# Lab Report 2: Machine Learning 732A99/732A68/TDDE01, Group B37

Erik Lundqvist (erilu777) Axel Strid (axest556)
Oliver Solvang Stoltz (oliso325)

December 13, 2024

## Statement of Contribution

- Erik Lundqvist: Coded assignment 1 together with Axel and Oliver. Worked on the code on tasks 2.1 - 2.2 together with Axel and Oliver. Worked on the code on assignment 3 together with Oliver. Wrote the lab report for assignment 3.
- Axel Strid: Coded assignment 1 together with Erik and Oliver. Started together with Erik and Oliver on assignment 2 with coding (2.1 - 2.2), and then finished the rest of the coding on assignment 2 by myself. Wrote the report regarding assignment 2.
- Oliver Solvang Stoltz: Coded assignment 1 together with Erik and Axel. Worked on the code on tasks 2.1 - 2.2 together with Axel and Erik. Worked on the code on assignment 3 together with Erik. Wrote the report for assignment 1 and 4

# 1 Explicit Regularization

Assignment 1 was to use a dataset with infrared absorbance spectrum to predict the fat content of samples of meat. in this assignment had five different subtask involving: Linear Regression, LASSO Regression, LASSO Model, Ridge regression and Cross-validation.

## 1.1 Linear Regression Analysis

This assignment began by splitting the data into a training and test dataset, each comprising 50% of the total data. The underlying probabilistic model is given by:

$$\text{Fat} \sim \mathcal{N}(\theta_0 + \theta_1 \cdot \text{Channel}_1 + \cdots + \theta_{100} \cdot \text{Channel}_{100}, \sigma^2),$$

where $\sigma = 0.3191$, $\theta_0 = -18.15$, $\theta_1 = 26530$, $\cdots$, $\theta_{100} = -12060$. After fitting the linear regression model, the *Mean Squared Error (MSE)* was calculated for both the training and test datasets, giving the following results:

$$\text{MSE}_{\text{train}} = 0.005709117, \quad \text{MSE}_{\text{test}} = 722.4294$$

The training error is low, but the test error is significantly higher, indicating overfitting in the model. Since the model is trained on all 100 channels, this leads to high model complexity, which causes overfitting in this case. Due to the significantly higher $\text{MSE}_{\text{test}}$, the overall model quality is low.

## 1.2 LASSO Regression Cost Function

The cost function that should be used for be optimized for this scenario is

$$\hat{\boldsymbol{\theta}}_{\text{lasso}} = \arg\min_{\boldsymbol{\theta}} \left\{ \frac{1}{n} \sum_{i=1}^{107} (y_i - \theta_0 - \theta_1 x_{1i} - \cdots - \theta_{100} x_{100i})^2 + \lambda \sum_{j=1}^{100} |\theta_j| \right\}$$

where $\lambda > 0$ is a penalty factor. The values of 107 and 100 correspond to the number of data points in the training dataset and the number of channels used as features, respectively.

## 1.3 LASSO Model Analysis

In this task, the training data was fitted to the LASSO model and then plotted to visualize the correlation between the LASSO coefficients and different $\log(\lambda)$ values. Each line corresponds to a LASSO coefficient, and if the coefficient is

separated from zero, that feature is used. Observing the plot, we can see that the penalty factor when only three features remain is approximately $-0.4 < \log(\lambda) < -0.1$, which leads $\lambda$ to be approximately $0.6703 \leq \lambda \leq 0.9048$. A $\lambda$ within this interval will give a model with only three features. In figure 1 the plot is shown.

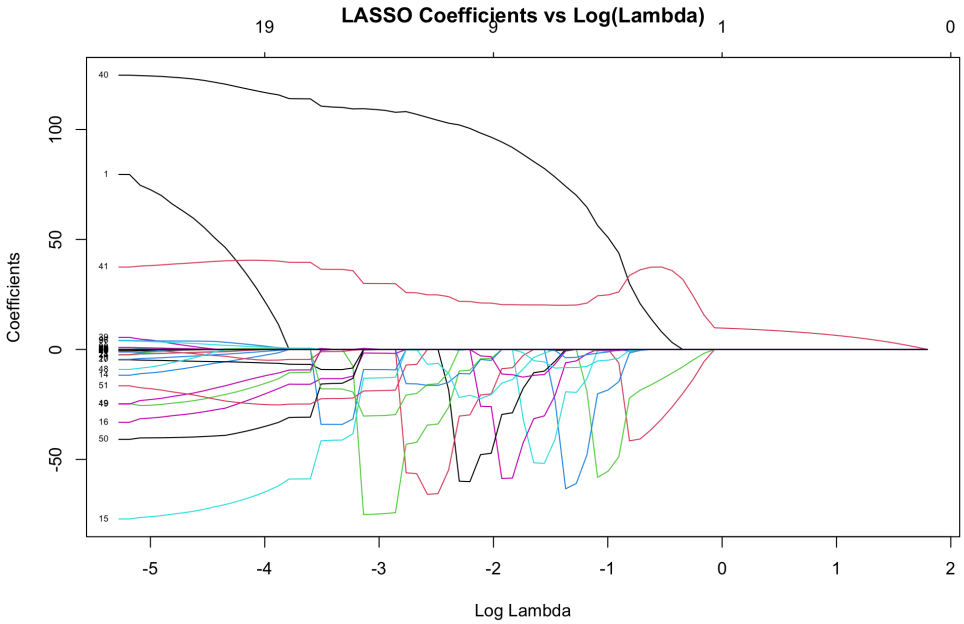


Figure 1: Relationship between LASSO coefficients and different $\log(\lambda)$ values.

## 1.4 Ridge Regression Comparison

In this task, instead of fitting the model to the LASSO model, the training data was fitted to the RIDGE model and then plotted to visualize the correlation between the Ridge coefficients and different $\log(\lambda)$ values. This plot is shown in figure 2

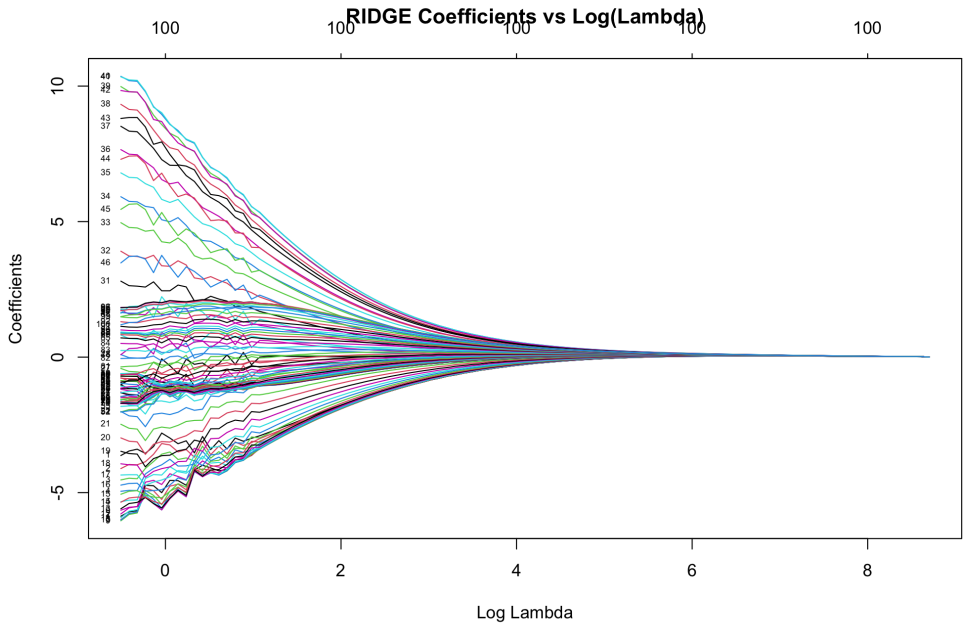


Figure 2: Relationship between RIDGE coefficients and different $\log(\lambda)$ values.

When comparing Figure 1 and Figure 2, we can see that RIDGE uses all features for every $\lambda$, while the LASSO model penalizes features, leading to the possibility of selecting fewer features. Since the $\text{MSE}_{\text{test}}$ was high, as shown in Assignment 1.1, which indicates overfitting, choosing the LASSO model would be better for this particular task.

## 1.5 Cross-Validation Analysis

In this task Cross Validation (CV) with a default number of folds was used to show the dependence between CV score and $\lambda$. In Figure 3 this plot is shown.

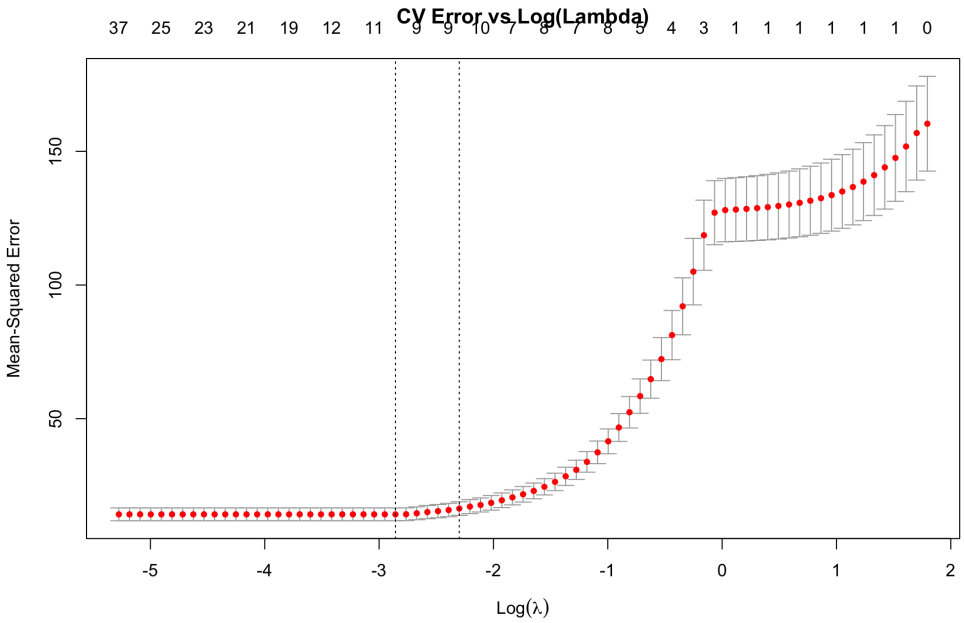


Figure 3: Dependece between Cross Validation (CV) score and different $\log(\lambda)$ values.

From this plot, we can see that with an increasing value of $\lambda$, the CV error starts to increase rapidly. This indicates that with higher $\lambda$, the model performs poorly on new unseen data. The optimal $\lambda$ was calculated to be $\lambda = 0.05744535$. With that value, the model had eight variables in addition to the intercept. If you compare the optimal value of $\lambda$ with $\log(\lambda) = -4$, the MSE is about the same. However, using the optimal $\lambda$ instead results in fewer variables, which makes the model less complex.

Finally, a scatter plot was created showing the test data compared to the predicted values for the optimal model. In Figure 4, we can see that the scatter plot demonstrates that the model is quite good at predicting fat using infrared absorbance spectra. However, there are some points, especially for higher test values, where the predictions do not work as well. Overall, the model performs reasonably well.

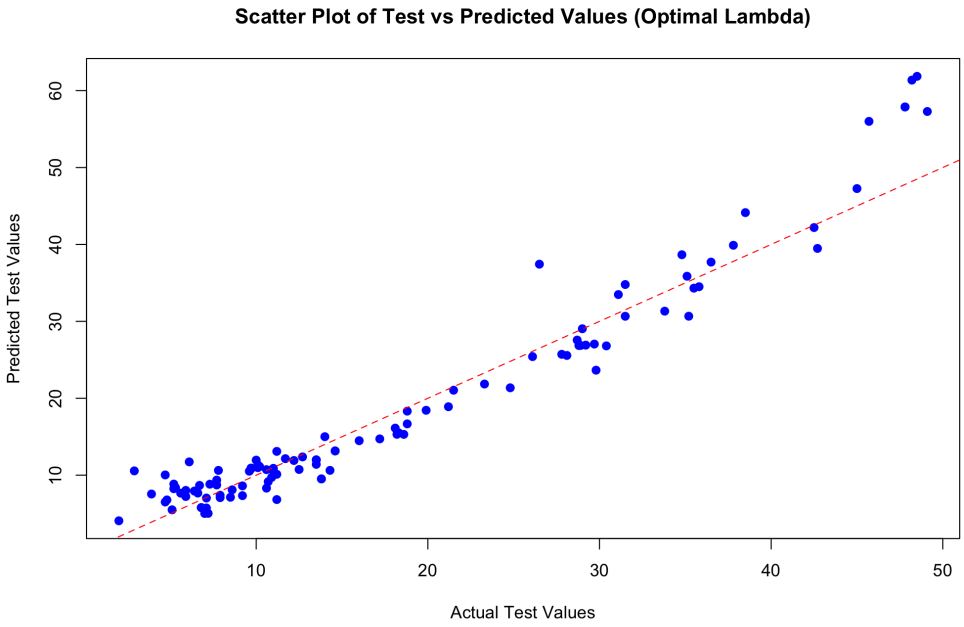


Figure 4: Scatter plot of test data compared to predicted values. The red line shows the perfect prediction.

# 2 Decision Trees and Logistic Regression

Assignment 2 focuses on predicting the outcome of a marketing campaign conducted by a Portuguese bank. The goal is to build and optimize predictive models—specifically Decision Trees and Logistic Regression—to classify whether a customer subscribed to a term deposit as a result of the campaign. By fitting these models to the training data, the assignment emphasizes improving their accuracy and understanding the impact of different parameters on model performance.

## 2.1 Data Preparation

The dataset `bank-full.csv` contains data related to a Portuguese banking institution's marketing campaigns, with the target variable indicating whether a customer subscribed to a term deposit (`y` = "yes" or "no"). We removed the `duration` column, as it depends on the outcome and would not be available before a campaign. The data was split into three sets: **40%** for training, **30%** for validation, and **30%** for testing, ensuring randomness and reproducibility by setting a fixed seed.

## 2.2 Decision Tree Models

We fitted three decision tree models with varying parameters on the training data:

- **Model A (Default):**
  - No changes to default settings.
  - Misclassification rates:
    - **Training:** 10.48%
    - **Validation:** 10.93%
    - **Tree size:** 6 terminal nodes
- **Model B (Smallest Node Size = 7000):**
  - The minimum number of observations per node was set to 7000.
  - This means that a node must have a minimum of 7000 observations to be eligible for a split. However, the observations may not be evenly distributed between the sub-nodes (for example, a split with 14,000 observations could result in 10,000 observations in the left sub-node and 4,000 in the right).
  - Misclassification rates:

    - **Training:** 10.48%
    - **Validation:** 10.93%
    - **Tree size:** 5 terminal nodes
  - This setting restricts splits, resulting in fewer but larger branches.

- **Model C (Minimum Deviance = 0.0005):**
  - Minimum deviance threshold set to 0.0005, making the tree more sensitive to potential splits.
  - This parameter control the minimum deviance for a node to be split further. The lower the value, the more sensitive the tree will be to potential splits.
  - Misclassification rates:
    - **Training:** 9.40%
    - **Validation:** 11.19%
    - **Tree size:** 122 terminal nodes
  - This resulted in a larger tree, but with higher complexity, because of the high sensitivity (low deviance value) for potential splits.

**Best Model: Model A** (Default) and **Model B** (with smallest node size of 7000) performed best in terms of validation misclassification rate. Increasing sensitivity **Model C** did not significantly improve performance.

**Model B** is preferred due to its smaller tree size, making it less complex and more interpretable.

**Model C**, on the other hand, had a much larger tree size (122 terminal nodes) and a higher validation misclassification rate (11.19%), indicating overfitting due to the higher sensitivity (lower deviance value) for potential splits.

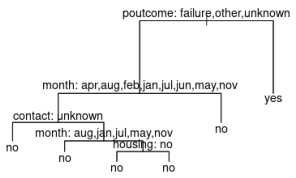


Figure 5: Model A (Default)

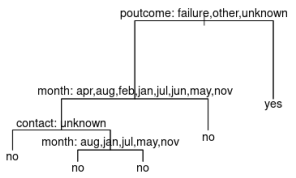


Figure 6: Model B (Smallest Node Size = 7000)

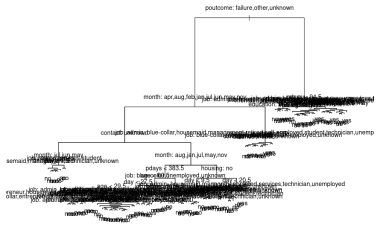


Figure 7: Model C (Minimum Deviance = 0.0005)

## 2.3 Optimal Tree Depth

To find the optimal depth, we pruned the tree from Model C (minimum deviance) for trees with **2 to 50 leaves**. The results showed that the **optimal tree** had **22 leaves**, where validation deviance was minimized.

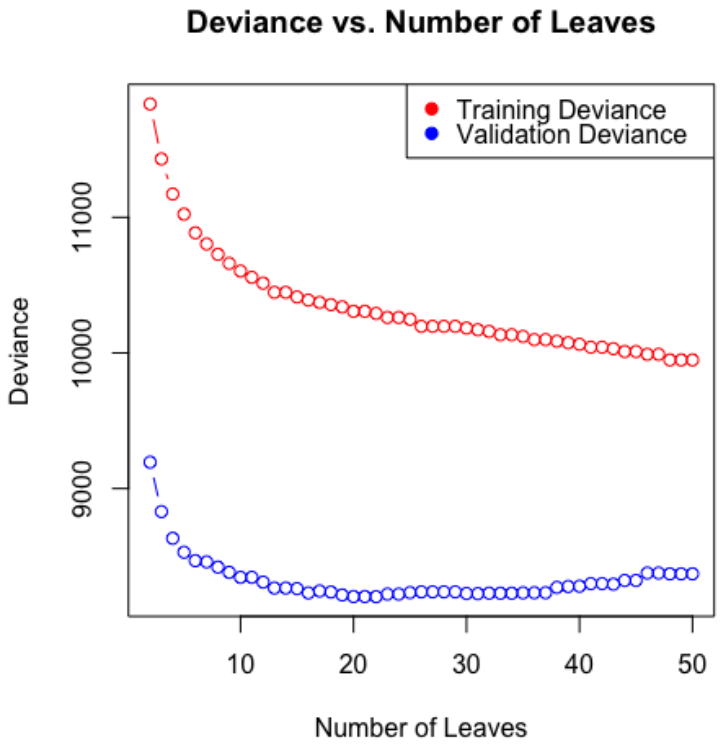


Figure 8: Dependence of training and validation deviances on the number of leaves.

**Deviance** is a measure of how well a model fits the data, particularly used for classification models. It measures the difference between the model's predicted probabilities and the actual outcomes in the data. A lower deviance suggests that the model is accurately predicting the outcomes, whereas a higher deviance suggests the opposite. In decision trees, deviance is used to determine when to stop splitting nodes, as further splits may reduce deviance on the training data, but lead to overfitting and poor generalization to new data.

**Interpretation:** The graph illustrates the relationship between the number of leaves and the deviance for both training and validation datasets. As the number of leaves increases:

- The **training deviance** decreases steadily, indicating reduced error on the training set as the tree becomes more complex.
- The **validation deviance** initially decreases, reaching its minimum at **22 leaves**, and then begins to increase. This suggests that beyond 22 leaves, the tree starts to overfit the training data, leading to poorer generalization.

This pattern reflects the **bias-variance tradeoff**:

- **Bias = Underfitting:** A smaller tree (fewer leaves) has higher bias, as it may underfit the data and fail to capture important patterns.
- **Variance = Overfitting:** A larger tree (more leaves) has higher variance, as it overfits the training data and performs worse on unseen data.
- The optimal tree with **22 leaves** strikes a balance between bias and variance, minimizing validation deviance and providing the best generalization.

**Most Important Variables:** The most significant predictors in the optimal tree were `poutcome`, `month`, and `contact`, since these are on the top level in the tree structure.

**Interpretation of Tree Structure:** The decision tree's first split is based on **poutcome** (outcome of the previous marketing campaign), indicating that previous success or failure in marketing campaigns is the strongest predictor for subscription. The second-level splits are based on **month** (the last contact month) and **contact** (the type of communication method), suggesting that the timing of the contact and the communication method also play important roles in determining the likelihood of subscription. **Month** frequently appears as a terminal node, highlighting its significance in the final decision. The tree has a total of 22 terminal nodes, with each node representing a distinct combination of factors influencing the subscription outcome.

## 2.4 Model Evaluation

We used the optimal tree (22 leaves) to predict on the test data and evaluated performance using:

- **Confusion Matrix:**

$$\begin{array}{cc} \text{True Positives (TP)} = 214 & \text{False Positives (FP)} = 107 \\ \text{False Negatives (FN)} = 1371 & \text{True Negatives (TN)} = 11872 \end{array}$$

The confusion matrix can also be presented in a table format as follows:

| **True** | **Predicted Yes** | **Predicted No** |
|---|---|---|
| **Yes** | 214 | 1371 |
| **No** | 107 | 11872 |

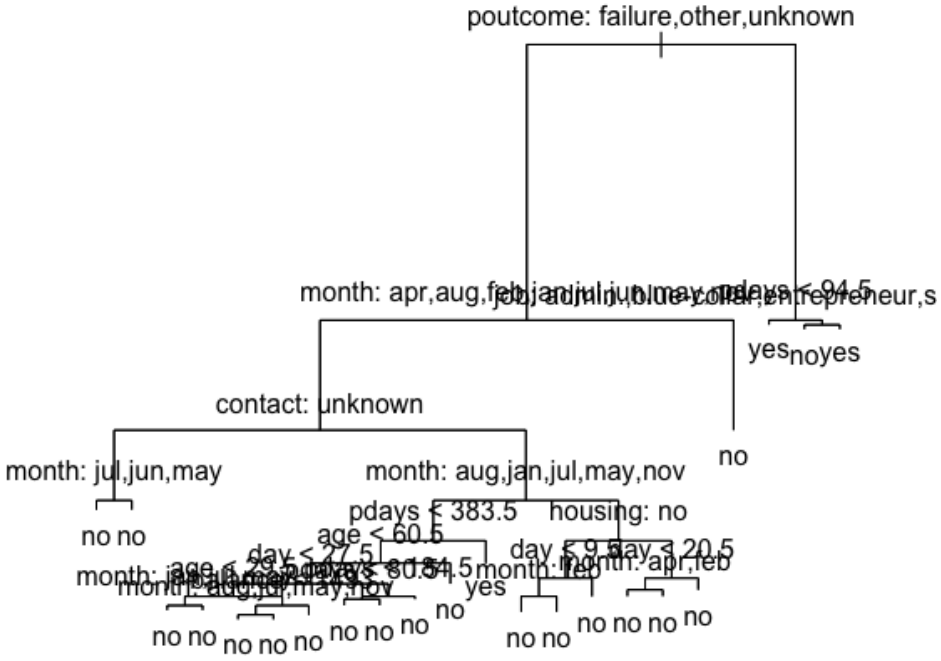


Figure 9: Decision Tree Structure for Marketing Campaign Subscription Prediction

- **Accuracy:**

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{214 + 11872}{214 + 11872 + 107 + 1371} = 0.891 = 89.1\%$$

- **Precision:**

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{214}{214 + 107} = 0.667 = 67.7\%$$

- **Recall:**

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{214}{214 + 1371} = 0.135 = 13.5\%$$

- **F1 Score:**

$$\text{F1 Score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \cdot 0.667 \cdot 0.135}{0.667 + 0.135} = 0.225 = 22.5\%$$

Given the imbalanced class distribution, **F1 score** is the preferred measure over accuracy, as accuracy does not account for the class imbalance (with many more "no" cases than "yes" cases). The F1 score, however, balances precision and recall and provides a more reliable measure in this scenario where false negatives (FN) are significant.

**Conclusion:** Despite the high accuracy of 89.1%, the model's low recall (13.5%) and low F1 score (22.5%) indicate that it struggles to correctly identify the minority class ("yes"). In this case, the F1 score is a better indicator of model performance than accuracy due to the significant imbalance between the classes.

## 2.5 Loss Matrix Analysis

We applied a custom **loss matrix** to penalize false negatives more heavily. The loss matrix was:

$$\begin{pmatrix} 0 & 5 \\ 1 & 0 \end{pmatrix}$$

This matrix assigns a higher penalty for predicting a "no" when the actual value is "yes" and a lower penalty for predicting a "yes" when the actual value is "no".

- **Confusion Matrix:**

$$\begin{array}{cc} \text{True Positives (TP)} = 0 & \text{False Positives (FP)} = 0 \\ \text{False Negatives (FN)} = 1585 & \text{True Negatives (TN)} = 11979 \end{array}$$

The confusion matrix can also be presented in a table format as follows:

| **True** | **Predicted Yes** | **Predicted No** |
|---|---|---|
| **Yes** | 0 | 1585 |
| **No** | 0 | 11979 |

- **Accuracy:**

$$\text{Accuracy} = 88.3\%$$

- **F1 Score:**

$$\text{F1 Score} = \textit{undefined}$$

**Observation:** The primary intent here is that the model should avoid predicting "no" when the true value is "yes" (i.e., false negatives) because false negatives have a higher penalty. This should ideally push the model to predict "yes" more often to avoid this costly mistake. Either we have failed in writing the correct code to solve this task, or the model becomes so conservative that it classifies every single case as "no".

## 2.6 ROC Curve Analysis

In this task, we used the optimal decision tree and a logistic regression model to classify the test data based on varying probability thresholds. The classification principle used is:

$$\hat{y} = \text{yes if } P(Y = \text{yes}|X) > z, \text{ otherwise } \hat{y} = \text{no},$$

where $z$ is a threshold value ranging from 0.05 to 0.95 in increments of 0.05. For each threshold, we computed the True Positive Rate (TPR) and False Positive Rate (FPR) for both models. These rates were then used to plot the Receiver Operating Characteristic (ROC) curves for both the optimal decision tree and logistic regression models.

Both ROC curves stop at FPR = 0.7 and TPR = 0.9, suggesting that at high thresholds, both models classify most cases as either "yes" or "no," with little distinction between them. This occurs because predicted probabilities for 'yes' become too low, leading to poor identification of the positive class. The similar curves indicate that both the decision tree and logistic regression models perform almost identically. The flattening at higher thresholds reflects limited sensitivity (recall) for the minority class, which is typical in imbalanced datasets.

**Why Precision-Recall Curve is a Better Option** The ROC curve can sometimes be misleading, especially when dealing with imbalanced datasets. In this case, a precision-recall (PR) curve could be a better option because it focuses on the performance with respect to the positive class ('yes'). The PR curve directly evaluates how well the model identifies the positive class by considering both precision (how many of the predicted positives are correct) and recall (how many of the actual positives are identified). Since the ROC curve can give overly optimistic results when there is a large class imbalance, the precision-recall curve provides a more insightful view of the model's true performance, particularly for the positive class.

Thus, a precision-recall curve would give a clearer picture of the models' ability to correctly classify positive outcomes (i.e., 'yes').

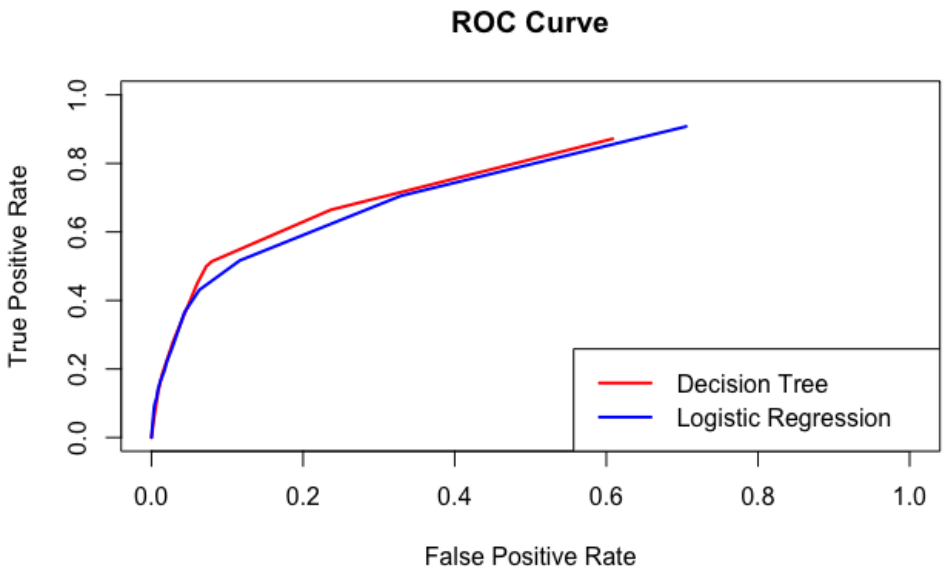


Figure 10: ROC Curve for Decision Tree and Logistic Regression Models

# 3 Principal Components and Implicit Regularization

This assignment analyzes crime rates across U.S. communities using various machine learning techniques. Principal Component Analysis (PCA) is first applied to understand the underlying structure of community characteristics, followed by linear regression modeling to predict violent crime rates. The analysis concludes with an investigation of implicit regularization through early stopping to improve model generalization.

## 3.1 PCA Implementation

To analyze the relationships between various community characteristics and crime rates, Principal Component Analysis (PCA) was performed on the communities dataset. As per the requirements, all variables except ViolentCrimesPerPop were scaled and centered using the preProcess function from the caret package. This prevents variables with larger scales from dominating the results.

After preprocessing, the covariance matrix of the scaled data was computed, and PCA was implemented using the eigen() function. The analysis showed that 35 principal components are required to explain at least 95% of the variance in

the data. The first two principal components capture a substantial portion of the total variance. Specifically:

- The first principal component explains 25.02% of the total variance
- The second principal component explains 16.93% of the total variance

## 3.2 Principal Component Analysis

A second PCA analysis was conducted using the princomp() function. The trace plot of the first principal component (Figure 11) reveals that many features have notable contributions, with income and family-related variables showing the strongest influence.

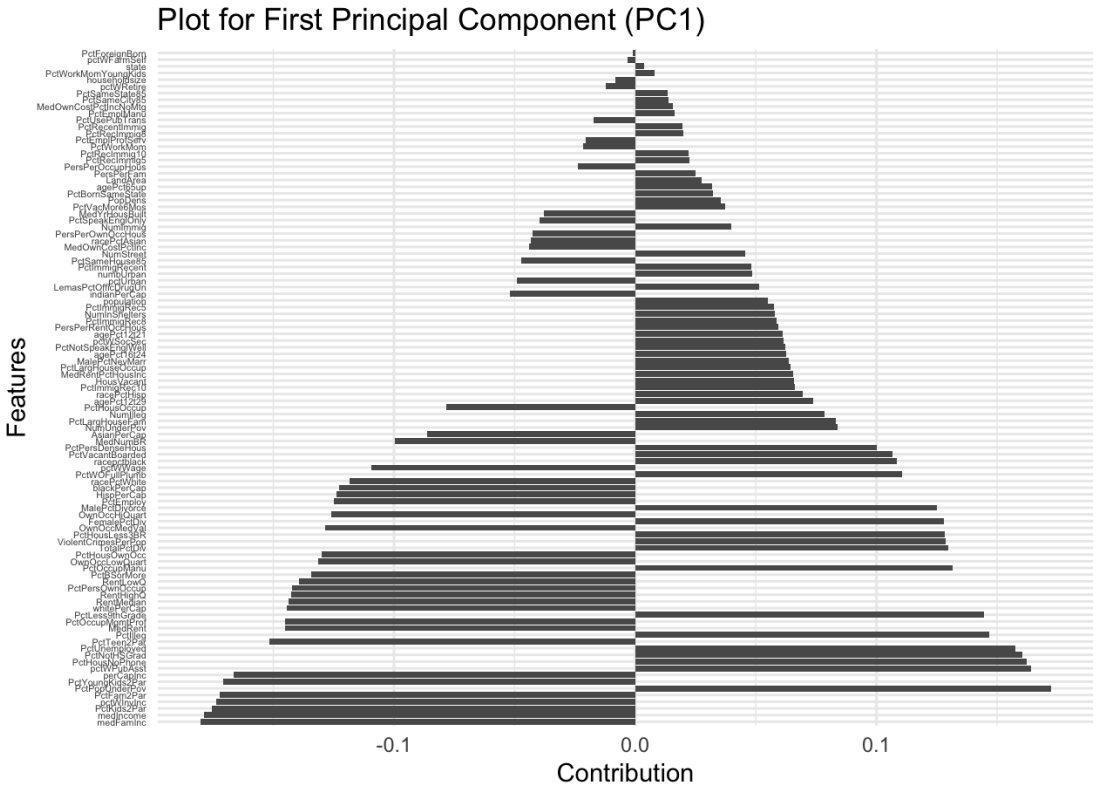


Figure 11: Feature contributions to the first principal component (PC1), with the most contributing features at the bottom

The five features with the largest absolute contributions to the first principal component are:

- Median family income (medFamInc)

- Median household income (medIncome)
- Percentage of kids in two-parent households (PctKids2Par)
- Percentage of households with investment income (pctWInvInc)
- Percentage of two-parent families (PctFam2Par)

These features share a common theme related to family structure and economic stability. Their strong contribution suggests that the first principal component primarily captures socioeconomic aspects of communities. The features with the biggest positive contributions to PC1 are percentage of population under poverty (PctPopUnderPov) and percentage of the population with public assistance in income (PctWPubAsst), showing negative correlation with the aforementioned features, which makes sense.

A visualization of the data points in PC1-PC2 space (Figure 12), colored by violent crime rates, reveals a notable pattern. Communities with higher crime rates (shown in red) tend to cluster towards the positive values of PC1, while communities with lower crime rates (shown in blue) are generally found in the negative PC1 region. This suggests a relationship between the socioeconomic factors captured by PC1 and crime rates.

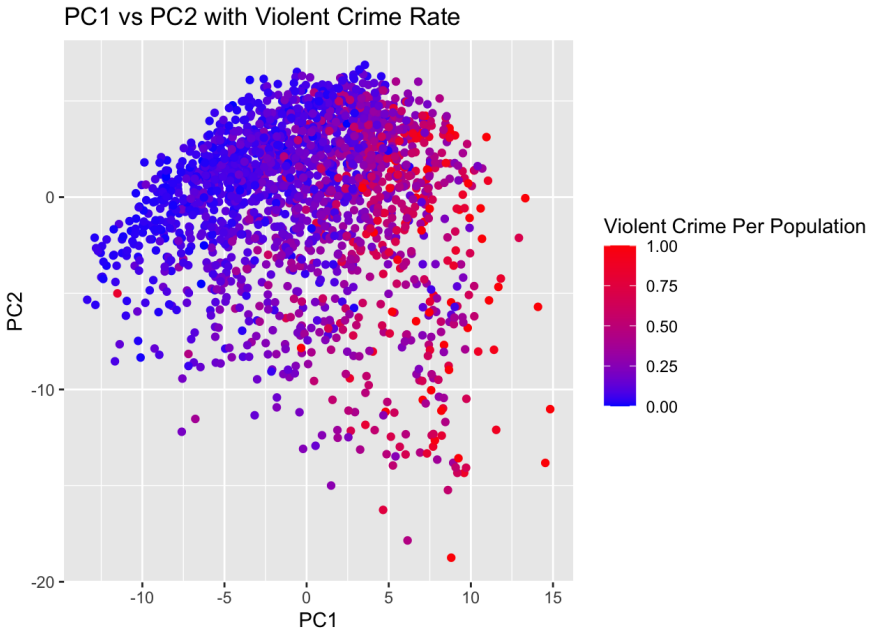


Figure 12: PC1 vs PC2 scatter plot colored by violent crime rates

## 3.3 Linear Regression Model

The data was randomly split into training and test sets with equal size (50/50 split). Both features and response variables were scaled using the preProcess function from the caret package. A linear regression model was then fitted using ViolentCrimesPerPop as the target variable and all other columns as features.

The model achieved an R-squared value of 0.7245 (adjusted R-squared: 0.6938), indicating that approximately 72% of the variance in violent crime rates can be explained by the community characteristics. The model's performance was evaluated using Mean Squared Error (MSE). The training MSE was 0.2752, while the test MSE was 0.4248. The higher test error compared to the training error indicates some degree of overfitting, suggesting that the model may not generalize as well to new, unseen data.

## 3.4 Early Stopping Analysis

To implement implicit regularization through early stopping, a cost function for linear regression without intercept was defined and optimized using the BFGS method. The cost function calculated the Mean Squared Error (MSE) on the training data, while simultaneously tracking both training and test errors at each iteration. The optimization was initialized with all parameters set to zero ($\theta_0 = 0$).

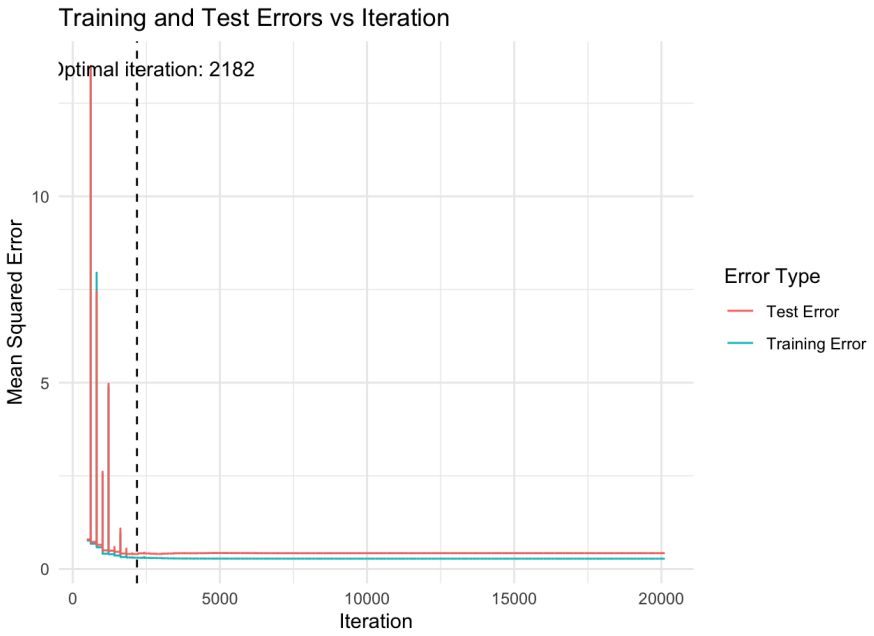


Figure 13: Training and test errors versus iteration number. Initial 500 iterations are excluded for better visualization of the error trends. The vertical dashed line indicates the optimal stopping point.

The evolution of training and test errors over iterations is shown in Figure 13. According to the early stopping criterion, the optimal iteration number was found to be 2182, where the test error reaches its minimum. At this optimal point, the training MSE is 0.3033 and the test MSE is 0.4002.

Comparing these results with the previous linear regression model (training MSE: 0.2752, test MSE: 0.4248), the early stopping approach achieves a slightly higher training error but a lower test error. This improvement in test error indicates that early stopping effectively prevents overfitting by identifying the optimal point to terminate the optimization process, resulting in better generalization performance.

# 4 Theory

In this assignment, three questions will be answered using the course book, Machine Learning: A First Course for Engineers and Scientists [1].

## 4.1 Reducing the Expected New Data Error

**What are the practical approaches for reducing the expected new data error, according to the book?** One approach to lowering the Expected New Data Error ($E_{\text{new}}$) is to increase the size of the training data, if possible. This could lead to a lower $E_{\text{new}}$, as $E_{\text{new}}$ typically decreases as $n$ increases [1, pp. 76]. Since $E_{\text{new}} = E_{\text{train}} + \text{generalization gap}$, another method to reduce $E_{\text{new}}$ could involve minimizing both $E_{\text{train}}$ and the generalization gap. This is achievable by selecting a model with the appropriate level of complexity. Increasing the model's flexibility reduces $E_{\text{train}}$, but it also increases the generalization gap. [1, pp. 76-77].

## 4.2 Aspects to concider in Minibatch Selection

**What important aspect should be considered when selecting minibatches, according to the book?** When using minibatches, it is important to ensure that the batches contain data that are well distributed and representative of the whole dataset. This can be accomplished by forming the minibatches randomly [1, pp. 125]. One implementation to achieve random minibatches is to randomly shuffle the training data and then divide it into minibatches with a fixed number of data points. After one epoch is completed, the entire training dataset is reshuffled, and the data is divided into new mini-batches to avoid them being static for the next epoch [1, pp. 125-126].

## 4.3 Data Imbalance Handling

**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.** For imbalanced datasets, the loss function can be modified to penalize certain misclassifications more heavily than others. For example, if misclassifying $Y = 1$ is $C$ times more severe than misclassifying $Y = -1$, the loss function can be changed to:

$$L(y, \hat{y}) = \begin{cases} 0 & \text{if } \hat{y} = y, \\ 1 & \text{if } \hat{y} \neq y \text{ and } y = -1, \\ C & \text{if } \hat{y} \neq y \text{ and } y = 1. \end{cases}$$

where $C$ is a factor, (Equation 5.19 in the book) [1, pp. 101-102]. To modify the data in imbalanced datasets and make them less imbalanced, the data points belonging to the underrepresented class can be duplicated $C$ times in the training dataset, where $C$ is a factor. [1, pp. 101-102].

# References

[1] A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön, *Machine Learning: A First Course for Engineers and Scientists*. Cambridge University Press, Jul. 2022. Pre-publication version. Accessed online 5 Dec 2024.

# Appendix: Code

## Assignment 1 Code

```r
#### INSTALL NECESSARY PACKAGES ####
library(glmnet)

#### DIVIDE THE DATA ####
# Load the data into a variable
data <- read.csv("tecator.csv", header = TRUE)

# Get the number of rows in the dataset
n <- nrow(data)

# Set a random seed for reproducibility
set.seed(12345)

# Partition 50% of the data for the training set
id <- sample(1:n, floor(n * 0.5))
train <- data[id, ]
test <- data[-id, ]

#### TASK 1.1 ####
# Train/fit the linear regression model, Fat is the target variable, . add all columns - takes away columns.
fit <- lm(Fat ~ . - Protein - Moisture - Sample, data = train)
summary(fit)

# Calculate MSE for training data
pred_train <- predict(fit, train)  # get predictions for training data
mse_train <- mean((train$Fat - pred_train)^2)  # calculate training MSE
mse_train # 0.005709117

# Calculate MSE for test data
pred_test <- predict(fit, test)  # get predictions for test data
mse_test <- mean((test$Fat - pred_test)^2)  # calculate test MSE
mse_test # 722.4294

```

```r
#### Task 1.2 ####

# Cost function (5,25) in text book & slide 20 in lecture 2d

#### Task 1.3 ####
# Prepare data: remove unnecessary columns
x_train <- as.matrix(train[, grep("^Channel", names(train))
    ↪ ])  # Select all Channels
y_train <- train$Fat

# Fit LASSO regression model with alpha=1
fit_lasso <- glmnet(x_train, y_train, alpha=1)
summary(fit_lasso)

plot(fit_lasso, xvar = "lambda", label = TRUE, main = "LASSO
    ↪  Coefficients vs Log(Lambda)")

#### Task 1.4 ####
# Fit LASSO regression model with alpha=1
fit_ridge <- glmnet(x_train, y_train, alpha=0)
summary(fit_ridge)

plot(fit_ridge, xvar = "lambda", label = TRUE, main = "RIDGE
    ↪  Coefficients vs Log(Lambda)")


#### Task 1.5 ####
# Perform cross-validation to find the optimal lambda
cv_lasso <- cv.glmnet(x_train, y_train, alpha=1) # 10 folds
    ↪ by default
# Plot CV error vs log(lambda)
plot(cv_lasso, main = "CV Error vs Log(Lambda)") # CV score
    ↪ = MSE
# The two vertical dashed lines correspond to:
# LambdaMIN: The value of lamda that minimizes the CV error.
# Lamda2: The largest lambda within one standard error of
    ↪ the minimum.

# Optimal lambda and corresponding number of variables
optimal_lambda <- cv_lasso$lambda.min # .min = optimal value
    ↪  (minimum CV error), min() = minimum value
optimal_lambda # 0.05744535
optimal_coef <- coef(cv_lasso, s = "lambda.min")
```

```r
optimal_coef
non_zero_vars <- sum(optimal_coef != 0) - 1  # Exclude intercept
non_zero_vars  # Number of variables = 8 + intercept

#test if it is true
x_test <- as.matrix(test[, grep("^Channel", names(test))])
y_test <- test$Fat

# Predict Fat values on the test set using the optimal lambda
pred_test <- predict(cv_lasso, s = "lambda.min", newx = x_test)

# Calculate MSE for the test set
mse_test <- mean((y_test - pred_test)^2)
mse_test

# Convert log(lambda) = -4 to lambda
lambda_at_neg4 <- exp(-4)

# Predict using the specified lambda
pred_test <- predict(cv_lasso, s = lambda_at_neg4, newx = x_test)

# Calculate MSE for the test set
mse_test <- mean((y_test - pred_test)^2)
mse_test

# scatter plot for actual vs predicted values at optimal lambda
plot(y_test, pred_test,
     xlab = "Actual Test Values",
     ylab = "Predicted Test Values",
     main = "Scatter Plot of Test vs Predicted Values (Optimal Lambda)",
     pch = 19, col = "blue")
abline(a = 0, b = 1, col = "red", lty = 2)
```

## Assignment 2 Code

```r
#### INSTALL NECESSARY PACKAGES ####
```

```r
install.packages("tree")
library(tree)
library(rpart)


#### TASK 2.1 - Divide the Data ####
# Load the data into a variable
data <- read.csv("Data/bank-full.csv",
                 header = T, sep=";",
                 stringsAsFactors = TRUE) # Note: read.csv2
                    ↪ takes the separator as ; as default

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

# Get the number of rows in the dataset
n <- dim(data)[1]

# Set a random seed for reproducibility
set.seed(12345)

# Partition 40% of the data for the training set
id <- sample(1:n, floor(n * 0.4))
train <- data[id, ]

# Partition 30% of the data for the validation set
id1 <- setdiff(1:n, id)
set.seed(12345)
id2 <- sample(id1, floor(n * 0.3))
valid <- data[id2, ]

# Use the rest for the test set
id3 <- setdiff(id1, id2)
test <- data[id3, ]




#### Task 2.2 - Different Decision Tree Models ####

# Model A - Default fit
fit_default <- tree(y ~ ., data = train)
```

```r
train_pred_default <- predict(fit_default, train, type = "class") # Prediction on train data
valid_pred_default <- predict(fit_default, valid, type = "class") # Prediction on validation data

# Calculate misclassification errors
train_mis_default <- mean(train_pred_default != train$y) # 0.1048441
valid_mis_default <- mean(valid_pred_default != valid$y) # 0.1092679


# Model B - Smallest allowed node size = 7000
# This means that a node must have minimum 7000 observations in order to be able to split
# However, a split of 14'000 can be 10'000 to left and 4'000 to right

n_train <- dim(train)[1] # Nr of obeservations (rows) in train data

# Fit the model with a smallest allowed node size of 7000
fit_min_node <- tree(y ~ .,
                     data = train,
                     control = tree.control(nobs = n_train,
                        minsize = 7000))  # Set the minimum node size to 7000

train_pred_min_node <- predict(fit_min_node, train, type = "class") # Prediction on train data
valid_pred_min_node <- predict(fit_min_node, valid, type = "class") # Prediction on validation data

# Calculate misclassification errors
train_mis_min_node <- mean(train_pred_min_node != train$y) # 0.1048441
valid_mis_min_node <- mean(valid_pred_min_node != valid$y) # 0.1092679


# Model C - Minimum deviance = 0.0005
# This parameter controls the minimum deviance for a node to be split further.
```

```r
# The lower the value, the more sensitive the tree will be to potential splits.

# Fit the tree model with a minimum deviance of 0.0005
fit_min_dev <- tree(y ~ .,
                        data = train,
                        control = tree.control(nobs = n_train, mindev = 0.0005))

train_pred_min_dev <- predict(fit_min_dev, train, type = "class") # Prediction on train data
valid_pred_min_dev <- predict(fit_min_dev, valid, type = "class") # Prediction on validation data

# Calculate misclassification errors
train_mis_min_dev <- mean(train_pred_min_dev != train$y) # 0.09400575
valid_mis_min_dev <- mean(valid_pred_min_dev != valid$y) # 0.1119221

# Visualize all trees
plot(fit_default)
text(fit_default, pretty = 0)
summary(fit_default)

plot(fit_min_node)
text(fit_min_node, pretty = 0)
summary(fit_min_node)

plot(fit_min_dev)
text(fit_min_dev, pretty = 0)
summary(fit_min_dev)



#### TASK 2.3 - Optimal Tree Depth (Number of Leaves) ####

# Clarification of Termnial Nodes, Leaves and Depth:
# Terminal nodes = Leaves = Final nodes in a decision tree with no further splits = Classification decisions
# Depth = Length of the longest path from the root node to a  terminal node (leaf)
```

```r

# Use model 2C
fit_full_tree <- fit_min_dev # Fitted with full depth from start = fully grown decision tree

# Create arrays to store deviance values for 1 to 50 terminal nodes (leaves)
trainDeviance <- rep(2,50)
validDeviance <- rep(2,50)

# Iterate over possible number of terminal nodes (leaves) from 1 to 50
for (i in 2:50) {
  # Prune the tree to have i terminal nodes
  prunedTree <- prune.tree(fit_full_tree, best = i) # best =  nr of terminal nodes (or leaves)

  # Make predictions on validation data
  valid_pred <- predict(prunedTree, newdata = valid, type = "tree")

  # Store deviance values
  trainDeviance[i] = deviance(prunedTree)
  validDeviance[i] = deviance(valid_pred)
}

# Plot train and validation deviances on nr of terminal nodes (leaves)
plot(2:50, trainDeviance[2:50], type = "b", col = "red",
     ylim = c(min(c(trainDeviance[3:50], validDeviance [3:50])), max(c(trainDeviance, validDeviance))), # Size of y-axis
     xlab = "Number of Leaves", ylab = "Deviance",
     main = "Deviance vs. Number of Leaves")
points(2:50, validDeviance[2:50], type = "b", col = "blue")
legend("topright", legend = c("Training Deviance", "Validation Deviance"),
       col = c("red", "blue"), pch = 19)

# Find optimal number of leaves <==> where validDeviance is minimized
```

```r
optimal_leaves <- which.min(validDeviance[2:50]) + 1 # Add 1 because starting from 2
optimal_leaves # 22

# Visualize tree with depth 22
optimalTree <- prune.tree(fit_full_tree, best = optimal_leaves)
plot(optimalTree)
text(optimalTree, pretty = 0)
summary(optimalTree)

# Most important variables for decision making in this tree (top nodes):
# poutcome, month, contact




#### TASK 2.4 - Confusion Matrix, Accuracy & F1 Score ####

# Confusion matrix based on test data
pred_test <- predict(optimalTree, newdata = test, type = "class") # Predictions on test set
confusion_matrix <- table(True = test$y, Predicted = pred_test)
print(confusion_matrix)

# Compute necessary variables
n_test <- dim(test)[1] # Nr of obeservations (rows) in train data
TP <- confusion_matrix["yes", "yes"] # True Positive. 214
TN <- confusion_matrix["no", "no"] # True Negative.   11872
FP <- confusion_matrix["no", "yes"] # False Positive  107
FN <- confusion_matrix["yes", "no"] # False Negative  1371

# Accuracy based on test data = (TP + TN) / n
accuracy <- (TP + TN) / n_test
accuracy # 0.8910351

# Precision, Recall, and F1 Score
precision <- TP / (TP + FP) # 0.6666667
recall <- TP / (TP + FN) # = True Positive Rates = sensitivity = recall = 0.1350157
```

```r
F1 <- (2 * precision * recall) / (precision + recall)
F1 # 0.224554

# Accuracy does not take imbalanced classes into account, but F1 score does
# Imbalanced classes meaning very large number of 1 and very small number of 0

sum(test$y == "yes") # 1585 cases
sum(test$y == "no")  # 11979 cases

# F1 is preferre due to the class imbalance.




#### TASK 2.5 - Decision Tree Classification with Loss Matrix ####

# A loss matrix assign different penalties to various types of misclassifications

# Custom Loss Matrix: TP=0, FN=5, FP=1, TN=0
loss_matrix <- matrix(c(0, 5, 1, 0), nrow = 2, byrow = TRUE,
                      dimnames = list(Observed = c("yes", "no"),
                                      Predicted = c("yes", "no")))

# Print loss matrix to verify correct setup
print(loss_matrix)

# Use "rpart" instead of "tree" library in order to use custom loss matrices
# Fit the tree on train(?) data with loss matrix
fit_tree_with_loss <- rpart(y ~ ., data = train, method = "class",
                            parms = list(loss = loss_matrix) )
summary(fit_tree_with_loss)

# Make predictions on the test data
```

```r
pred_test_loss <- predict(fit_tree_with_loss, newdata = test , type = "class")

# Confusion matrix based on test data
confusion_matrix <- table(True = test$y, Predicted = pred_test_loss)
print(confusion_matrix)




#### TASK 2.6 - Optimal Tree & Logistic Regression ####

# Find probabilities of predicting y = 'yes' --> Compare with different thresholds and classify -->
# --> Compute TPR & FPR for each threshold

# FIt a logistic regression model
fit_logistic <-  glm(y ~ ., data = train, family = "binomial")

# Predict probabilities on test data
# type = "response" will give predicted PROBABILITIES for y = 'yes'
prob_logistic <- predict(fit_logistic, newdata = test, type = "response")
# type = "vector" will give predicted probabilities for each class, select y = 'yes' using [, "yes"]
prob_tree <- predict(optimalTree, newdata = test, type = "vector")[, "yes"]

# Define thresholds
thresholds <- seq(0.05, 0.95, by = 0.05)

# Initialize vectors to store TPR and FPR values for both models
tpr_tree <- numeric(length(thresholds)) # Array with equal amount of spots as number of thresholds
fpr_tree <- numeric(length(thresholds))
tpr_logistic <- numeric(length(thresholds))
fpr_logistic <- numeric(length(thresholds))

```

```r
# Function to calculate TPR = True Positive Rate & FPR = False Positive Rates
compute_tpr_fpr <- function(probabilities, actuals, threshold) {
  predicted <- ifelse(probabilities > threshold, "yes", "no")

  # Confusion matrix
  confusion_matrix <- table(True = actuals, Predicted = predicted)
  print(confusion_matrix)

  TP <- confusion_matrix["yes", "yes"] # True Positive.
  TN <- confusion_matrix["no", "no"] # True Negative.
  FP <- confusion_matrix["no", "yes"] # False Positive
  FN <- confusion_matrix["yes", "no"] # False Negative

  # Compute TPR and FPR
  TPR <- TP / (TP + FN)  # True Positive Rate (Sensitivity = Recall)
  FPR <- FP / (FP + TN)  # False Positive Rate (1 - Specificity)

  return(c(TPR = TPR, FPR = FPR))
}

# Loop through each threshold and compute TPR and FPR for bOTh models
for (i in 1:length(thresholds)) {

  # For Decision Tree
  tpr_fpr_tree <- compute_tpr_fpr(prob_tree, test$y, thresholds[i])
  tpr_tree[i] <- tpr_fpr_tree[1]
  fpr_tree[i] <- tpr_fpr_tree[2]

  # For the Logistic Regression
  tpr_fpr_logistic <- compute_tpr_fpr(prob_logistic, test$y, thresholds[i])
  tpr_logistic[i] <- tpr_fpr_logistic[1]
  fpr_logistic[i] <- tpr_fpr_logistic[2]

}
```

```r

# PLot ROC curve for both models
plot(fpr_tree, tpr_tree, type = "l", col = "red", lwd = 2,
     xlab = "False Positive Rate", ylab = "True Positive Rate",
     main = "ROC Curve", xlim = c(0, 1), ylim = c(0, 1))
lines(fpr_logistic, tpr_logistic, col = "blue", lwd = 2)
# Add legend
legend("bottomright", legend = c("Decision Tree", "Logistic Regression"),
       col = c("red", "blue"), lwd = 2)
```

## Assignment 3 Code

```r

#### INSTALL NECESSARY PACKAGES ####
install.packages("dplyr")
install.packages("caret")
install.packages("ggplot2")
library(caret)
library(dplyr)
library(ggplot2)

#### Load the Data ####
# Load the data into a variable
data <- read.csv("communities.csv", header = T)


####################
##### Task 3.1 #####
####################

features<- data %>% select(-ViolentCrimesPerPop)

# Use caret to scale and center the data
scaler <- preProcess(features, method = c("center", "scale"))
data_scaled <- predict(scaler, data) # Scaling all varaibles except ViolentCrimesPerPop

# Compute covariance matrix
cov_matrix <- cov(data_scaled)
```

```r

# Perform PCA using eigen()
pca_result <- eigen(cov_matrix)
pca_result

# Extract eigenvalues and eigenvectors
eigenvalues <- pca_result$values
eigenvectors <- pca_result$vectors

# Proportion of variance explained by each component
eigen_percent <- eigenvalues / sum(eigenvalues)
cumulative_eigen_percent <- cumsum(eigen_percent)
eigen_percent
cumulative_eigen_percent

# Determine the number of components needed to explain at least 95% of the variance
n_commponets <- which(cumulative_eigen_percent >= 0.95)[1]
n_commponets # 35

#calculate the two most significant components,
first_component_proportion <- eigen_percent[1]
first_component_proportion # 0.2502494
second_component_proportion <- eigen_percent[2]
second_component_proportion # 0.1693082

####################
##### Task 3.2 #####
####################

pca_princomp <- princomp(data, cor = TRUE)

summary(pca_princomp)

# Extract loadings (coefficients of each feature in the principal components)
loadings <- pca_princomp$loadings[, 1]  # Loadings for PC1


loadings_df <- data.frame(
  Feature = names(loadings),
  Contribution = loadings
)
```

```r

ggplot(loadings_df, aes(x = reorder(Feature, -abs(Contribution)), y = Contribution)) +
  geom_bar(stat = "identity") +
  coord_flip() +  # Flip coordinates for better readability
  labs(title = "Plot for First Principal Component (PC1)",
       x = "Features",
       y = "Contribution") +
  theme_minimal() +
  theme(axis.text.y = element_text(size = 4))  # Adjust font size for y-axis labels


# Identify the top 5 features with the largest absolute loadings in PC1
top5_features <- sort(abs(loadings), decreasing = TRUE)[1:5]
top5_feature_names <- names(top5_features)

# Print the top 5 features in PC1
print(top5_feature_names)
print(top5_features)


# Create a data frame with PC1, PC2, and crime levels
pca_scores <- data.frame(
  FirstPC = pca_princomp$scores[, 1],  # Scores for PC1
  SecondPC = pca_princomp$scores[, 2], # Scores for PC2
  CrimeRate = data$ViolentCrimesPerPop  # Crime levels
)

# Plot PC1 vs PC2 with crime levels as color
ggplot(pca_scores, aes(x = FirstPC, y = SecondPC, color = CrimeRate)) +
  geom_point() +
  scale_color_gradient(low = "blue", high = "red") +  # Different color gradient
  labs(
    title = "PC1 vs PC2 with Violent Crime Rate",
    x = "PC1",
    y = "PC2",
    color = "Violent Crime Per Population"
  )

```

```r
####################
##### Task 3.3 #####
####################

# Get the number of rows in the dataset
n <- nrow(data)

# Set a random seed for reproducibility
set.seed(12345)

# Partition 50% of the data for the training set
id <- sample(1:n, floor(n * 0.5))
train <- data[id, ]
test <- data[-id, ]

# Scale the training data (features and target)
scaler2 <- preProcess(train, method = c("center", "scale"))
train_scaled <- predict(scaler2, train)
test_scaled <- predict(scaler2, test)


# Train/fit the linear regression model, Violent crime per pop is the target variable, . adds all columns - takes away columns.
fit <- lm(ViolentCrimesPerPop ~ . , data = train_scaled)
summary(fit)

# Predict and calculate MSE for training data
pred_train_scaled <- predict(fit, train_scaled)  # get predictions for training data
mse_train_scaled <- mean((train_scaled$ViolentCrimesPerPop - pred_train_scaled)^2)  # calculate training MSE
mse_train_scaled # 0.2752071

# Calculate MSE for test data
pred_test_scaled <- predict(fit, test_scaled)  # get predictions for training data
mse_test_scaled <- mean((test_scaled$ViolentCrimesPerPop - pred_test_scaled)^2)  # calculate training MSE
mse_test_scaled # 0.4248011

####################
##### Task 3.4 #####
```

```
####################

# Initialize storage variables and counter
train_MSE <- list()    # Will store training errors for each iteration
test_MSE <- list()     # Will store test errors for each iteration
k <- 0                 # Counter to keep track of iterations

# Find which column contains the target variable (ViolentCrimesPerPop) to make it dynamic
target_col <- grep("ViolentCrimesPerPop", colnames(train_scaled))

# Prepare matrices for training and testing
# Separate features (X) from target values (y)
train_X <- as.matrix(train_scaled[, -target_col])  # All columns except target
test_X <- as.matrix(test_scaled[, -target_col])    # All columns except target
train_y <- as.matrix(train_scaled[, target_col])   # Only target column
test_y <- as.matrix(test_scaled[, target_col])     # Only target column

# Cost function that will be used in optimization
# 1. Tracks iteration number
# 2. Calculates predictions and errors
# 3. Stores errors for later analysis
cost_function <- function(theta) {
  # Increment iteration
  .GlobalEnv$k <- .GlobalEnv$k + 1

  # Make predictions using matrix multiplication
  train_pred <- train_X %*% theta
  test_pred <- test_X %*% theta

  # Calculate MSE for both sets (MSE is our loss function)
  cost_train <- mean((train_y - train_pred)^2)
  cost_test <- mean((test_y - test_pred)^2)

  # Store MSE in global lists
  .GlobalEnv$train_MSE[[k]] <- cost_train
```

```
  .GlobalEnv$test_MSE[[k]] <- cost_test

  # Return training cost (this is what optim will try to
    ↪ minimize)
  return(cost_train)
}

# Run the optimization using BFGS method
# Start with all parameters (theta)(the coefficients) set to
  ↪  zero
initial_theta <- rep(0, ncol(train_X))
result <- optim(par = initial_theta,
                fn = cost_function,
                method = "BFGS")

# Convert lists of errors to vectors for easier plotting
train_errors <- unlist(train_MSE)
test_errors <- unlist(test_MSE)

# Start plotting from iteration 500
# Create a data frame for plotting
plot_data <- data.frame(
  iteration = 500:length(train_errors),
  train = train_errors[500:length(train_errors)],
  test = test_errors[500:length(test_errors)]
)

# Find the optimal iteration number using early stopping
  ↪ criterion
# Look for the iteration with minimum test error (after
  ↪ iteration 500)
optimal_iteration <- which.min(test_errors[500:length(test_
  ↪ errors)]) + 499

# Create the plot using ggplot2
library(ggplot2)
ggplot(plot_data, aes(x = iteration)) +
  geom_line(aes(y = train, color = "Training Error")) +
  geom_line(aes(y = test, color = "Test Error")) +
  geom_vline(xintercept = optimal_iteration,
             linetype = "dashed",
             color = "black") +
  annotate("text",
```

```
            x = optimal_iteration + 500,
            y = max(plot_data$train),
            label = paste("Optimal iteration:", optimal_
               ↪ iteration)) +
  labs(title = "Training and Test Errors vs Iteration",
       x = "Iteration",
       y = "Mean Squared Error",
       color = "Error Type") +
  theme_minimal()

# Print the results
cat("\nResults of early stopping analysis:\n")
cat("Optimal iteration number:", optimal_iteration, "\n")
cat("Training MSE at optimal iteration:", train_errors[
   ↪ optimal_iteration], "\n")
cat("Test MSE at optimal iteration:", test_errors[optimal_
   ↪ iteration], "\n")
```