Lab 2 Report

Group A11 - Christian Kammerer, Victor Guillo, Jakob Lindner

2025-01-24

Statement of Contribution

Assignment 1 was mainly contributed by Jakob Lindner. Christian Kammerer worked on Assignments 2, Victor Guillo on Assignment 3 and 4. Every student was responsible for the respective parts of the lab report.

Assignment 1

1.

The underlying linear model can be described with the following formula:

$$\text{Fat} = \beta_0 + \sum_{i=1}^{100} (\beta_i \text{Channel}_i) + \epsilon$$

The model is fit using the lm-method in R:

```
model_lr <- lm(Fat~.- Protein - Moisture - Sample, data=train)
```

A function is defined to calculate the errors. It take the true and predicted values as inputs and returns the mean sqaured error:

```
mean_squared_error <- function(true, predicted){
  error <- mean((true-predicted)^2)
  return(error)
}
```

The train error is very low, but the test error is very high. This strongly indicates that the model is highly overfitted and the quality of the model is bad.

```
## MSE on train data: 0.005709117
```

```
## MSE on test data: 722.4294
```

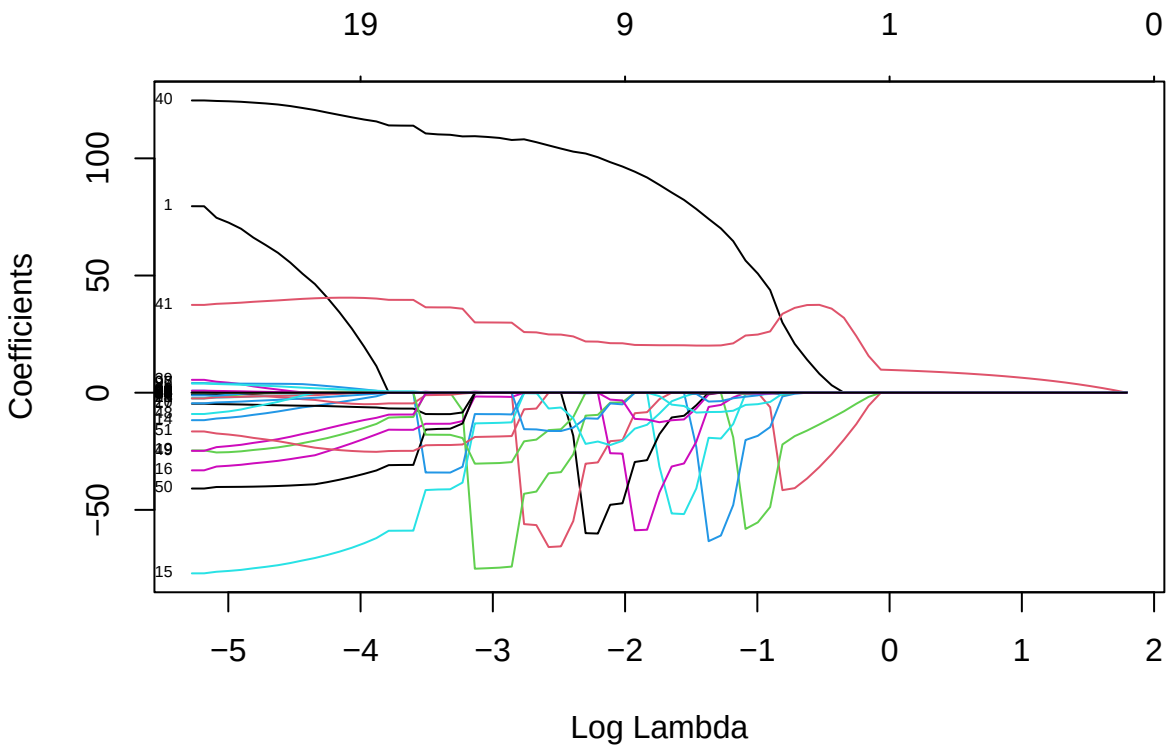
2.

The cost function to optimize for the LASSO regression looks like this:

$$\mathcal{L}(\beta) = \frac{1}{n} \sum_{i=1}^{n} \left(\text{Fat}_i - \beta_0 - \sum_{j=1}^{100} \beta_j \text{Channel}_{ij} \right)^2 + \lambda \sum_{j=1}^{100} |\beta_j|$$

3.

```
features <- train[,2:101]
target <- train$Fat
model_lasso <- glmnet(as.matrix(features), target, alpha=1, family="gaussian")
plot(model_lasso, xvar="lambda", label=TRUE)
```



The plot shows how the coefficients of the different channels get zero for higher lambda values. It shows that many of the 100 features do not have a high influence even for low lambda and go to zero quiet early (around log lambda = -4). After that, around 10 features are included, the respective value of the coefficients may increase and decrease until log lambda =-1, where only 5 features are left, with channel 41 being the feature that remains last.

```
# Find the lambda value that results in exactly three non-zero coefficients

num_features <- apply(coef(model_lasso) != 0, 2, sum) - 1  # Exclude the intercept
lambda_values <- model_lasso$lambda
selected_lambdas <- lambda_values[which(num_features == 3)]
```

The following three values for lambda can be chosen for a model with only three features:

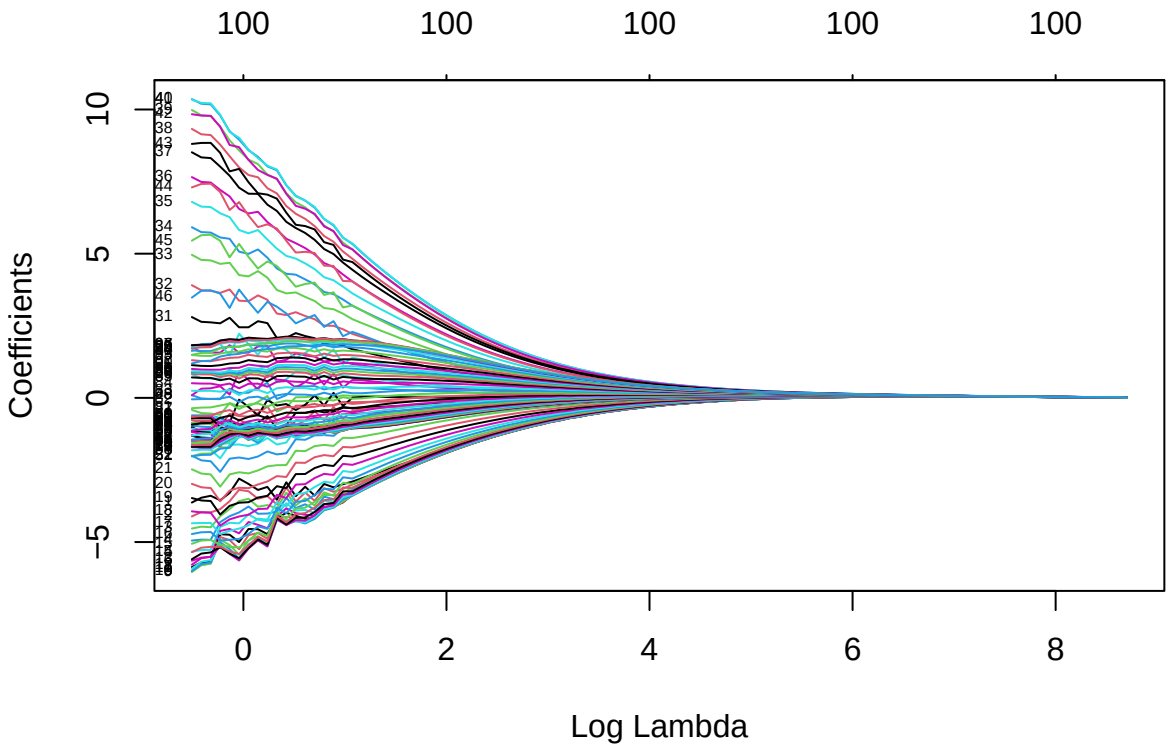
```
## Possible lambdas for exactly 3 features: 0.8530452 0.777263 0.7082131
```

```
## Corresponding log(lambda) values: -0.1589428 -0.2519765 -0.3450102
```

The log lambda values correspond with the visualization in the plot. The black line gets to zero at around -0.33 and the green line close to 0. This is the intervall, where exactly three coefficients are not zero.

4.

```
model_ridge <- glmnet(as.matrix(features), target, alpha=0, family="gaussian")
plot(model_ridge, xvar="lambda", label=TRUE)
```




```
num_features <- apply(coef(model_ridge) != 0, 2, sum) - 1

lambda_values <- model_ridge$lambda
selected_lambdas <- lambda_values[which(num_features == 3)]
```

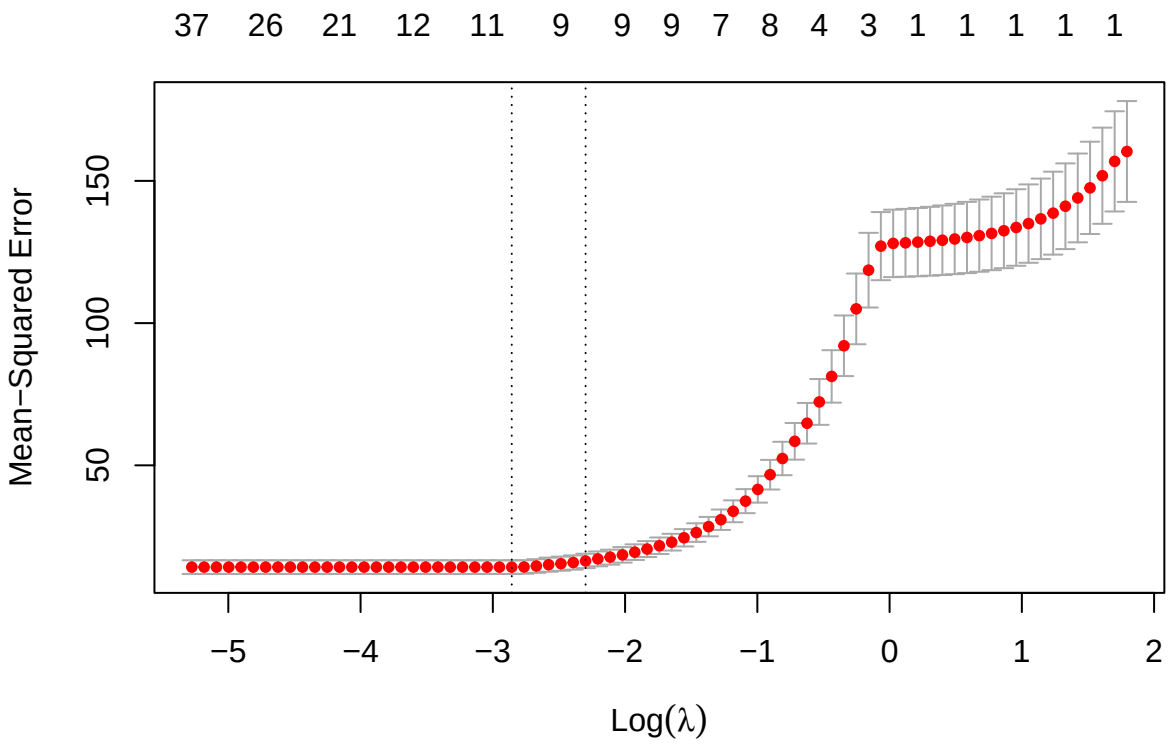
The plot for ridge regression draws a different picture than the one for LASSO. Here all coefficients get smaller for higher lambda, but none ever gets zero, as the following output shows ("Number of features for every lambda value is 100). Therefore the LASSO regression simplifies the model, while Ridge regression only shrinks the coefficients.

```
##
## Possible lambdas for exactly 3 features:

## Number of features for lambda values:  100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 10
```

5.

Cv.glmnet is used to perform cross validation on LASSO:

```
model_cv <- cv.glmnet(as.matrix(features), target, alpha=1, family="gaussian")
plot(model_cv)
```



The plot shows, that the MSE is constantly very low for small lambda until -2.8, then increases exponentially until almost 0 and then increases slower again.

The optimal lambda is stored in the model_cv objected and can be retrieved with $lambda.min. For this lambda 8 features are used. The optimal lambda does not seem to produce a significantly better result then for lambda -4, as the graph is constant for values smaller than the optimal lambda.

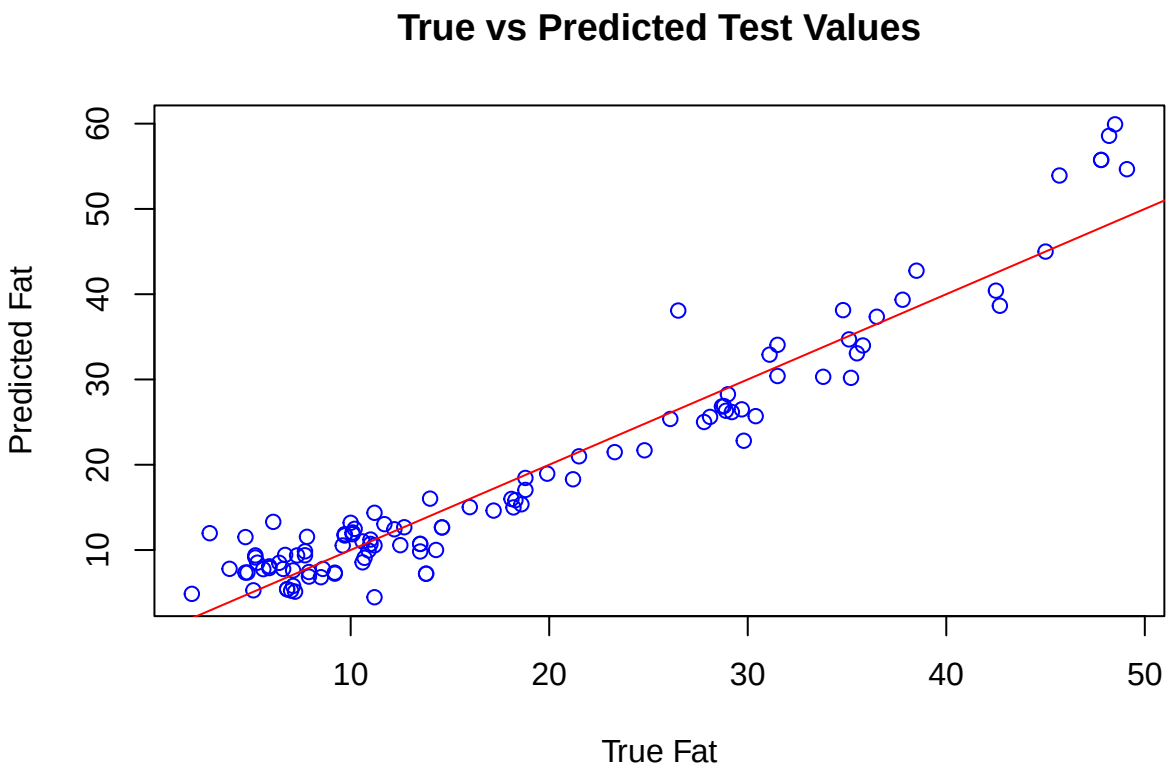
```
optimal_lambda <- model_cv$lambda.min
optimal_coefficients <- coef(model_cv, s = "lambda.min")
num_nonzero <- sum(optimal_coefficients != 0) - 1
```

```
## Optimal lambda: 0.05744535
```

```
## Number of selected features: 8
```

In the following the predictions and true values of the test data are plotted. The red line indicates the values where the prediction would be the same as the actual value.

```
y_test_pred <- predict(model_cv,newx = as.matrix(test[,2:101]), s="lambda.min")

plot(test$Fat, y_test_pred, main = "True vs Predicted Test Values",
     xlab = "True Fat", ylab = "Predicted Fat", col = "blue")
abline(0, 1, col = "red") # plot the line of y = x
```



Most points are very close to the red line, so the prediction quality is good. It is a big improvement compared to the linear regression in the first task, as the test error is significantly lower:

```
## Test error for LASSO: 13.67339
```

```
knitr::opts_chunk$set(echo = TRUE)
# Load necessary libraries
library(tree)
library(ggplot2)
library(dplyr)
library(rpart)

# Load data
data <- read.csv('bank-full.csv', sep = ";")

# Remove duration column
data <- data[, !names(data) %in% c("duration")]

# Convert all "chr" cols to factors
cols_to_convert <- c("job", "marital", "education", "default", "housing", "loan",
                     "contact", "month", "poutcome", "y") # Specify columns to convert
data[cols_to_convert] <- lapply(data[cols_to_convert], factor)

# Split the data as shown in the lecture
# Train, Test, Validation Split
n=dim(data)[1]
```

```r
set.seed(12345)
id=sample(1:n, floor(n*0.4))
train=data[id,]
id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.3))
valid=data[id2,]
id3=setdiff(id1,id2)
test=data[id3,]
```

Assignment 2

Experiment regarding over- and underfitting

Task statement: Fit decision trees to the training data so that you change the default settings one by one (i.e. not simultaneously): a. Decision Tree with default settings. b. Decision Tree with smallest allowed node size equal to 7000. c. Decision trees minimum deviance to 0.0005. and report the misclassification rates for the training and validation data. Which model is the best one among these three? Report how changing the deviance and node size affected the size of the trees and explain why.

Table 1: Results from Experiment 1

Model	Missclassification.Error.Training	Missclassification.Error.Validation	Tree_Size (Leaf Count)
Default	0.1048	0.1093	6
Min Node Size = 7000	0.1142	0.1208	2
Min Deviance = 0.0005	0.0940	0.1119	122

As we can observe in the table above, the model which performs best on the validation data set, is the one that is being trained using the default model parameters. This is because the two other models, are either underfitting, or overfitting the training data.

Why is model 2 underfitting? Model 2 requires at least 7000 observations per node. As the total training data set size spans 18084 observations, this prevents the model from performing more than one split. As it is pretty much impossible to split the data only once, such that the minority class, which only accounts for roughly 11.4% becomes the majority class in one leaf node. This causes the model to output the label of the majority class "no", irregardless of what split is performed. This causes the model to have a consistent missclassification error of 11.4%, which again is equal to the percentage of minority samples.

Why is model 3 overfitting? The fact that model 3 performs significantly better on the training data, than on the validation data is indicative of overfitting. This is caused by us lowering the minimum deviance parameter from the default value of 0.01 to 0.0005. This causes the tree to branch out to an extreme degree, which is shown by the 122 leaf nodes that are present.

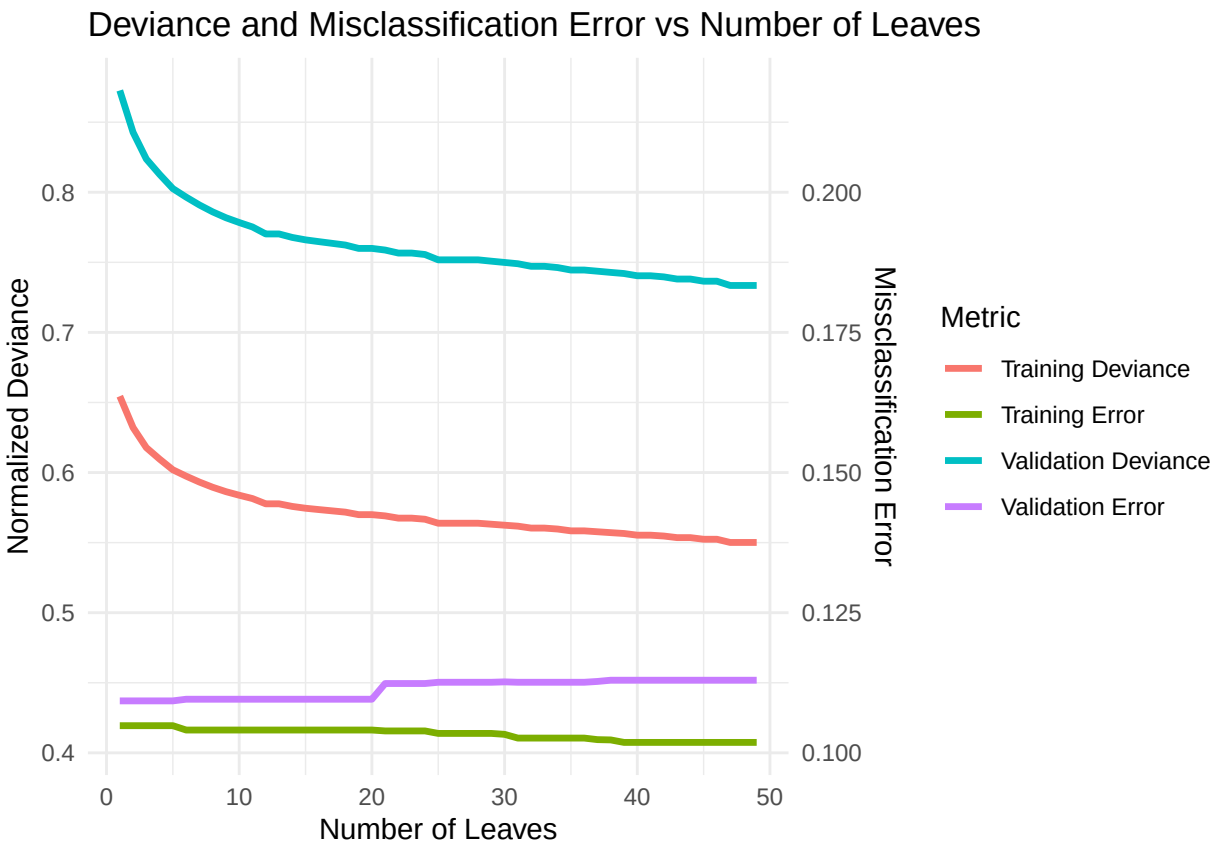
Determining the optimal tree depth

Task statement: Use training and validation sets to choose the optimal tree depth in the model 2c: study the trees up to 50 leaves. Present a graph of the dependence of deviances for the training and the validation data on the number of leaves and interpret this graph in terms of bias-variance tradeoff. Report the optimal

amount of leaves and which variables seem to be most important for decision making in this tree. Interpret the information provided by the tree structure (not everything but most important findings).

```
# Prune the tree to have up to 50 terminal nodes
pruned_trees <- lapply(2:50, function(k) prune.tree(min_deviance_tree, best = k))

# Calculate deviances and misclassification errors for training and validation sets
results <- data.frame(
  Leaves = integer(),
  Train_Deviance = numeric(),
  Valid_Deviance = numeric(),
  Train_Error = numeric(),
  Valid_Error = numeric()
)

for (i in seq_along(pruned_trees)) {
  pruned_tree <- pruned_trees[[i]]

  # Deviances
  train_dev <- deviance(pruned_tree)
  valid_dev <- deviance(pruned_tree, newdata = valid)

  # Misclassification error
  train_pred <- predict(pruned_tree, train, type = "class")
  valid_pred <- predict(pruned_tree, valid, type = "class")
  train_error <- mean(train_pred != train$y)
  valid_error <- mean(valid_pred != valid$y)

  results <- rbind(results, data.frame(
    Leaves = i,
    Train_Deviance = train_dev,
    Valid_Deviance = valid_dev,
    Train_Error = train_error,
    Valid_Error = valid_error
  ))
}

# Normalize the deviance for comparability
results <- results %>%
  mutate(
    Train_Deviance_Normalized = Train_Deviance / nrow(train),
    Valid_Deviance_Normalized = Valid_Deviance / nrow(valid)
  )

# Create the plot
p <- ggplot(results, aes(x = Leaves)) +
  # Deviances on primary y-axis
  geom_line(aes(y = Train_Deviance_Normalized, color = "Training Deviance"), linewidth = 1.2) +
  geom_line(aes(y = Valid_Deviance_Normalized, color = "Validation Deviance"), linewidth = 1.2) +

  # Errors on secondary y-axis
  geom_line(aes(y = Train_Error * 4, color = "Training Error"), linewidth = 1.2) +
  geom_line(aes(y = Valid_Error * 4, color = "Validation Error"), linewidth = 1.2) +
```

```r
  # Y-axis scaling
  scale_y_continuous(
    name = "Normalized Deviance",  # Primary y-axis label
    sec.axis = sec_axis(~ . / 4, name = "Missclassification Error")  # Secondary y-axis scaling and lab
  ) +

  # Labels and theme
  labs(title = "Deviance and Misclassification Error vs Number of Leaves",
       x = "Number of Leaves", color = "Metric") +
  theme_minimal()


print(p)
```



The deviance plot for both the training and validation datasets steadily decreases with an increasing number of leaf nodes. This behavior is unexpected, as we would typically expect the validation deviance to increase at some point due to overfitting as the model grows more complex and adapts to the noise in the training data.

A likely explanation for this behavior lies in the stark imbalance of the dataset. In such cases, more confident predictions for the majority class can dominate the overall deviance, effectively offsetting the expected increase in validation deviance caused by overfitting.

To address this, we also plotted the misclassification error for both datasets. On the validation set, the misclassification error remains relatively stable until around 20 leaves. Beyond 25 leaves, however, the misclassification error on the training set decreases noticeably. This suggests that the model initially reduces

bias until approximately 20 leaves, at which point it begins to overfit. After this point, the variance increases, and the model becomes less generalizable despite continued improvements in deviance.

In summary, the bias is reduced up to a leaf count of 20, after which the model starts to overfit. This conclusion is supported by the behavior of the misclassification error, even if the validation deviance does not show the expected increase due to the imbalanced dataset.

To determine the importance of an individual feature we can compute the reduction of deviance that each feature is responsible for. This is achieved by comparing the deviance of the two child nodes after a split and comparing it with the deviance of the parent node. This is done for every single non-leaf node and the reduction is then summed over the features. This leads to the following result:

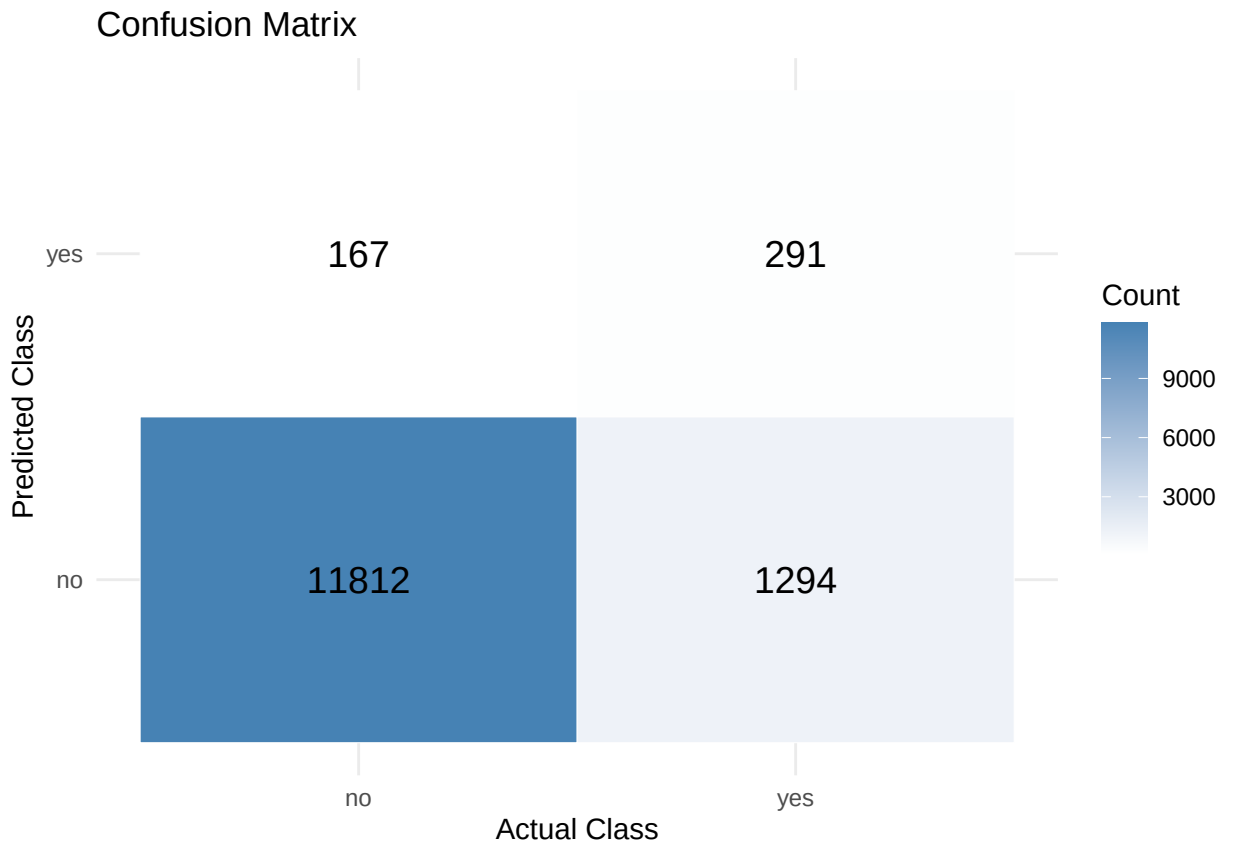
Task statement: Estimate the confusion matrix, accuracy and F1 score for the test data by using the optimal model from step 3. Comment whether the model has a good predictive power and which of the measures (accuracy or F1-score) should be preferred here.

```
# Prune tree to 20 leaves
pruned_tree_20 <- prune.tree(min_deviance_tree, best = 20)

# Extract the frame of the tree
tree_frame <- pruned_tree_20$frame
node_numbers <- as.numeric(rownames(tree_frame))  # Get node numbers

# Initialize a data frame to store deviance reductions
deviance_reduction <- data.frame(Variable = character(), Reduction = numeric())

# Loop through all internal nodes
for (i in seq_len(nrow(tree_frame))) {
  # Skip leaf nodes
  if (tree_frame$var[i] == "<leaf>") next

  # Parent node deviance
  parent_node <- node_numbers[i]
  parent_dev <- tree_frame$dev[i]

  # Child node numbers
  left_child <- 2 * parent_node
  right_child <- 2 * parent_node + 1

  # Match child nodes in the frame
  left_dev <- tree_frame$dev[node_numbers == left_child]
  right_dev <- tree_frame$dev[node_numbers == right_child]

  # Ensure both child nodes exist before calculating reduction
  if (length(left_dev) > 0 && length(right_dev) > 0) {
    # Calculate reduction in deviance
    reduction <- parent_dev - (left_dev + right_dev)

    # Store the result
    deviance_reduction <- rbind(deviance_reduction,
                                data.frame(Variable = tree_frame$var[i], Reduction = reduction))
  }
}

# Summarize the total reduction by variable
deviance_importance <- aggregate(Reduction ~ Variable, data = deviance_reduction, sum)
```

```r
deviance_importance <- deviance_importance[order(-deviance_importance$Reduction), ]

# Print the results
print(deviance_importance)
```

```
##   Variable  Reduction
## 8 poutcome 1014.73901
## 6    month  700.61614
## 4  contact  258.64239
## 5      day  156.22640
## 3  housing  137.56257
## 7    pdays  131.75735
## 1      age  109.64081
## 2  balance   34.25974
```

This metric indicates that the most important features are *poutcome, month and contact*, which makes sense as these are the variables which are used in the first, second and fourth split. (The third split is performed on a tiny fraction of the data and is a leaf node whose output is the majority class in both cases)




```
## Accuracy: 0.8923
```

```
## 0.6353712
```

```
## F1-Score: 0.2849
```

The models predictive power is quite low. It almost always predicts the majority class and even if it predicts the minority class it is wrong in more than one third of all cases. This leads to low recall and precision values, all the while upholding high accuracy due to the imbalanced data.

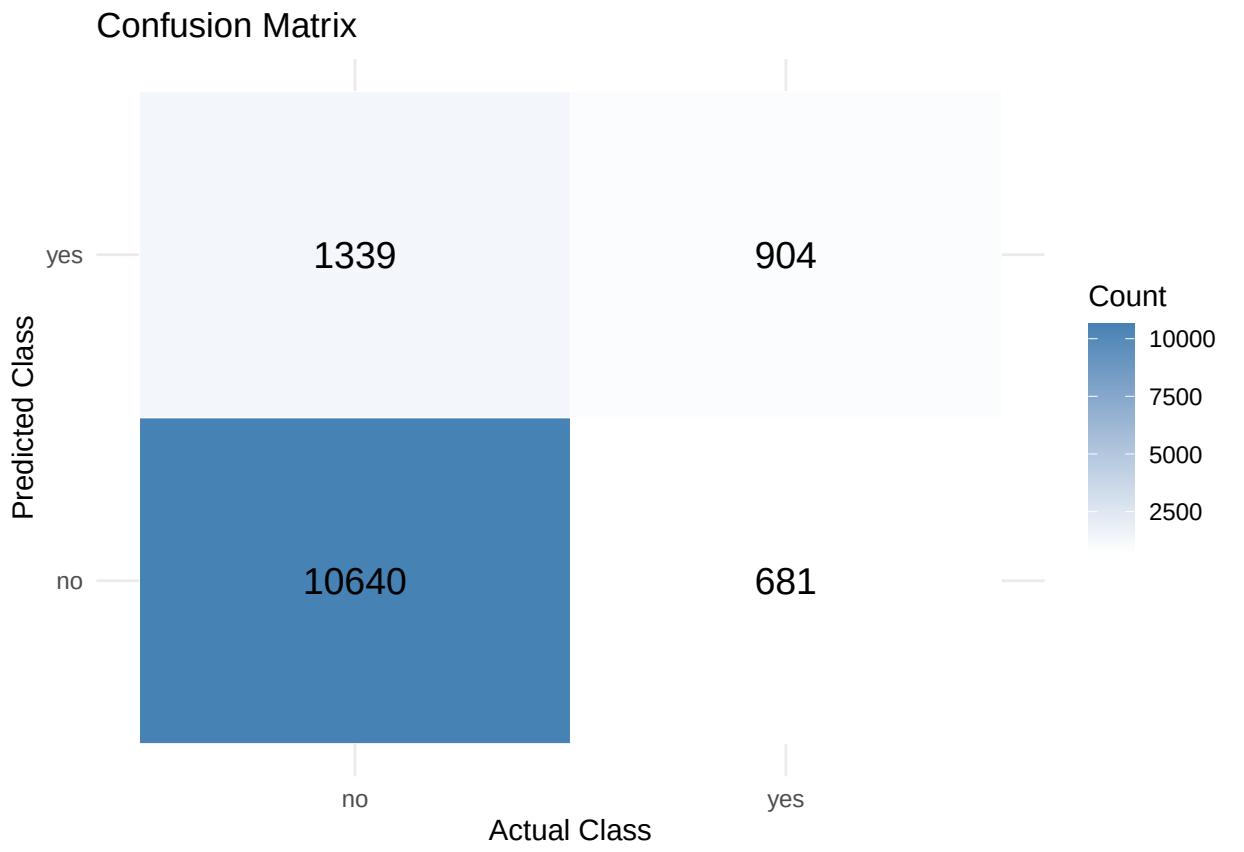
Therefore the accuracy is highly inappropriate and it is best to make use of a metric such as the F1-Score that leverages both precision and recall and therefore giving a more complete picture of the models performance.

Task statement: Perform a decision tree classification of the test data with the following loss matrix,

$$\text{Predicted} L = \text{Observed} \begin{bmatrix} & \text{yes} & \text{no} \\ \text{yes} & 0 & 5 \\ \text{no} & 1 & 0 \end{bmatrix}$$

and report the confusion matrix for the test data. Compare the results with the results from step 4 and discuss how the rates has changed and why.

```
# Define the loss matrix
loss_matrix <- matrix(c(0, 1, 5, 0), nrow = 2, byrow = TRUE,
                      dimnames = list(c("yes", "no"), c("yes", "no")))

# Train the decision tree with the loss matrix
custom_tree_rpart <- rpart(
  y ~ .,
  data = train,
  parms = list(loss = loss_matrix),
  control = rpart.control(minsplit = 1, cp = 0.001)
)
```



```
## Accuracy: 0.8511
```

```
## F1-Score: 0.4723
```

Since we are now putting a lot more emphasis on the minority class, the model has begone predicting the minority class significantly more aggressively, which has lead to a large increase in recall. Although we are sacrificing both on precision, as well as overall accuracy, we can confidently state that our model is now performing a lot better, which is reflected in the significantly higher F1-Score.

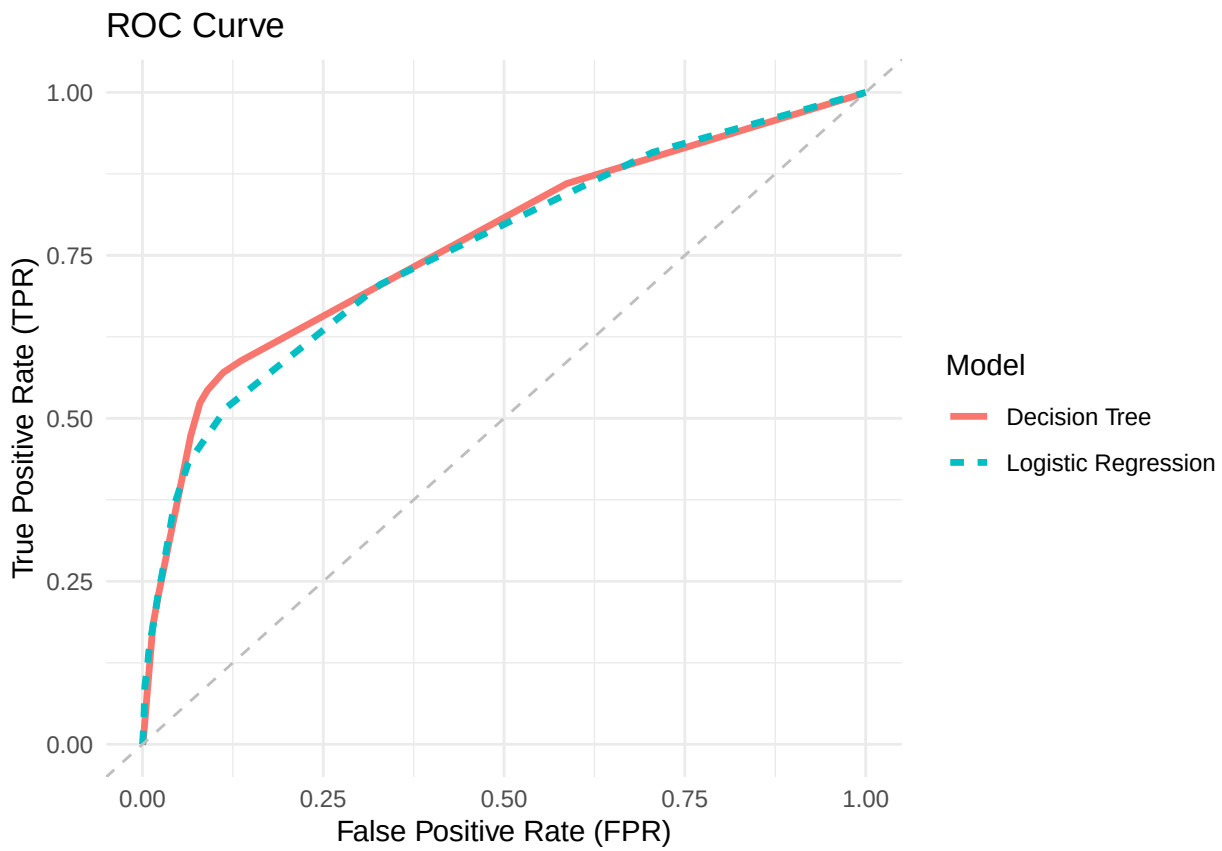
Task statement: Use the optimal tree and a logistic regression model to classify the test data by using the following principle:

$$\hat{Y} = \text{yes if } p(Y = \text{'yes'}|X) > \pi, \text{ otherwise } \hat{Y} = \text{no}$$

where $\pi = 0.05, 0.1, 0.15, ...0.9, 0.95$. Compute the TPR and FPR values for the two models and plot the corresponding ROC curves. Conclusion? Why precisionrecall curve could be a better option here?

```
tree_probs <- predict(custom_tree_rpart, test, type = "prob")[, "yes"]
logistic_model <- glm(y ~ ., data = train, family = "binomial")
logit_probs <- predict(logistic_model, test, type = "response")

# Compute TPR and FPR for both models
thresholds <- seq(0.000001, 0.999999, by = 0.05) # for a more intuitive graph

# Initialize an empty data frame to store results
roc_data <- data.frame(
  Threshold = numeric(),
  TPR = numeric(),
  FPR = numeric(),
  Model = character()
)

# Compute TPR and FPR for each threshold
for (threshold in thresholds) {
  tree_preds <- ifelse(tree_probs > threshold, "yes", "no")
  confusion_tree <- table(Predicted = tree_preds, Actual = test$y)

  TPR_tree <- if ("yes" %in% colnames(confusion_tree)) {
    confusion_tree["yes", "yes"] / sum(confusion_tree[, "yes"])
  } else {
    0
  }

  FPR_tree <- if ("no" %in% colnames(confusion_tree)) {
    confusion_tree["yes", "no"] / sum(confusion_tree[, "no"])
  } else {
    0
  }

  logit_preds <- ifelse(logit_probs > threshold, "yes", "no")
  confusion_logit <- table(Predicted = logit_preds, Actual = test$y)

  TPR_logit <- if ("yes" %in% colnames(confusion_logit)) {
    confusion_logit["yes", "yes"] / sum(confusion_logit[, "yes"])
  } else {
```

```
    0
  }

  FPR_logit <- if ("no" %in% colnames(confusion_logit)) {
    confusion_logit["yes", "no"] / sum(confusion_logit[, "no"])
  } else {
    0
  }

  # Append results for each model
  roc_data <- rbind(
    roc_data,
    data.frame(Threshold = threshold, TPR = TPR_tree, FPR = FPR_tree, Model = "Decision Tree"),
    data.frame(Threshold = threshold, TPR = TPR_logit, FPR = FPR_logit, Model = "Logistic Regression")
  )
}

# Plot ROC Curve
p <- ggplot(roc_data, aes(x = FPR, y = TPR, color = Model, linetype = Model)) +
  geom_line(linewidth = 1.2) +
  geom_abline(slope = 1, intercept = 0, linetype = "dashed", color = "grey") +
  labs(
    title = "ROC Curve",
    x = "False Positive Rate (FPR)",
    y = "True Positive Rate (TPR)",
    color = "Model",
    linetype = "Model"
  ) +
  xlim(0, 1) +
  ylim(0, 1) +
  theme_minimal()

print(p)
```



In a dataset that suffers from high class-imbalance the ROC curve can be deceiving as the False Positive Rate remains low, even if the model predicts "no" for most instances.

The precision-recall curve on the other hand focuses solely on the positive class and therefore better represents the models performance if class imbalance is a decisive factor.

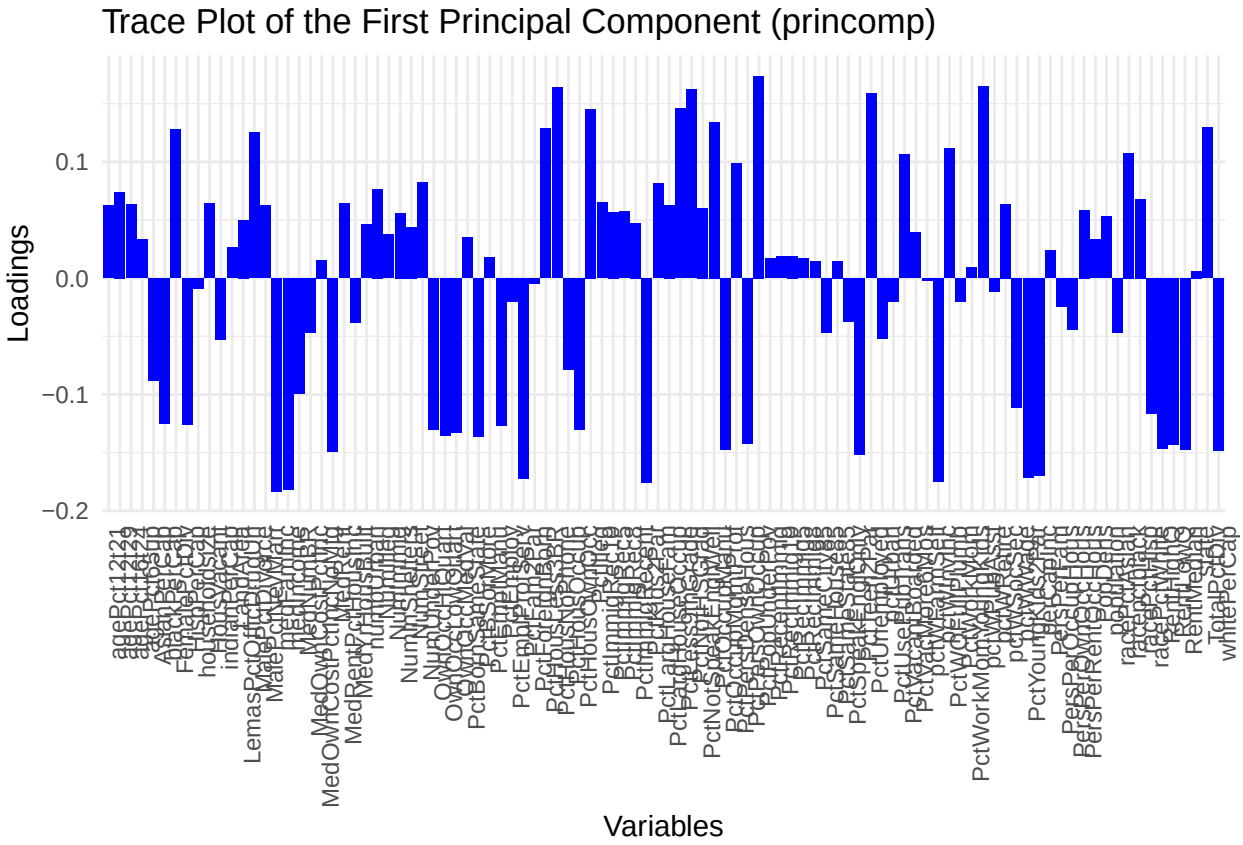
Assignment 3

Question 1

```
## Number of components to explain at least 95% of variance: 35
```

```
## Proportion of variance explained by the first two components: 0.4195296
```

Question 2




```
##                       Variable    Loading
## medFamInc           medFamInc -0.1833080
## medIncome           medIncome -0.1819830
## PctKids2Par       PctKids2Par -0.1755423
## pctWInvInc         pctWInvInc -0.1748683
## PctPopUnderPov PctPopUnderPov  0.1737978
```

medFamInc: median family income (differs from household income for non-family households) (numeric - decimal)

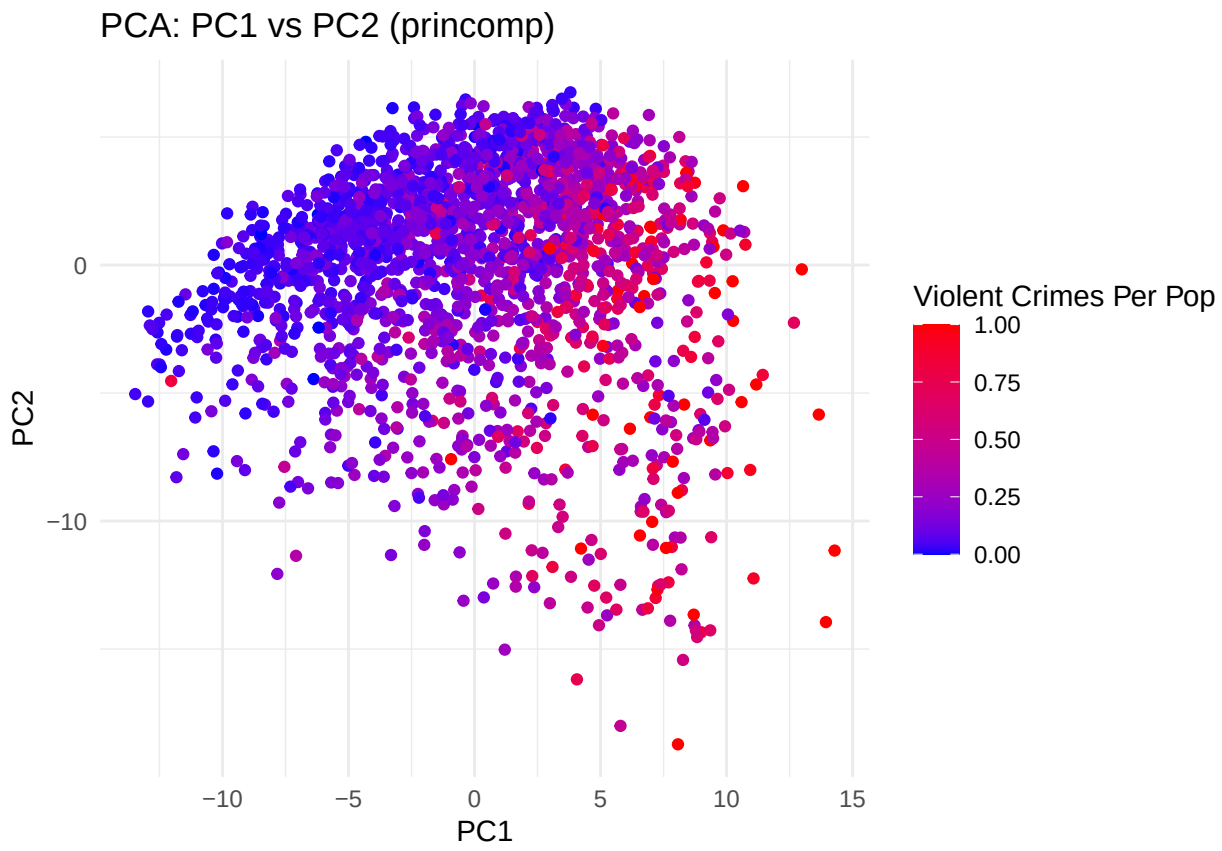
medIncome : median household income (numeric - decimal)

PctKids2Par : percentage of kids in family housing with two parents (numeric - decimal)

pctWInvInc : percentage of households with investment / rent income in 1989 (numeric - decimal)

PctPopUnderPov : percentage of people under the poverty level (numeric - decimal)

These 5 features represents the socioeconomic gradient. It seems logical that poverty can leed to crime.




```
## Correlation of PC1 with ViolentCrimesPerPop: 0.6293296
```

```
## Correlation of PC2 with ViolentCrimesPerPop: -0.2729067
```

PC1, the component representing a socioeconomic gradient dominated by poverty, has a stronger correlation with the target variable, violent crimes per population, compared to PC2. This suggests that poverty is a significant factor influencing crime rates.

The second principal component, related to recent immigration, shows a weaker correlation with crime but may still play a role indirectly, potentially through its association with socioeconomic challenges.

To deepen this analysis, we can examine the interaction between immigration and poverty within these populations to understand their combined impact on crime rates.

Question 3

```
## Training Error: 0.2752071
```

```
## Test Error: 0.382816
```

The test error is higher than the training error, but the difference is not very large. This indicates that the model generalizes reasonably well and is not heavily overfitting.

Question 4

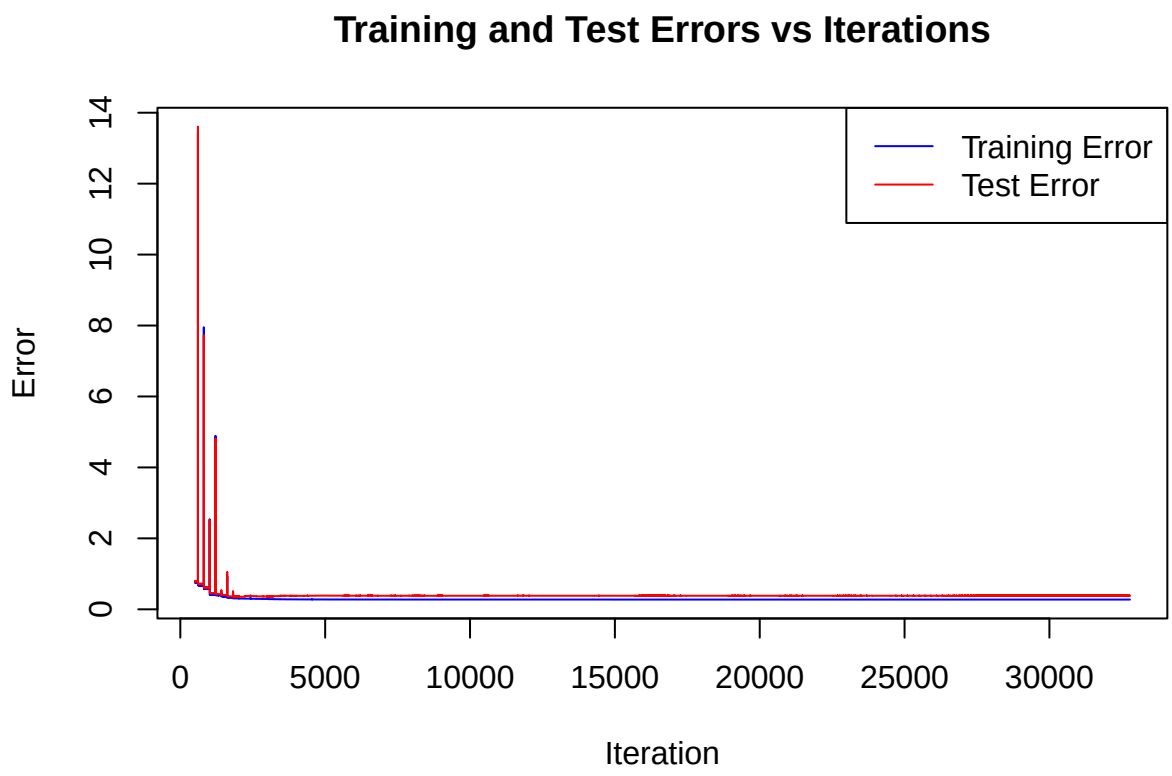
```r
# Define the cost function
linear_regression_cost <- function(theta, X, y) {
  # Compute predictions
  predictions <- X %*% theta

  # Compute MSE (mean squared error)
  mse <- mean((predictions - y)^2)

  return(mse)
}
```

```r
# Optimize using BFGS with training data
train_test_errors <- function(train_X, train_y, test_X, test_y, max_iter = 2000) {
  # Initialize theta (parameter vector) to zeros
  initial_theta <- rep(0, ncol(train_X))

  # Store training and test errors for each iteration
  train_errors <- numeric(max_iter)
  test_errors <- numeric(max_iter)

  # Define a wrapper for the optim function to track errors
  cost_tracking <- function(theta) {
    # Compute training and test errors
    train_errors[curr_iter <<- curr_iter + 1] <<- linear_regression_cost(theta, train_X, train_y)
    test_errors[curr_iter] <<- linear_regression_cost(theta, test_X, test_y)

    # Return the cost for the optim function
    return(train_errors[curr_iter])
  }

  # Initialize iteration counter
  curr_iter <<- 0

  # Use optim to minimize the cost function
  optim_res <- optim(
    par = initial_theta,
    fn = cost_tracking,
    method = "BFGS",
    control = list(maxit = max_iter)
  )

  # Return training and test errors
  return(list(train_errors = train_errors, test_errors = test_errors, optim_res = optim_res))
}
```

```r
# Use the scaled train/test datasets from Question 3
train_X <- as.matrix(train_scaled %>% select(-ViolentCrimesPerPop)) # Features
train_y <- train_scaled$ViolentCrimesPerPop                         # Target
test_X <- as.matrix(test_scaled %>% select(-ViolentCrimesPerPop))
test_y <- test_scaled$ViolentCrimesPerPop
```

```r
# Run the optimization and track errors
results <- train_test_errors(train_X, train_y, test_X, test_y, max_iter = 2000)
```

```r
# Plot the training and test errors
train_errors <- results$train_errors
test_errors <- results$test_errors

# Remove the first 500 iterations
plot_range <- 501:length(train_errors)
# Adjust the range of the y-axis to focus on the error values
error_range <- range(train_errors[plot_range], test_errors[plot_range])
```




```
## Optimal Iteration: 2182
```

```
## Training Error at Optimal Iteration: 0.3032999
```

```
## Test Error at Optimal Iteration: 0.3552952
```

The model in Part 4 generalizes well, with the optimal iteration (2182) achieving the lowest test error of 0.303. Beyond this iteration, continued optimization does not improve performance and risks overfitting.

Training Error Comparison:

Part 3: 0.275 Part 4: 0.303 The lower training error in Part 3 reflects its use of an intercept, providing greater flexibility to fit the training data.

Test Error Comparison:

Part 3: 0.382 Part 4: 0.355 The lower test error in Part 4 suggests that the simpler model without an intercept generalizes better, avoiding overfitting seen in Part 3.

While the model in Part 3 fits the training data better, the simpler model in Part 4 strikes a better balance between training and test performance, indicating improved generalization.

Theory

What are the practical approaches for reducing the expected new data error, according to the book?

Practical approaches for reducing the expected new data error (E_new): To minimize the expected new data error (E_new), it is essential to simultaneously reduce the training error (E_train) and the generalization gap. Increasing the size of the training dataset is a practical approach since it typically decreases the generalization gap while slightly increasing E_train. Adjusting the model's flexibility—either increasing it if E_train is too high or decreasing it to reduce overfitting—is another key strategy. Cross-validation can help monitor the trade-off between E_train and E_new effectively. Pages 63-71

What important aspect should be considered when selecting minibatches, according to the book?

An important aspect to consider when selecting minibatches is subsampling, as explained on page 124 of the book. Subsampling involves selecting only a subset of the training data to compute the gradient at each iteration, which reduces computational cost while maintaining sufficient information for optimization. The book emphasizes that this method efficiently balances the use of all training data over multiple iterations without needing to process the entire dataset at once, making it suitable for large datasets.

Provide an example of modifications in a loss function and in data that can be done to take into account the data imbalance, according to the book

Example of modifications in a loss function and data to address data imbalance: An example of modifications to a loss function and data to address data imbalance is provided on page 101-102. The book explains that the loss function can be modified to assign different penalties to different types of errors. For instance, in a binary classification problem, a misclassification loss can be adjusted such that predicting the positive class ($y=1$) incorrectly is considered C times more severe than predicting the negative class ($y=-1$) incorrectly. Alternatively, the data itself can be adjusted by duplicating the positive class examples C times in the dataset, effectively balancing the data without modifying the loss function.

Appendix

```r
# Assignment 1

data <- read.csv("tecator.csv", header = TRUE)

n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]
test=data[-id,]
```

```r
# 1 #

# train model using the channels as features
model_lr <- lm(Fat~.- Protein - Moisture - Sample, data=train)

# function for mean squared error
mean_squared_error <- function(true, predicted){
  error <- mean((true-predicted)^2)
  return(error)
}

# get the train error
train_error <- mean_squared_error(train$Fat, predict(model_lr))
train_error

# get the test error
test_error <- mean_squared_error(test$Fat, predict(model_lr, newdata = test))
test_error

# The model is highly overfitted, as the training error is very low, but the test error is very high


# 2 #
# \mathcal{L}(\beta) = \frac{1}{n} \sum_{i=1}^n \left( \text{Fat}_i - \beta_0 - \sum_{j=1}^{100} \beta_


# 3 #
library(glmnet)
features <- train[,2:101]
target <- train$Fat
model_lasso <- glmnet(as.matrix(features), target, alpha=1, family="gaussian")
plot(model_lasso, xvar="lambda", label=TRUE)
# Only three features at log lambda, where channel4 goes to zero ->

# Find the lambda value that results in exactly three non-zero coefficients

num_features <- apply(coef(model_lasso) != 0, 2, sum) - 1  # Exclude the intercept
lambda_values <- model_lasso$lambda
selected_lambdas <- lambda_values[which(num_features == 3)]
cat("Possible lambdas for exactly 3 features:", selected_lambdas, "\n")
log(selected_lambdas)


# 4 #

model_ridge <- glmnet(as.matrix(features), target, alpha=0, family="gaussian")
plot(model_ridge, xvar="lambda", label=TRUE)

num_features <- apply(coef(model_ridge) != 0, 2, sum) - 1  # Exclude the intercept
cat("Number of features for lambda values: ", num_features)
lambda_values <- model_ridge$lambda
selected_lambdas <- lambda_values[which(num_features == 3)]
cat("\nPossible lambdas for exactly 3 features:", selected_lambdas, "\n")
```

```r
log(selected_lambdas)


# 5 #
model_cv <- cv.glmnet(as.matrix(features), target, alpha=1, family="gaussian")
plot(model_cv)
model_cv$lambda.min
log(model_cv$lambda.min)

optimal_coefficients <- coef(cv_lasso, s = "lambda.min")
num_nonzero <- sum(optimal_coefficients != 0) - 1  # Exclude intercept
cat("Number of selected features:", num_nonzero, "\n")
predict(model_cv,newx = as.matrix(test[,2:101]))

y_test_pred <- predict(model_cv,newx = as.matrix(test[,2:101]), s="lambda.min")
mean_squared_error(test$Fat, y_test_pred)


plot(test$Fat, y_test_pred, main = "True vs Predicted Test Values",
     xlab = "True Fat", ylab = "Predicted Fat", col = "blue")
abline(0, 1, col = "red")
correlation <- cor(test$Fat, y_test_pred)
correlation

# Assignment 2

knitr::opts_chunk$set(echo = TRUE)
# Load necessary libraries
library(tree)
library(ggplot2)
library(dplyr)
library(rpart)

# Load data
data <- read.csv('bank-full.csv', sep = ";")

# Remove duration column
data <- data[, !names(data) %in% c("duration")]

# Convert all "chr" cols to factors
cols_to_convert <- c("job", "marital", "education", "default", "housing", "loan",
                     "contact", "month", "poutcome", "y") # Specify columns to convert
data[cols_to_convert] <- lapply(data[cols_to_convert], factor)

# Split the data as shown in the lecture
# Train, Test, Validation Split
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.4))
train=data[id,]
id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.3))
```

```r
valid=data[id2,]
id3=setdiff(id1,id2)
test=data[id3,]

evaluate_decision_tree <- function(train_data, valid_data, model, model_name) {
  # Predict on training and validation data
  train_pred <- predict(model, train_data, type = "class")
  valid_pred <- predict(model, valid_data, type = "class")

  # Calculate missclassification rates
  train_mis <- mean(train_pred != train_data$y)
  valid_mis <- mean(valid_pred != valid_data$y)

  # Return results as a list
  list(
    Model = model_name,
    Train_Misclassification = train_mis,
    Validation_Misclassification = valid_mis,
    Model_Object = model
  )
}

train_length <- length(train[,1]) # Number of training observations

control_default <- tree.control(nobs = train_length)  # Default settings
control_node_size <- tree.control(mincut = 7000, nobs = train_length)  # Minimum node size = 7000
control_deviance <- tree.control(mindev = 0.0005, nobs = train_length)  # Minimum deviance = 0.0005

default_tree <- tree(y ~ ., data = train, control = control_default)
min_node_size_tree <- tree(y ~ ., data = train, control = control_node_size)
min_deviance_tree <- tree(y ~ ., data = train, control = control_deviance)

results <- list(
  evaluate_decision_tree(train, valid, default_tree, "Default"),
  evaluate_decision_tree(train, valid, min_node_size_tree, "Min Node Size = 7000"),
  evaluate_decision_tree(train, valid, min_deviance_tree, "Min Deviance = 0.0005")
)

results_df <- do.call(rbind, lapply(results, function(x) data.frame(
  Model = x$Model,
  "Missclassification Error Training" = round(x$Train_Misclassification, 4),
  "Missclassification Error Validation" = round(x$Validation_Misclassification, 4)
)))


tree_sizes <- sapply(results, function(x) {
  model <- x$Model_Object
  length(unique(model$where))  # Count leaf nodes
})

# Add tree sizes to the results
results_df["Tree_Size (Leaf Count)"] <- tree_sizes
knitr::kable(results_df, caption = "Results from Experiment 1")
```

```r
# Prune the tree to have up to 50 terminal nodes
pruned_trees <- lapply(2:50, function(k) prune.tree(min_deviance_tree, best = k))

# Calculate deviances and misclassification errors for training and validation sets
results <- data.frame(
  Leaves = integer(),
  Train_Deviance = numeric(),
  Valid_Deviance = numeric(),
  Train_Error = numeric(),
  Valid_Error = numeric()
)

for (i in seq_along(pruned_trees)) {
  pruned_tree <- pruned_trees[[i]]

  # Deviances
  train_dev <- deviance(pruned_tree)
  valid_dev <- deviance(pruned_tree, newdata = valid)

  # Misclassification error
  train_pred <- predict(pruned_tree, train, type = "class")
  valid_pred <- predict(pruned_tree, valid, type = "class")
  train_error <- mean(train_pred != train$y)
  valid_error <- mean(valid_pred != valid$y)

  results <- rbind(results, data.frame(
    Leaves = i,
    Train_Deviance = train_dev,
    Valid_Deviance = valid_dev,
    Train_Error = train_error,
    Valid_Error = valid_error
  ))
}

# Normalize the deviance for comparability
results <- results %>%
  mutate(
    Train_Deviance_Normalized = Train_Deviance / nrow(train),
    Valid_Deviance_Normalized = Valid_Deviance / nrow(valid)
  )

# Create the plot
p <- ggplot(results, aes(x = Leaves)) +
  # Deviances on primary y-axis
  geom_line(aes(y = Train_Deviance_Normalized, color = "Training Deviance"), linewidth = 1.2) +
  geom_line(aes(y = Valid_Deviance_Normalized, color = "Validation Deviance"), linewidth = 1.2) +

  # Errors on secondary y-axis
  geom_line(aes(y = Train_Error * 4, color = "Training Error"), linewidth = 1.2) +
  geom_line(aes(y = Valid_Error * 4, color = "Validation Error"), linewidth = 1.2) +

  # Y-axis scaling
  scale_y_continuous(
```

```
    name = "Normalized Deviance",  # Primary y-axis label
    sec.axis = sec_axis(~ . / 4, name = "Missclassification Error")  # Secondary y-axis scaling and lab
  ) +

  # Labels and theme
  labs(title = "Deviance and Misclassification Error vs Number of Leaves",
       x = "Number of Leaves", color = "Metric") +
  theme_minimal()


print(p)
# Prune tree to 20 leaves
pruned_tree_20 <- prune.tree(min_deviance_tree, best = 20)

# Extract the frame of the tree
tree_frame <- pruned_tree_20$frame
node_numbers <- as.numeric(rownames(tree_frame))  # Get node numbers

# Initialize a data frame to store deviance reductions
deviance_reduction <- data.frame(Variable = character(), Reduction = numeric())

# Loop through all internal nodes
for (i in seq_len(nrow(tree_frame))) {
  # Skip leaf nodes
  if (tree_frame$var[i] == "<leaf>") next

  # Parent node deviance
  parent_node <- node_numbers[i]
  parent_dev <- tree_frame$dev[i]

  # Child node numbers
  left_child <- 2 * parent_node
  right_child <- 2 * parent_node + 1

  # Match child nodes in the frame
  left_dev <- tree_frame$dev[node_numbers == left_child]
  right_dev <- tree_frame$dev[node_numbers == right_child]

  # Ensure both child nodes exist before calculating reduction
  if (length(left_dev) > 0 && length(right_dev) > 0) {
    # Calculate reduction in deviance
    reduction <- parent_dev - (left_dev + right_dev)

    # Store the result
    deviance_reduction <- rbind(deviance_reduction,
                                data.frame(Variable = tree_frame$var[i], Reduction = reduction))
  }
}

# Summarize the total reduction by variable
deviance_importance <- aggregate(Reduction ~ Variable, data = deviance_reduction, sum)
deviance_importance <- deviance_importance[order(-deviance_importance$Reduction), ]
```

```r
# Print the results
print(deviance_importance)
# Predict on the test data
test_pred <- predict(pruned_tree_20, test, type = "class")

# Confusion matrix
confusion_matrix <- table(Predicted = test_pred, Actual = test$y)

# Convert confusion matrix to a data frame for ggplot2
confusion_df <- as.data.frame(as.table(confusion_matrix))
colnames(confusion_df) <- c("Predicted", "Actual", "Count")

# Plot the confusion matrix
p <- confusion_plot <- ggplot(confusion_df, aes(x = Actual, y = Predicted, fill = Count)) +
  geom_tile(color = "white") +
  geom_text(aes(label = Count), size = 5) +
  scale_fill_gradient(low = "white", high = "steelblue") +
  labs(
    title = "Confusion Matrix",
    x = "Actual Class",
    y = "Predicted Class",
    fill = "Count"
  ) +
  theme_minimal()
print(p)

TP <- confusion_matrix["yes", "yes"]
FN <- confusion_matrix["no", "yes"]
TN <- confusion_matrix["no", "no"]
FP <- confusion_matrix["yes", "no"]

accuracy <- (TP + TN) / (TP + TN + FP + FN)
cat(sprintf("Accuracy: %.4f\n", accuracy))

# Calculate Precision
precision <- TP / (TP + FP)

# Calculate Recall
recall <- TP / (TP + FN)

cat(precision)
# Calculate F1-Score
f1_score <- 2 * (precision * recall) / (precision + recall)
cat(sprintf("F1-Score: %.4f\n", f1_score))

# Define the loss matrix
loss_matrix <- matrix(c(0, 1, 5, 0), nrow = 2, byrow = TRUE,
                      dimnames = list(c("yes", "no"), c("yes", "no")))

# Train the decision tree with the loss matrix
custom_tree_rpart <- rpart(
  y ~ .,
  data = train,
```

```r
  parms = list(loss = loss_matrix),
  control = rpart.control(minsplit = 1, cp = 0.001)
)

# Predict on the test data
test_pred_rpart <- predict(custom_tree_rpart, test, type = "class")

# Confusion matrix
confusion_matrix_rpart <- table(Predicted = test_pred_rpart, Actual = test$y)

# Convert confusion matrix to a data frame for ggplot2
confusion_df <- as.data.frame(as.table(confusion_matrix_rpart))
colnames(confusion_df) <- c("Predicted", "Actual", "Count")

# Plot the confusion matrix
p <- confusion_plot <- ggplot(confusion_df, aes(x = Actual, y = Predicted, fill = Count)) +
  geom_tile(color = "white") +
  geom_text(aes(label = Count), size = 5) +
  scale_fill_gradient(low = "white", high = "steelblue") +
  labs(
    title = "Confusion Matrix",
    x = "Actual Class",
    y = "Predicted Class",
    fill = "Count"
  ) +
  theme_minimal()
print(p)

TP <- confusion_matrix_rpart["yes", "yes"]
FN <- confusion_matrix_rpart["no", "yes"]
TN <- confusion_matrix_rpart["no", "no"]
FP <- confusion_matrix_rpart["yes", "no"]

accuracy <- (TP + TN) / (TP + TN + FP + FN)
cat(sprintf("Accuracy: %.4f\n", accuracy))

# Calculate Precision
precision <- TP / (TP + FP)

# Calculate Recall
recall <- TP / (TP + FN)

# Calculate F1-Score
f1_score <- 2 * (precision * recall) / (precision + recall)
cat(sprintf("F1-Score: %.4f\n", f1_score))

tree_probs <- predict(custom_tree_rpart, test, type = "prob")[, "yes"]
logistic_model <- glm(y ~ ., data = train, family = "binomial")
logit_probs <- predict(logistic_model, test, type = "response")

# Compute TPR and FPR for both models
thresholds <- seq(0.000001, 0.999999, by = 0.05) # for a more intuitive graph
```

```r
# Initialize an empty data frame to store results
roc_data <- data.frame(
  Threshold = numeric(),
  TPR = numeric(),
  FPR = numeric(),
  Model = character()
)

# Compute TPR and FPR for each threshold
for (threshold in thresholds) {
  tree_preds <- ifelse(tree_probs > threshold, "yes", "no")
  confusion_tree <- table(Predicted = tree_preds, Actual = test$y)

  TPR_tree <- if ("yes" %in% colnames(confusion_tree)) {
    confusion_tree["yes", "yes"] / sum(confusion_tree[, "yes"])
  } else {
    0
  }

  FPR_tree <- if ("no" %in% colnames(confusion_tree)) {
    confusion_tree["yes", "no"] / sum(confusion_tree[, "no"])
  } else {
    0
  }

  logit_preds <- ifelse(logit_probs > threshold, "yes", "no")
  confusion_logit <- table(Predicted = logit_preds, Actual = test$y)

  TPR_logit <- if ("yes" %in% colnames(confusion_logit)) {
    confusion_logit["yes", "yes"] / sum(confusion_logit[, "yes"])
  } else {
    0
  }

  FPR_logit <- if ("no" %in% colnames(confusion_logit)) {
    confusion_logit["yes", "no"] / sum(confusion_logit[, "no"])
  } else {
    0
  }

  # Append results for each model
  roc_data <- rbind(
    roc_data,
    data.frame(Threshold = threshold, TPR = TPR_tree, FPR = FPR_tree, Model = "Decision Tree"),
    data.frame(Threshold = threshold, TPR = TPR_logit, FPR = FPR_logit, Model = "Logistic Regression")
  )
}

# Plot ROC Curve
p <- ggplot(roc_data, aes(x = FPR, y = TPR, color = Model, linetype = Model)) +
  geom_line(linewidth = 1.2) +
  geom_abline(slope = 1, intercept = 0, linetype = "dashed", color = "grey") +
  labs(
```

```
    title = "ROC Curve",
    x = "False Positive Rate (FPR)",
    y = "True Positive Rate (TPR)",
    color = "Model",
    linetype = "Model"
  ) +
  xlim(0, 1) +
  ylim(0, 1) +
  theme_minimal()

print(p)

#Assignment 3


library(dplyr)
library(ggplot2)
library(caret)


## Question 1


communities <- read.csv("C:/Users/victo/OneDrive/Bureau/A1_SML/Machine Learning/Labs/Lab 2/communities.



# Scale all variables except 'ViolentCrimesPerPop'
communities_scaled <- communities %>%
  select(-ViolentCrimesPerPop) %>%  # Exclude the target variable
  scale()




# Compute the covariance matrix
cov_matrix <- cov(communities_scaled)

# PCA using eigen decomposition
eigen_decomp <- eigen(cov_matrix)

# Extract eigenvalues
eigen_values <- eigen_decomp$values

# Compute the proportion of variance explained
var_explained <- eigen_values / sum(eigen_values)




# Find the number of components needed to explain at least 95% variance
cum_var_explained <- cumsum(var_explained)
num_components <- which(cum_var_explained >= 0.95)[1]
```

```r
# Calculate the proportion of variance explained by the first two components
first_two_var <- sum(var_explained[1:2])

cat("Number of components to explain at least 95% of variance:", num_components, "\n")
cat("Proportion of variance explained by the first two components:", first_two_var, "\n")


## Question 2


# PCA with princomp
pca_princomp <- princomp(communities_scaled)



# PCA scores and loadings
pca_scores <- as.data.frame(pca_princomp$scores)
pc1_loadings <- pca_princomp$loadings[, 1]  # First principal component loadings




# Trace plot for the first principal component
pc1_data <- data.frame(
  Variable = colnames(communities_scaled),
  Loading = pc1_loadings
)

ggplot(pc1_data, aes(x = Variable, y = Loading)) +
  geom_bar(stat = "identity", fill = "blue") +
  theme_minimal() +
  labs(
    title = "Trace Plot of the First Principal Component (princomp)",
    x = "Variables",
    y = "Loadings"
  ) +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))




# Top 5 variables contributing to PC1
top5_pc1 <- pc1_data %>%
  arrange(desc(abs(Loading))) %>%
  slice(1:5)
print(top5_pc1)




# Add target variable to PCA scores
pca_scores$ViolentCrimesPerPop <- communities$ViolentCrimesPerPop
```

```r
# Plot PC1 vs PC2
ggplot(pca_scores, aes(x = Comp.1, y = Comp.2, color = ViolentCrimesPerPop)) +
  geom_point() +
  scale_color_gradient(low = "blue", high = "red") +
  labs(title = "PCA: PC1 vs PC2 (princomp)",
       x = "PC1",
       y = "PC2",
       color = "Violent Crimes Per Pop") +
  theme_minimal()




cor_pc1 <- cor(pca_scores$Comp.1, pca_scores$ViolentCrimesPerPop)
cor_pc2 <- cor(pca_scores$Comp.2, pca_scores$ViolentCrimesPerPop)

cat("Correlation of PC1 with ViolentCrimesPerPop:", cor_pc1, "\n")
cat("Correlation of PC2 with ViolentCrimesPerPop:", cor_pc2, "\n")


## Question 3


# Set seed for reproducibility
set.seed(12345)

# Number of rows in the dataset
n <- nrow(communities)

# Partition the data into training (50%) and test (50%) sets
id <- sample(1:n, floor(n * 0.5))  # Randomly select 50% of the data for training
train_data <- communities[id, ]
test_data <- communities[-id, ]

# Scale features and the target variable in both training and test datasets
train_scaled <- train_data %>%
  mutate(across(-ViolentCrimesPerPop, scale)) %>%
  mutate(ViolentCrimesPerPop = scale(ViolentCrimesPerPop))

test_scaled <- test_data %>%
  mutate(across(-ViolentCrimesPerPop, scale)) %>%
  mutate(ViolentCrimesPerPop = scale(ViolentCrimesPerPop))




# Fit linear regression model
model <- lm(ViolentCrimesPerPop ~ ., data = train_scaled)

# Compute training error
train_predictions <- predict(model, train_scaled)
train_error <- mean((train_predictions - train_scaled$ViolentCrimesPerPop)^2)
```

```r
# Compute test error
test_predictions <- predict(model, test_scaled)
test_error <- mean((test_predictions - test_scaled$ViolentCrimesPerPop)^2)



cat("Training Error:", train_error, "\n")
cat("Test Error:", test_error, "\n")


## Question 4



# Define the cost function
linear_regression_cost <- function(theta, X, y) {
  # Compute predictions
  predictions <- X %*% theta

  # Compute MSE (mean squared error)
  mse <- mean((predictions - y)^2)

  return(mse)
}




# Optimize using BFGS with training data
train_test_errors <- function(train_X, train_y, test_X, test_y, max_iter = 2000) {
  # Initialize theta (parameter vector) to zeros
  initial_theta <- rep(0, ncol(train_X))

  # Store training and test errors for each iteration
  train_errors <- numeric(max_iter)
  test_errors <- numeric(max_iter)

  # Define a wrapper for the optim function to track errors
  cost_tracking <- function(theta) {
    # Compute training and test errors
    train_errors[curr_iter <<- curr_iter + 1] <<- linear_regression_cost(theta, train_X, train_y)
    test_errors[curr_iter] <<- linear_regression_cost(theta, test_X, test_y)

    # Return the cost for the optim function
    return(train_errors[curr_iter])
  }

  # Initialize iteration counter
  curr_iter <<- 0

  # Use optim to minimize the cost function
  optim_res <- optim(
```

```
    par = initial_theta,
    fn = cost_tracking,
    method = "BFGS",
    control = list(maxit = max_iter)
  )

  # Return training and test errors
  return(list(train_errors = train_errors, test_errors = test_errors, optim_res = optim_res))
}




# Use the scaled train/test datasets from Question 3
train_X <- as.matrix(train_scaled %>% select(-ViolentCrimesPerPop)) # Features
train_y <- train_scaled$ViolentCrimesPerPop                         # Target
test_X <- as.matrix(test_scaled %>% select(-ViolentCrimesPerPop))
test_y <- test_scaled$ViolentCrimesPerPop



# Run the optimization and track errors
results <- train_test_errors(train_X, train_y, test_X, test_y, max_iter = 2000)



# Plot the training and test errors
train_errors <- results$train_errors
test_errors <- results$test_errors

# Remove the first 500 iterations
plot_range <- 501:length(train_errors)
# Adjust the range of the y-axis to focus on the error values
error_range <- range(train_errors[plot_range], test_errors[plot_range])




# Plot training and test errors
plot(plot_range, train_errors[plot_range], type = "l", col = "blue",
     xlab = "Iteration", ylab = "Error",
     main = "Training and Test Errors vs Iterations",
     ylim = error_range)
lines(plot_range, test_errors[plot_range], col = "red")
legend("topright", legend = c("Training Error", "Test Error"),
       col = c("blue", "red"), lty = 1)




# Identify the iteration with the minimum test error
optimal_iteration <- which.min(test_errors)
```

```
cat("Optimal Iteration:", optimal_iteration, "\n")
cat("Training Error at Optimal Iteration:", train_errors[optimal_iteration], "\n")
cat("Test Error at Optimal Iteration:", test_errors[optimal_iteration], "\n")
```