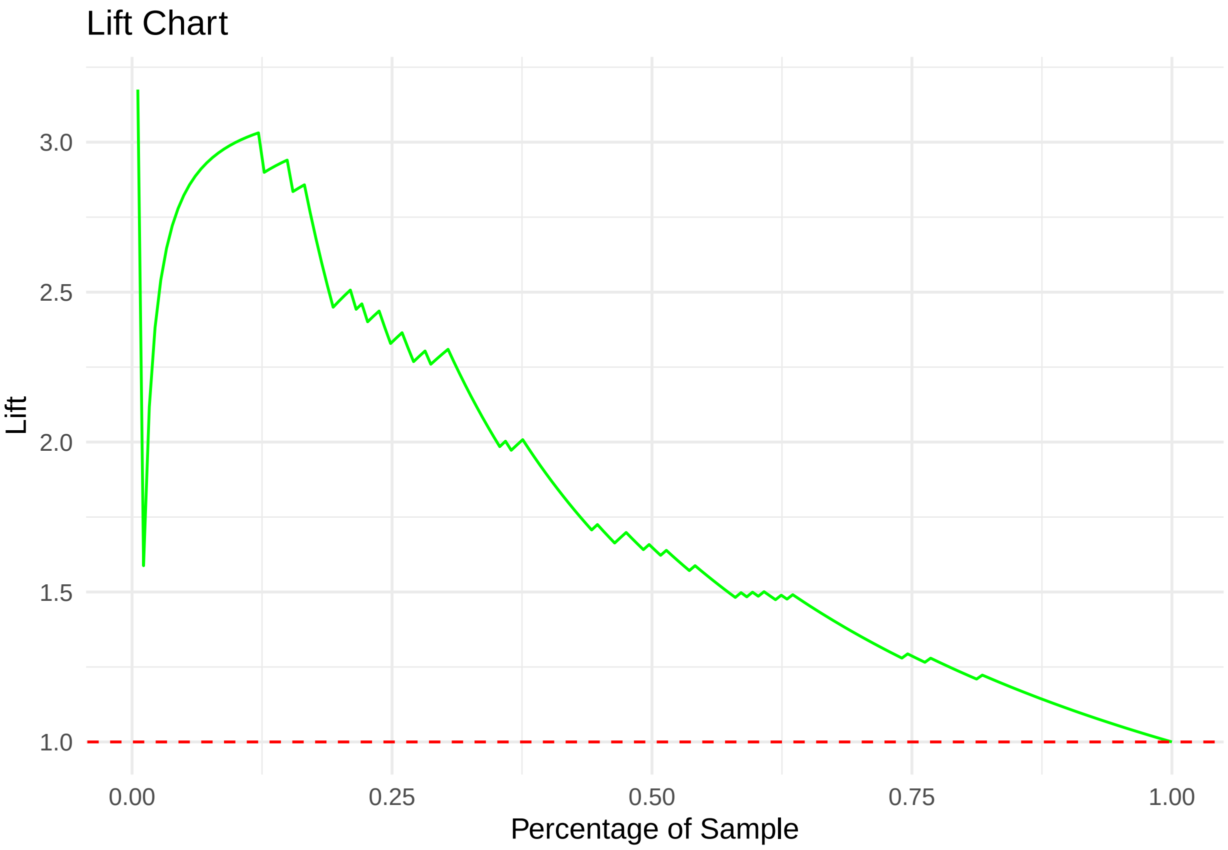
b. Lift Chart: The Lift Chart measures the improvement in model performance over random selection. A lift of 3 at the start means the model is three times better at identifying positives than random selection in the top deciles. As more of the sample is included, the lift decreases and eventually converges 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
# Load ggplot2 for plotting
library(ggplot2)

# 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()
```

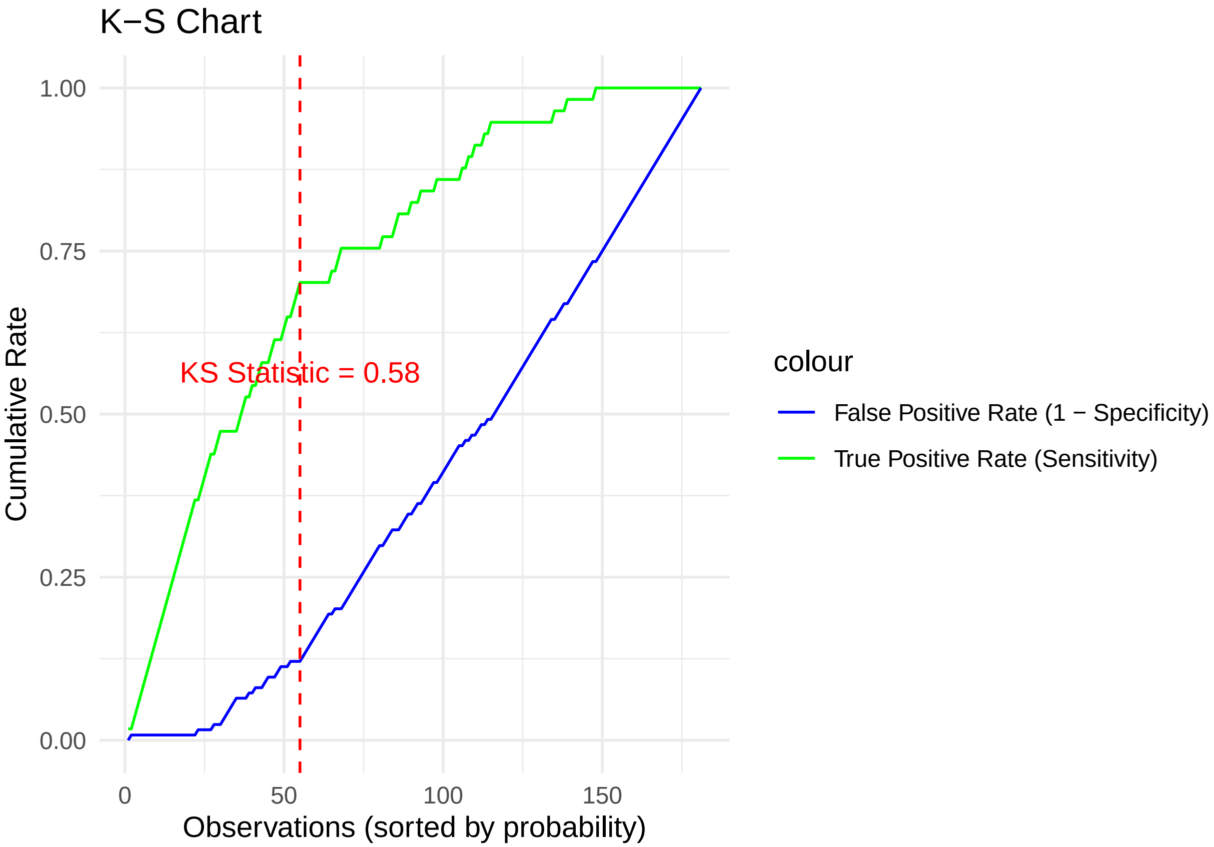


```r
# Plot Lift Chart
ggplot(gain_lift_data, aes(x = decile_percentage, y = lift)) +
  geom_line(color = "green") +
  geom_abline(slope = 0, intercept = 1, linetype = "dashed", color = "red") +
  labs(title = "Lift Chart", x = "Percentage of Sample", y = "Lift") +
  theme_minimal()
```



iii. 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
```

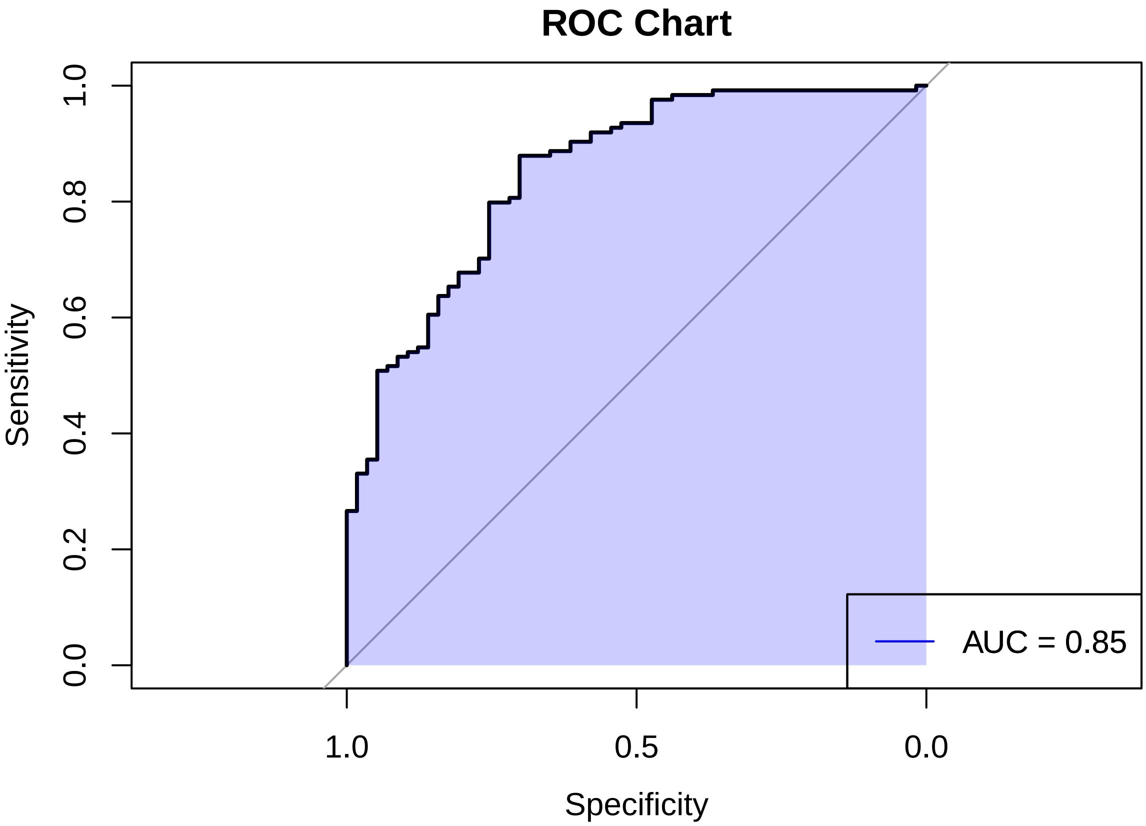
iv. Plot ROC Curve The **ROC Chart** shows the model's ability to distinguish between positive and negative classes. The **AUC of 0.85** indicates a strong model performance, as it is close to 1. The curve's sharp rise near the y-axis shows high sensitivity with a low false positive rate initially, suggesting 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).

```
# Assuming you have classification data with 'scored.probability' and 'class' variables
# 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
```

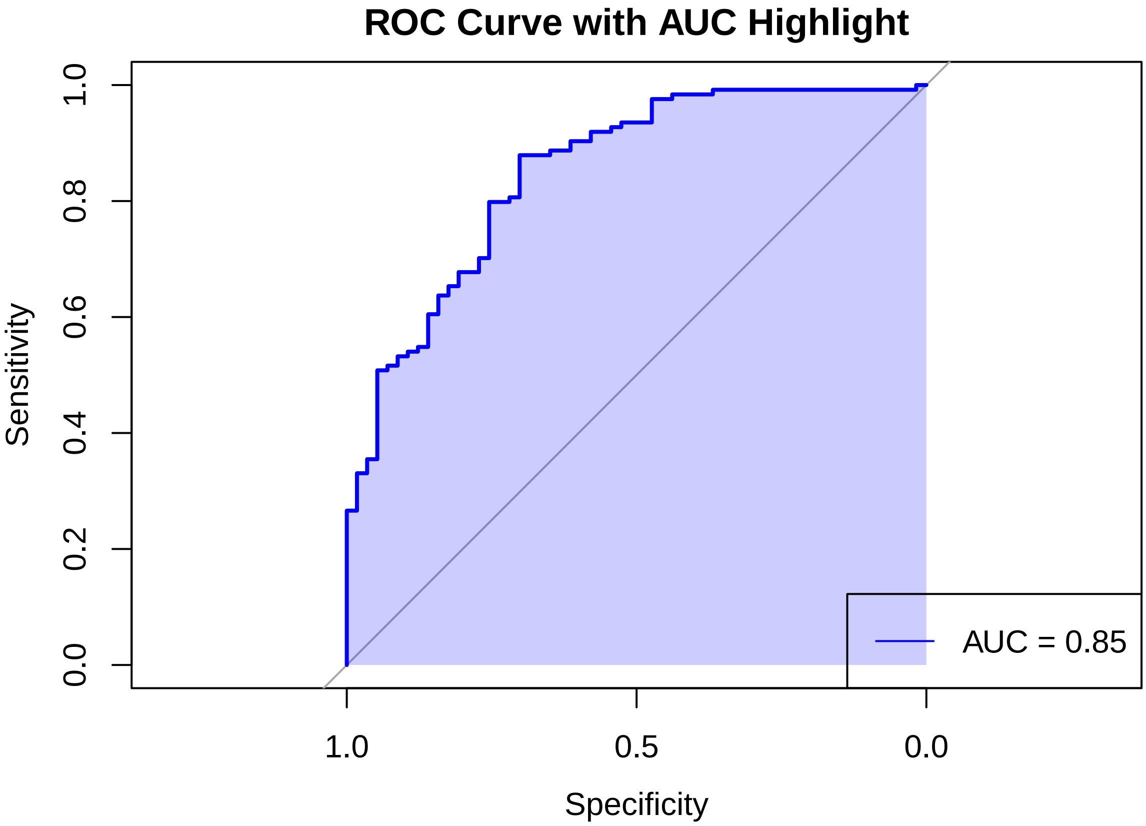
```
# Plot the ROC curve
plot(roc_obj, main = "ROC Chart")

# 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)
auc_value <- auc(roc_obj)
polygon(c(0, roc_obj$specificities, 1), c(0, roc_obj$sensitivities, 0), col = rgb(0, 0, 1, 0.2), border

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



```
# Assuming classification_data with 'scored.probability' and 'class'
roc_obj <- roc(classification_data$class, classification_data$scored.probability)
```

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

```r
# Plot the ROC curve
plot(roc_obj, main = "ROC Curve with AUC Highlight", col = "blue")

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

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



Conclusion:

In conclusion, this assignment underscores the importance of clarity and justification in the model-building process. By providing a comprehensive overview of data preparation, model selection, and evaluation methodologies, the analysis aims to equip readers with the knowledge necessary to navigate the complexities of predictive modeling.