Computer lab 2

Andreas Arrestam
Emma Bertmar
Manfred Clase

Contents

Statement of Contribution

During Lab 2, each team member was responsible for one assignment, which included both the code and the corresponding part of the report. Emma Bertmar was responsible for Assignment 1, Andreas Arrestam for Assignment 2, and Manfred Clase for Assignment 3. However, all team members discussed the tasks together and supported each other in completing the code and writing the report sections. Once all parts were finished, the members sat together to review the problems and solutions. This ensured that every group member understood the code, the results, and the conclusions.

1. Assignment 1. Explicit regularization

The purpose of this assignment was to evaluate and compare Linear, LASSO, and Ridge regression models for predicting meat fat content using near infrared absorbance spectrum characteristics (Channels) as features.

1.1 Linear Regression

Initially, the provided dataset was randomly divided into a training set (50%) and a test set (50%). A linear regression model was fitted to the training data using the Channels as features. The underlying probabilistic model is given by

$$\text{Fat} = \theta_0 + \theta_1 \,\text{Channel}_1 + \cdots + \theta_{100} \,\text{Channel}_{100} + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2)$$

where $\theta_0, \ldots, \theta_{100}$ are the regression coefficients and ε is the normally distributed error term with mean 0 and variance σ^2.

The fitted model was evaluated using the Mean Squared Error on both the training and test sets. The obtained errors were:

$$\text{Training MSE} \approx 0.0057 \qquad \text{Test MSE} \approx 722.43$$

The training error is very low, while the test error is very high. This large difference indicates that the model is overfitting the data. A reason for this could be that the model is complex, as it uses 100 Channels as features.

1.2 LASSO Cost Function

The cost function that should be optimized is

$$\hat{\theta}_{lasso} = \text{argmin} \left\{ \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \theta_0 - \theta_1 x_{1i} - \cdots - \theta_p x_{pi}\right)^2 + \lambda \sum_{j=1}^{p} |\theta_j| \right\}$$

Here, n is the number of observations in the training dataset, $\lambda > 0$ is the penalty factor and p is the number of Channels used as features. From the provided data, $n = 107$ and $p = 100$.

1.3 LASSO Regression

A LASSO model was fitted to the training data and a plot was created illustrating how each regression coefficient depends on $-\log(\lambda)$, see Figure **??**. As the penalty factor λ increases (and thus $-\log(\lambda)$

decreases from right to left), more coefficients are forced to be zero. The y axis in the figure shows the estimated weight of each feature.

To select a model with only three features, the plot shows that the value of the penalty factor must be approximately

$$0.1 < -\log(\lambda) < 0.4,$$
$$-0.4 < \log(\lambda) < -0.1,$$
$$e^{-0.4} < \lambda < e^{-0.1},$$
$$0.6703 < \lambda < 0.9048.$$

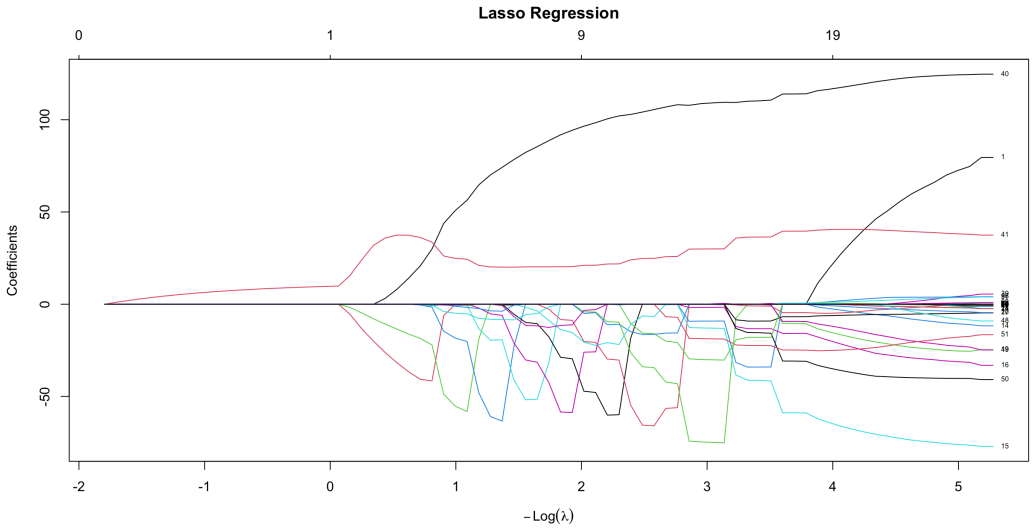


Figure 1.1: Plot of the coefficients in LASSO regression.

1.4 Ridge Regression

A Ridge model was fitted to the training data, and a plot was generated showing how each regression coefficient depends on $-\log(\lambda)$, see Figure 1.2. As the penalty factor λ increases (and thus $-\log(\lambda)$ decreases from right to left), more coefficients are shrunk down to zero but never reach exactly zero.

Comparing Figure **??** and Figure 1.2, it can be observed that LASSO regression with L1 regularization performs feature selection by setting some coefficients exactly to zero, while Ridge regression with L2 regularization only shrinks the coefficients without eliminating any features. Since the test MSE in Section 1.1 was large, indicating overfitting, LASSO regression is likely a better choice here.

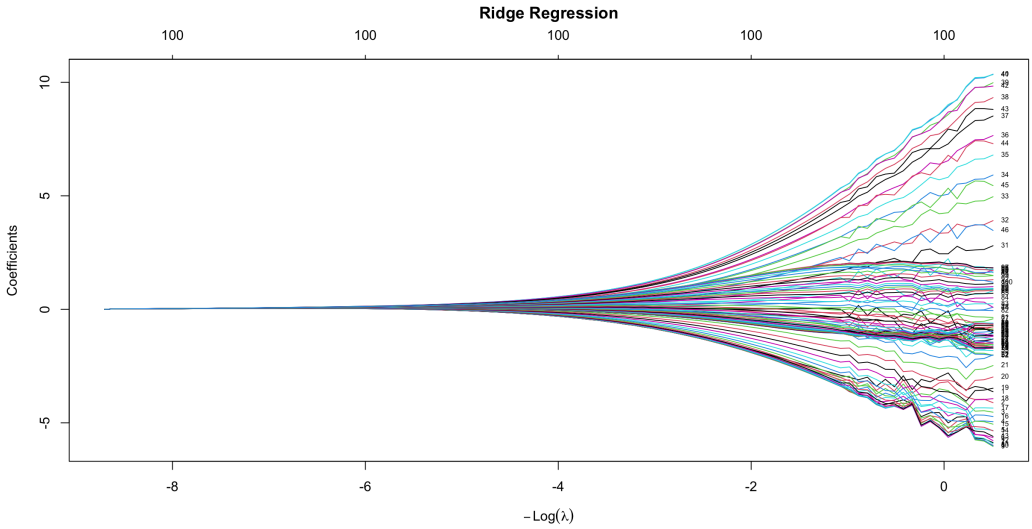


Figure 1.2: Plot of the coefficients in Ridge regression.

1.5 Optimal LASSO with cross validation

Cross validation was used to compute the optimal LASSO model, using the default number of folds. A plot was created showing how the cross validation score depends on $-\log(\lambda)$, see Figure 1.3. As the penalty factor λ increases (and thus $-\log(\lambda)$ decreases from right to left), the cross validation error increases. Therefore, for larger λ, the model performs poorly on unseen data.

The optimal penalty factor was found to be $\lambda \approx 0.0574$. With this λ, the model used nine variables, including the intercept. By comparing the optimal λ value, where $-\log(0.0574) \approx 1.2$, with $-\log(\lambda) = 4$, it can be seen in the plot that the MSE is similar for both penalty values. One difference is the number of variables used, which is fewer for the optimal λ, resulting in a less complex model.

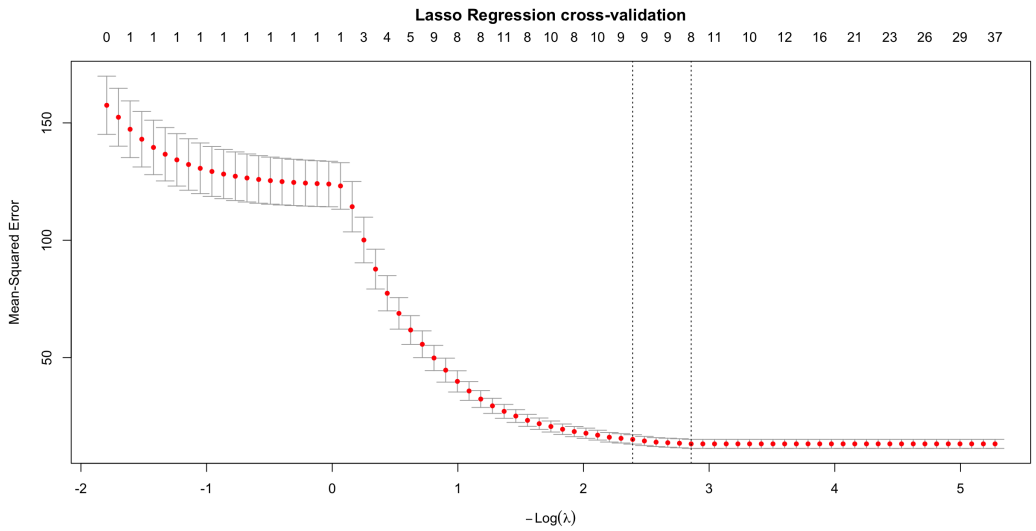


Figure 1.3: Plot of the dependence of the cross-validation score.

A scatter plot was created comparing the actual Fat test values with the predicted test values for the model corresponding to the optimal λ, see Figure 1.3. The plot indicates that the model predicts Fat reasonably well using the Channels. However, there are still some observations for which the predictions are less accurate, mostly for large predicted test values.

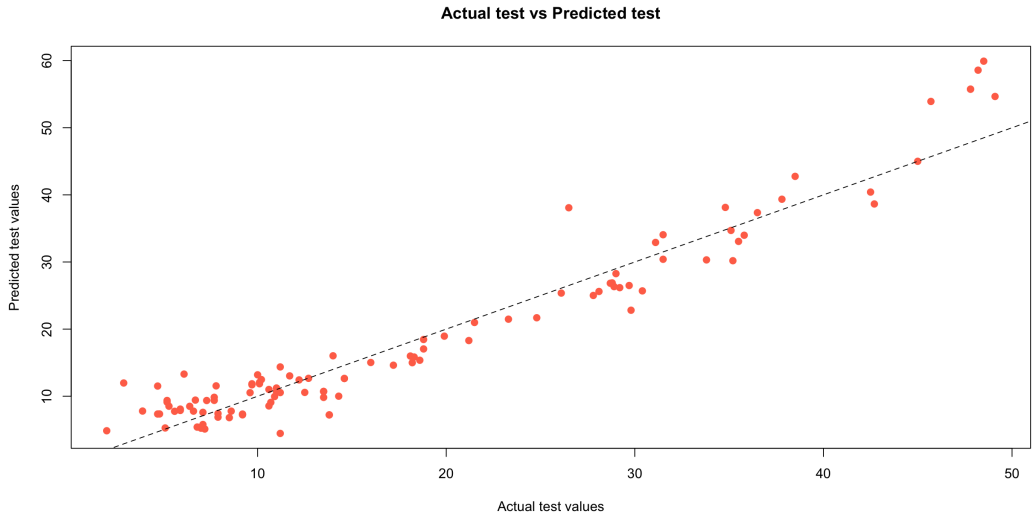


Figure 1.4: Scatter plot of actual test values compared to predicted test values.

2. Assignment 2. Decision trees and logistic regression for bank marketing

2.1 Data partitioning

For this assignment the training set was 40%, the validation set 30% and test set 30% of the complete dataset. To reduce bias, the dataset was randomized before it was divided into the different sets.

- **Training set**: Used to fit the model.
- **Validation set**: Used to tune the hyperparameters.
- **Test set**: Used to evaluate the performance of the model.

2.2 Decision trees

Three different decision trees were fitted with different settings. The first tree used the default settings, the second tree only allowed the smallest node size to be equal to 7000 and the third tree was fitted with minimum deviance 0.0005.

When setting the smallest allowed node size means that there is no node in the tree that has fewer than 7000 observations. When setting the minimum deviance to 0.0005 means that a split in the tree will only be made if it reduces the deviance by at least 0.0005.

Default settings

- Missclassification rate for training data: 10.5%
- Missclassification rate for validation data: 10.9%
- Number of terminal nodes: 6

Smallest node size equal to 7000

- Missclassification rate for training data: 10.5%
- Missclassification rate for validation data: 10.9%
- Number of terminal nodes: 5

Minimum deviance equal to 0.0005

- Missclassification rate for training data: 9.4%
- Missclassification rate for validation data: 11.1%

- Number of terminal nodes: 122

From the results, it can be observed that the models with the lowest validation error are the tree with default settings and the tree with the smallest node size equal to 7000. Both have the same validation error, but the tree with the smallest allowed node size set to 7000 is the simplest of the two, as it has less amount of terminal nodes.

In contrast, the tree trained with a minimum deviance of 0.0005 is extremely large. It achieves a much lower training error but a higher validation error, indicating that it is overfitting and therefore generalizes poorly to unseen data.

2.3 Optimal tree depth

To find the optimal tree depth, the deviance for the first 50 leaves was computed using both the training and validation sets. Figure 2.1 shows how the deviance changes as the tree grows.

The optimal tree depth is the one that produces the lowest validation deviance. From the graph, the validation deviance reaches its minimum at 22 leaves, which is therefore the optimal tree depth.

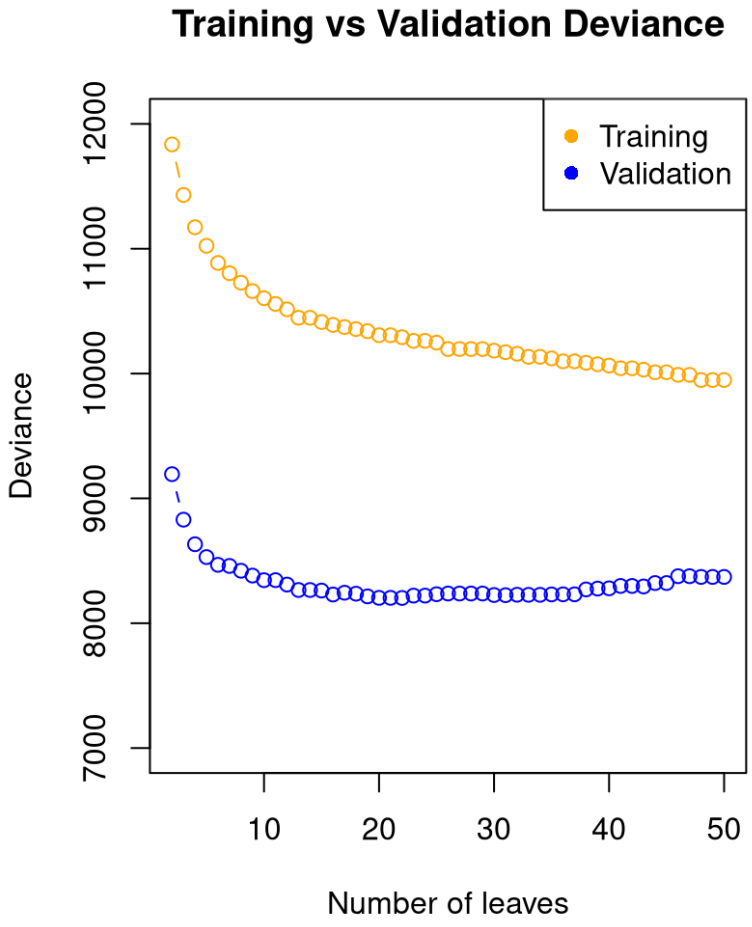


Figure 2.1: Dependence of deviance for training and validation data.

The deviance curves reveal the expected bias-variance behavior. The training deviance decreases steadily as the number of leaves increases, which means that increasing the number of trees fits the training data well, reducing bias. The validation deviance decreases initially, meaning that moderate tree growth improves generalization. After 22 leaves, the validation deviance starts increasing even though the training deviance is decreasing. This means that the model is becoming too complex.

The most important variables for decision making in this tree are the variables that are at the top level of the tree. These three were **poutcome**, **month** and **contact**. In Figure 2.2, we can see that the variable poutcome is the first split, meaning that the outcome of the previous marketing campaign has the biggest effect on whether the bank term deposit would be yes. The split at the second level is based on month, and the third level split is based on contact. The figure also indicates that month appears several times as a terminal node, which means that this variable provides the strongest and

most informative splits to predict the outcome.

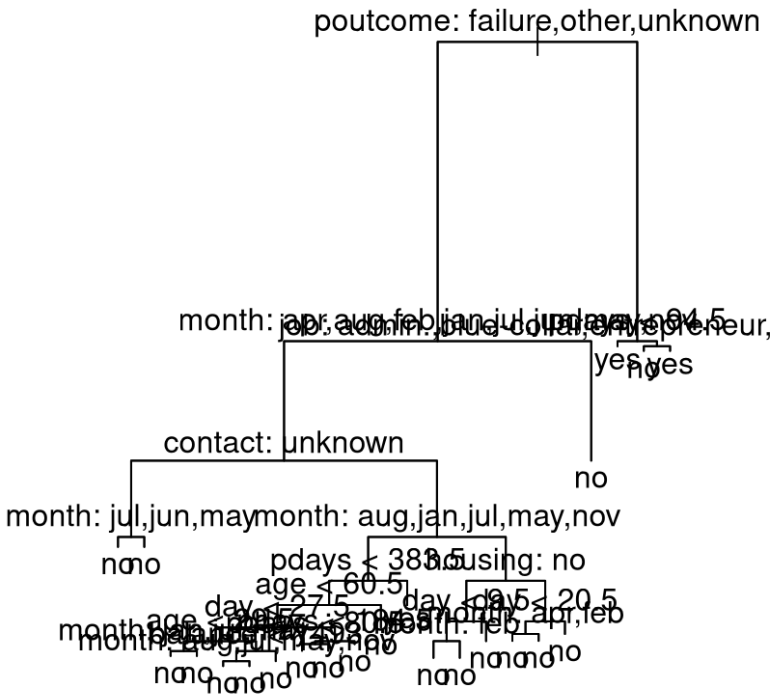


Figure 2.2: Tree with optimal tree depth.

2.4 Confusion matrix, accuracy and F1

From the optimal model in Section 2.3, a confusion matrix was constructed (see Table 2.1). Calculations for the accuracy and F1 score were also performed.

Table 2.1: Confusion matrix for the decision tree predictions.

Predicted	**no**	**yes**
Actual no	11872	107
Actual yes	1371	214

$$\textbf{Accuracy} = 1 - \frac{\text{Number of misclassified observations}}{\text{Total number of observations}} = \frac{\text{Number of correct predictions}}{\text{Total number of observations}} = 0.891$$

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

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

$$\mathbf{F_1} = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = 2 \cdot \frac{0.667 \times 0.135}{0.667 + 0.135} \approx 0.225$$

The model has high accuracy, but this is misleading since the dataset is imbalanced. Because of this, the model predicts the majority class (no) well, which results in high accuracy but fails at detecting the positive class (yes).

The imbalanced dataset makes accuracy an unreliable measure for the model. Therefore, the F1-score would be the better choice because it accounts for both precision and recall on the minority class (yes).

2.5 Classification with loss matrix

The loss matrix given in the assignment was used to classify a new decision tree on the test data. For this to work, the loss matrix needed to be swapped in the code to match the ordering of yes and no. The loss function makes it so that the cost of predicting yes when the true class is no is very costly (5x). But predicting no when the true class is yes is less costly(1).

2.5.1 Effect of Loss Matrix on Model Predictions

Table 2.2 shows the confusion matrix and performance metrics for the optimal decision tree without applying a loss matrix. The model achieves high accuracy (0.891) because it predicts the majority class (`no`) very well. However, recall (0.135) and F1-score (0.225) are very low, indicating that the model performs poorly at detecting positive cases (`yes`).

After applying the loss matrix

$$L = \begin{bmatrix} 0 & 5 \\ 1 & 0 \end{bmatrix},$$

which heavily penalizes predicting `yes` when the true class is `no`, making the model more conservative in avoiding costly false positives. The resulting confusion matrix is shown in Table 2.3.

Table 2.2: Confusion matrix for the decision tree predictions without a loss matrix.

Actual \Predicted	**no**	**yes**
no	11872	107
yes	1371	214

Table 2.3: Confusion matrix for the decision tree predictions with the loss matrix applied.

Actual \Predicted	**no**	**yes**
no	11030	949
yes	771	814

While accuracy slightly decreases (0.87), recall (0.51) and F1-score (0.48) improve significantly. This demonstrates that the loss matrix effectively adjusts the decision threshold to better detect positive cases at the cost of a small increase in false positives. Overall, the model becomes more balanced and suitable for imbalanced datasets where detecting positives is important.

2.6 ROC curves

The ROC curves were computed for the optimal decision tree and a logistic regression model. With the following principle to classify the test data:

$$\hat{Y} = \begin{cases} \text{yes} & \text{if } P(Y = \text{yes} \mid X) > \pi, \\ \text{no} & \text{otherwise,} \end{cases}$$

where

$$\pi = 0.05, 0.10, 0.15, \ldots, 0.90, 0.95.$$

For each threshold, the true positive rate and the false positive rate were calculated for both models. These values were used to plot the receiver operating characteristic (ROC) curve. In Figure 2.3, the two curves are similar. This indicates that both models have a similar ability to distinguish between the positive and negative classes.

In imbalanced datasets, ROC curves can be misleading because the false positive rate is dominated by the large number of negatives. Precision-recall curves focus on the minority (positive) class, making them a better choice to evaluate the model's ability to detect positives.

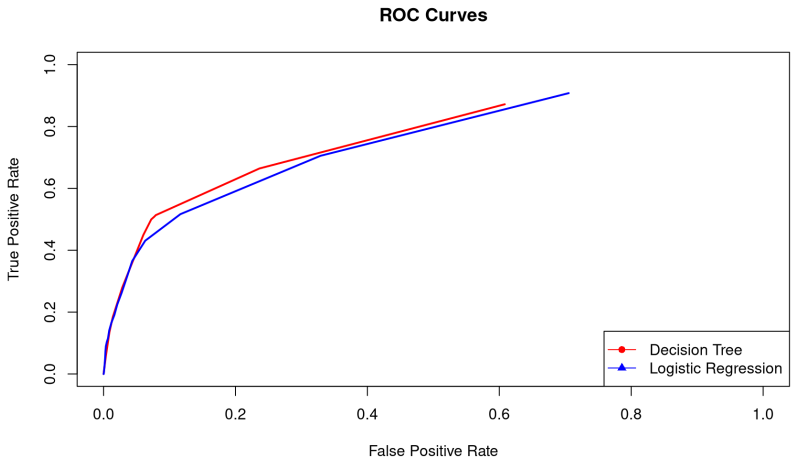


Figure 2.3: ROC curves.

3. Assignment 3. Principal components and implicit regularization

3.1 Task 1: Principal Component Analysis Using eigen()

3.1.1 Implementation

All variables except `ViolentCrimesPerPop` were centered and scaled using the `preProcess()` function from the `caret` package. Principal Component Analysis (PCA) was performed by computing the covariance matrix of the scaled data and applying eigenvalue decomposition using the `eigen()` function:

```
library(caret)
# names(communities) gives ViolentCrimesPerPop index 101
scaler = preProcess(communities[,-101], method = c("center", "scale"))
communities_scaled <- predict(scaler, communities[,-101])

#calculate the covariance matrix
cov <- cov(communities_scaled)
pca_results_1 <- eigen(cov) #solving equation C x u = lambda * u
```

The eigenvalues represent the variance explained by each principal component. The cumulative proportion of variance was computed to determine the number of components needed to capture at least 95% of the total variance.

3.1.2 Results

3.1.2.1 Components Required for 95% Variance

The number of components required for 95% variance was retreived and the result showed that 35 of the most contributing features was needed to contain 95% of the variance.

```
components_for_95_percent <- which(cumulative_variance >= 0.95)[1]
print(components_for_95_percent)
```

Result: Principal components are required to explain at least 95% of the variance in the data.

3.1.2.2 Variance Explained by First Two Components

```
variance_proportion[1]  # PC1
variance_proportion[2]  # PC2
```

The proportion of variance explained by each of the first two principal components is:

- **PC1:** 25.01699%
- **PC2:** 16.93597%
- **Combined (PC1 + PC2):** 41.95296%

3.1.3 Interpretation

The first principal component (PC1) captures the largest proportion of variance in the dataset, representing the direction of maximum variability. The second principal component (PC2) is orthogonal to PC1 and captures the next largest amount of variance. Together, the first two components explain approximately **41.95296%** of the total variance, indicating that a significant portion of the data's variability can be represented in a two-dimensional space.

The fact that 35 components are needed to reach 95% variance suggests that the dataset exhibits moderate to high dimensionality, with information distributed across multiple components rather than concentrated in just a few.

3.2 Task 2: PCA Analysis Using princomp()

3.2.1 Implementation

PCA was repeated using the `princomp()` function on the scaled data:

```
pca_results_2 <- princomp(communities_scaled)
```

3.2.2 Trace Plot of First Principal Component

Figure below shows the trace plot of the first principal component loadings. Each point represents the contribution (loading) of a feature to PC1.

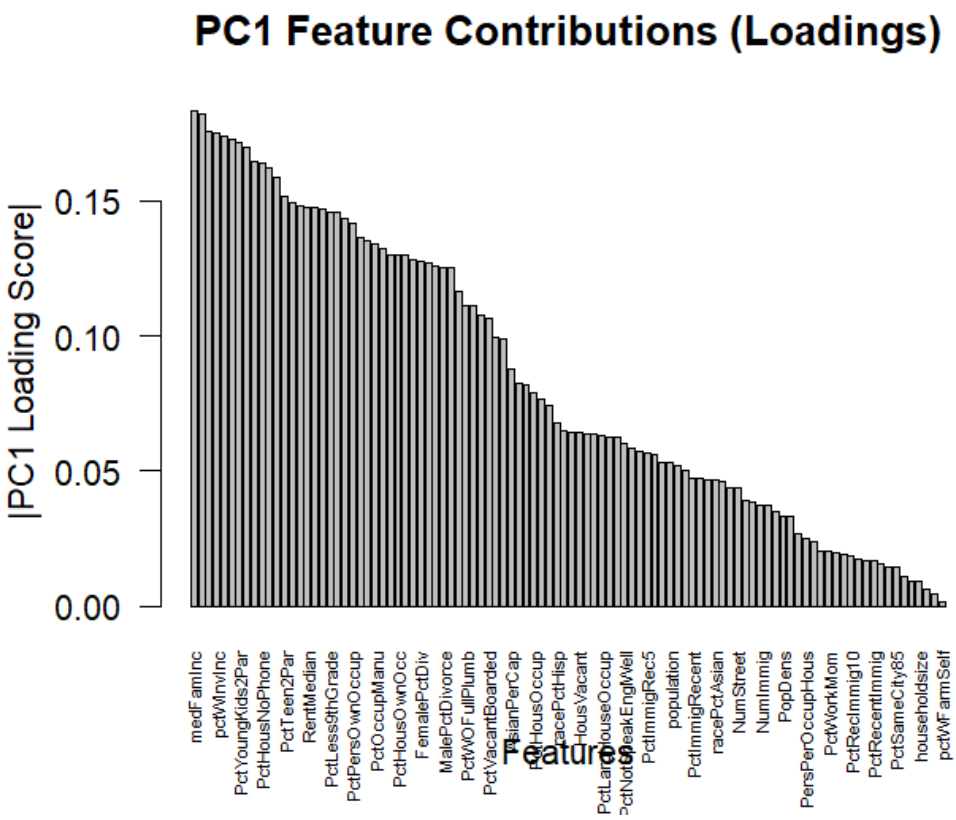


Figure 3.1: Trace plot of PC1 loading scores for all features.

From this result we can conclude that most features have relatively small loadings (close to zero). The loadings are fairly distributed with only a few features showing notable contributions in the range of 0.15-0.18 in absolute magnitude. No single feature dominates PC1.

3.2.3 Top 5 Contributing Features

The code below was run in order to list the top 5 most contributing features.

```
pc1_loadings <- pca_results_2$loadings[,1]
sorted_abs_loadings <- sort(abs(pc1_loadings), decreasing=TRUE)
top_5_features <- names(sorted_abs_loadings[1:5])
print(top_5_features)
print(sorted_abs_loadings[1:5])
```

The five features contributing most to the first principal component (by absolute value) are:

1. **medFamInc** (loading: **0.1833080**)
2. **medIncome** (loading: **0.1819830**)
3. **PctKids2Par** (loading: **0.1755423**)
4. **pctWInvInc** (loading: **0.1748683**)
5. **PctPopUnderPov** (loading: **0.1737978**)

3.2.4 Common Characteristics and Relationship to Crime

From this analysis we conclude that five features share a common theme: they all relate to **socioeconomic status** and **family structure**. The features can be grouped as:

- *Income indicators:* medFamInc, medIncome, and pctWInvInc all measure economic well-being
- *Poverty measure:* PctPopUnderPov directly indicates economic hardship
- *Family stability:* PctKids2Par reflects household structure

These features have a clear logical relationship to crime levels. It suggests that poverty, low income, and unstable family structures are associated with higher crime rates. Communities with lower median incomes and higher poverty rates often face economic stress, limited opportunities, and social challenges that correlate with increased violent crime. Similarly, family structure (single-parent vs. two-parent households) has could be linked to community stability. The fact that PC1 is dominated by socioeconomic factors suggests that economic and social conditions are the primary source of variation across communities in this dataset.

3.2.5 PC Scores Plot

The PC2 was plotted against PC1.

Figure 3.2 shows the PC scores in the (PC1, PC2) coordinate system, with points colored by `ViolentCrimesPerPop`.

3.2.6 Analysis of PC Scores Plot

The PC scores plot reveals several important patterns:

- **Clear gradient along PC1:** There is a visible color gradient from blue (low crime) to red (high crime) along the PC1 axis. Communities with negative PC1 scores tend to have lower crime rates (blue), while those with positive PC1 scores tend to have higher crime rates (red). This confirms that PC1 captures a primary dimension related to crime levels.
- **PC2 contribution:** The vertical spread (PC2 axis) shows less clear association with crime rate.

Overall, the plot demonstrates that PCA effectively captures meaningful variation in the data, with the first principal component strongly associated with crime levels through socioeconomic factors.

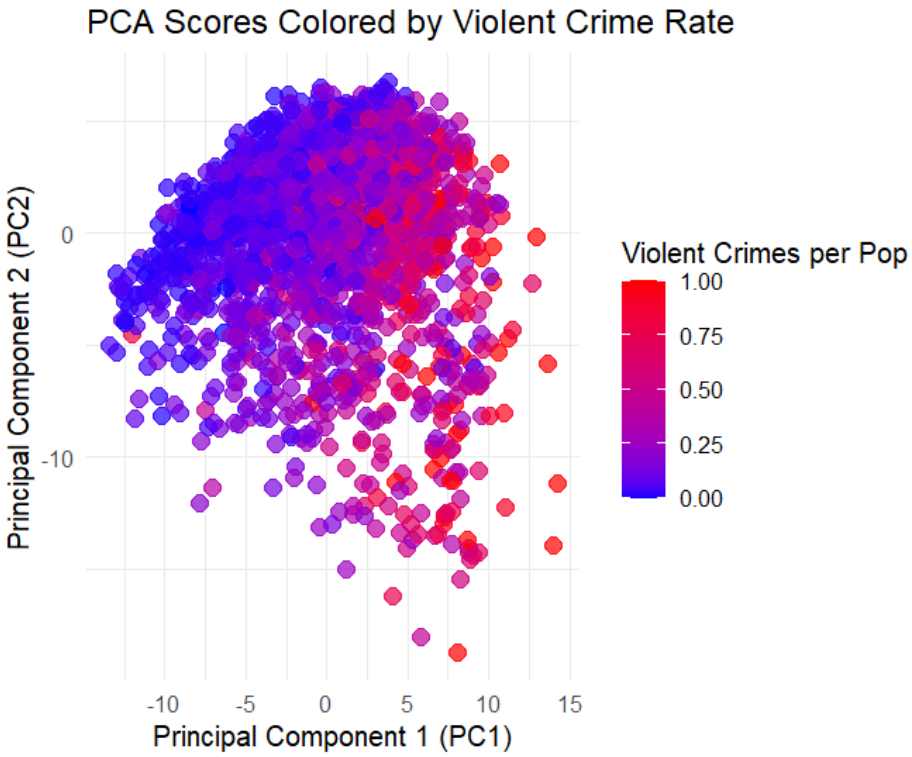


Figure 3.2: PCA scores plot colored by violent crime rate. Blue indicates low crime, red indicates high crime.

3.3 Task 3: Linear Regression Model

3.3.1 Implementation

The original dataset was split into training and test sets using a 50/50 split with a fixed random seed for reproducibility:

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

Both features and the response variable were scaled appropriately. The scaler was fitted on the training data and applied to both training and test sets to prevent data leakage:

```
scaler <- preProcess(train, method = c("center", "scale"))
```

```
train_scaled <- predict(scaler, train)
test_scaled <- predict(scaler, test)
```

A linear regression model was estimated using all features to predict `ViolentCrimesPerPop`:

```
model <- lm(ViolentCrimesPerPop ~ ., data = train_scaled)
preds_train <- predict(model, train_scaled)
preds_test <- predict(model, test_scaled)
```

3.3.2 Results

```
MSE_train <- mean((train_scaled$ViolentCrimesPerPop - preds_train)^2)
MSE_test <- mean((test_scaled$ViolentCrimesPerPop - preds_test)^2)
print(paste("Training MSE:", MSE_train))
print(paste("Test MSE:", MSE_test))
```

Table 3.1: Model performance on training and test sets

Metric	Value
Training MSE	**0.01406865**
Test MSE	**0.02171593**

3.3.3 Model Quality Assessment

The linear regression model demonstrates **good predictive performance** with relatively low errors on both training and test sets. The training MSE of **0.01406865** indicates that the model fits the training data well, while the test MSE of **0.02171593** shows reasonable generalization to unseen data. Since the training MSE is slightly larger than testing MSE it could be argued that the model is slightly overfitted.

3.3.3.1 Overall Quality

- **Low absolute errors:** Both MSE values are close to zero (for scaled data), indicating accurate predictions
- **Reasonable generalization:** The small gap between training and test errors suggests the model generalizes well

3.4 Task 4: BFGS Optimization

3.4.1 Implementation

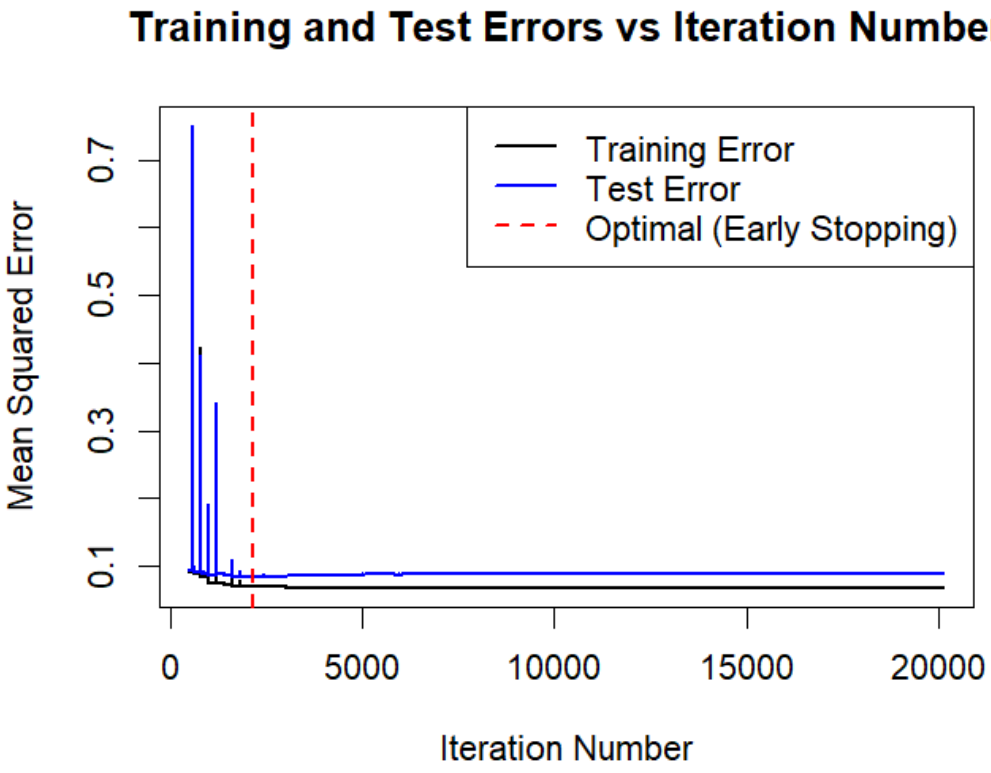
A cost function representing MSE for linear regression without intercept was implemented. The function computes predictions and returns training MSE while storing both training and test errors at each iteration:

```
 # Cost function that tracks errors at each iteration
cost_function <- function(theta) {
  # Training error (this is what BFGS minimizes)
  predictions_train <- X_train %*% theta
  mse_train <- mean((y_train - predictions_train)^2)

  # Test error
  predictions_test <- X_test %*% theta
  mse_test <- mean((y_test - predictions_test)^2)

  # Store both errors using global assignment
  train_errors <<- c(train_errors, mse_train)
  test_errors <<- c(test_errors, mse_test)

  # Return training error for optimization
  return(mse_train)
}
```

The BFGS algorithm was applied with initial parameter vector $\theta = 0$ using `optim()` without gradient specification.

3.4.2 Results

Figure below shows the error trajectories from iteration 500 onwards.



The optimal iteration according to early stopping is **2166**, where test error is minimized with training MSE = 0.01406865 and test MSE = 0.02171593.

3.4.3 Conclusions

Task 4 without intercept achieves similar performance to Task 3 with intercept (both have generalization gaps of 0.0076). However, early stopping at iteration 2166 is necessary to prevent overfitting, as training error continues decreasing while test error increases beyond this point. The intercept-free model requires careful iteration control, whereas Task 3's closed-form solution naturally avoids this issue.

4. Assignment 4. Theory

This chapter answers some questions using the course book *Machine Learning A First Course for Engineers and Scientists* (2022), written by Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten and Thomas B. Schön.

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

According to the book, there are two main ways to reduce the expected new data error. The first is to increase the size of the training data. This will typically decrease the generalization gap since the training dataset will include more variance. Making the model less flexible will reduce model complexity. This means that the model will be less prone to overfitting, typically increasing training error while reducing error on unseen data (pages 76-77).

4.2 What important aspect should be considered when selecting mini-batches, according to the book?

An important aspect to consider when selecting mini-batches is that they are balanced and representative of the entire dataset. A mini-batch containing only one class will provide a poor approximation of the gradient for the full dataset. When selecting mini-batches, it is important to first randomly shuffle the training data and then divide it into mini-batches. This shuffling process is repeated at the beginning of every new epoch (pages 125-126).

4.3 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

According to the book, one way to modify the data for imbalanced problems is to duplicate all positive training data points C times. This results in the model seeing positives more often, which increases their influence on the loss during training. Another approach is to modify the loss function itself so that misclassifying the positive class is C times more severe than misclassifying the negative class (pages 101-102).

A. Code

A.1 Full R code for assignment 1

```
library(dplyr)
library(glmnet)

# Load data
dataframe = read.csv("tecator.csv")

# Select data that is to be used
data = dataframe %>%
  select(Fat, Channel1:Channel100)

# Partition into 50% training data and 50% test data
n = dim(data)[1]
set.seed(12345)
id = sample(1:n, floor(n*0.5))
train_df = data[id,]
test_df = data[-id,]


# ---------------- Task 1: Linear regression ----------------
# Fit a linear regression to the training data
model = lm(Fat ~ ., data = train_df)
summary(model)

# Estimate the training and test errors
# Get predictions for training data and calculate MSE
predictions_train = predict(model, train_df)
MSE_train = mean((train_df$Fat - predictions_train)^2)

# Get predictions for test data and calculate MSE
predictions_test  = predict(model, test_df)
MSE_test = mean((test_df$Fat - predictions_test)^2)

print(paste("traning MSE = ", MSE_train))
print(paste("test MSE = ", MSE_test))


# ---------------- Task 3: Lasso regression ----------------
# Prepare training data, remove unnecessary columns
x = as.matrix(train_df %>% select(-Fat))
y = as.matrix(train_df %>% select(Fat))
```

```
# Fit a Lasso regression model with alpha = 1
model_lasso = glmnet(x, y, alpha = 1, family = 'gaussian')
model_lasso

# Plot the coefficients for Lasso regression
plot(model_lasso,
     xvar = "lambda",
     label = TRUE,
     main = "",
     ylab = "Coefficients")

title(main = "Lasso Regression",
      line = 2.5)



# ---------------- Task 4: Ridge regression ----------------
# Fit a Ridge regression model with alpha = 0
model_ridge = glmnet(x, y, alpha = 0, family = 'gaussian')
model_ridge

# Plot the coefficients for Ridge regression
plot(model_ridge,
     xvar = "lambda",
     label = TRUE,
     main = "",
     ylab = "Coefficients")

title(main = "Ridge Regression",
      line = 2.5)


# ---------------- Task 5: Cross-Validation ----------------
# Find optimal lambda with cross-validation. 10 folds by default.
cv_lasso = cv.glmnet(x, y, alpha = 1, family = 'gaussian')

# Optimal lambda and number of variables
coef(cv_lasso, s = "lambda.min")
cv_lasso$lambda.min

# Plot the cross-validation for Lasso regression
plot(cv_lasso)

title(main = "Lasso Regression cross-validation",
      line = 2.5)


# Prepare test data, remove unnecessary columns
x_test = as.matrix(test_df %>% select(-Fat))
```

```
y_test = as.matrix(test_df %>% select(Fat))

# Make predictions for minimum lambda value
predicted_test = predict(cv_lasso, newx = x_test, s = "lambda.min", type = "
    response")

plot(x = as.vector(y_test),
     y = as.vector(predicted_test),
     xlab = "Actual test values",
     ylab = "Predicted test values",
     xlim = c(min(y_test), max(y_test)),
     ylim = c(min(predicted_test), max(predicted_test)),
     col = "coral1",
     pch = 16,  # filled circles
     cex = 1.2,  # size of dots
     main = "Actual test vs Predicted test")

abline(a = 0, b= 1, col = "black", lty = 2)
```

A.2 Full R code for assignment 2

```
install.packages('readxl')
install.packages('kknn')
install.packages('tree')

library(readxl)
library(kknn)
library(tree)

#Each row in dataset is the image of a handwritten digit, each column
    represents the pixel intensity values and
#And the last column shows the actual digit 0-9.

#### Exxercise 1####
data = read.csv2("/home/arre/Universitet/TDDE01/Lab2/bank-full.csv", header =
    TRUE, stringsAsFactors = TRUE)

data$duration = c() #remove duration variable

#cols_factor <- c("job","marital","education","default","housing","loan","
    contact","month","poutcome","y") #make non numeric variables categorical
#data[, cols_factor] <- lapply(data[, cols_factor], factor)


colnames(data)[ncol(data)] <- "output" #give a label to last column, this is
    the target variable
```

```
n = dim(data)[1] #total number of instances
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,]


####Excercise 2 #####
missclass=function(X,X1){
  n=length(X)
  return(1-sum(diag(table(X,X1)))/n)
}

tree1 = tree(output~., data = train)
tree2 = tree(output~., data=train, minsize=7000)
tree3 = tree(output~., data=train, mindev=0.0005)

pred1_train <- predict(tree1, type="class")
pred2_train <- predict(tree2, type="class")
pred3_train <- predict(tree3, type="class")

# Validation predictions
pred1_valid <- predict(tree1, newdata = valid, type="class")
pred2_valid <- predict(tree2, newdata = valid, type="class")
pred3_valid <- predict(tree3, newdata = valid, type="class")

# Misclassification
mc1_train <- missclass(pred1_train, train$output) # = 0.1048441
mc1_valid <- missclass(pred1_valid, valid$output) # = 0.1092679

mc2_train <- missclass(pred2_train, train$output) #=0.1048441
mc2_valid <- missclass(pred2_valid, valid$output) #=0.1092679

mc3_train <- missclass(pred3_train, train$output) #0.09400575
mc3_valid <- missclass(pred3_valid, valid$output) #0.1119221


####Exercise 3#####


#code from lecture 2b
deviance_train=rep(0, 50)
deviance_valid=rep(0, 50)
```

```
for(k in 2:50){
 pruned_tree = prune.tree(tree3, best=k)
 pred=predict(pruned_tree, newdata = valid, type="tree")
 deviance_train[k] = deviance(pruned_tree)
 deviance_valid[k] = deviance(pred)
}


plot(2:50, deviance_train[2:50], type="b", col="orange",
     xlab="Number of leaves", ylab="Deviance",
     main="Training vs Validation Deviance", ylim = range(7000, 12000))

lines(2:50, deviance_valid[2:50], type="b", col="blue")
legend("topright", legend=c("Training", "Validation"),
       col=c("orange", "blue"), pch=16)

optimal_leaves = which.min(deviance_valid[-1]) + 1 #21 leaves, gives 22
    optimal leaves because of index 2:50 in loop.

optimal_tree=prune.tree(tree3, best = optimal_leaves)
plot(optimal_tree)
text(optimal_tree, pretty=0)
summary(optimal_tree)
#missclass rate 0.1039

######Exercise 4 ########

predicted_tree = predict(optimal_tree, newdata = test, type = "class")
confusion_matrix = table(test$output, predicted_tree)

accuracy=1-missclass(predicted_tree, test$output) #accuracy = 0.8910351

precision = confusion_matrix[2,2]/(confusion_matrix[2,2] + confusion_matrix
    [1,2]) #TP/(TP+FP)= 0.66667
recall = confusion_matrix[2,2]/(confusion_matrix[2,2] + confusion_matrix[2,1])
     #TP/(TP+FN) = 0.135

F1 = 2* (precision*recall)/(precision+recall) #F1 = 0,224554 quite low => bad
    at detecting the positive class


  #####Exercise 5 ####### to be finished
#Loss matrix needs to be swapped from the one given in assigment correct order
     no, yes
LossMatrix = matrix(c(0, 1, 5, 0), nrow = 2, ncol = 2, byrow = TRUE)
colnames(LossMatrix) = rownames(LossMatrix) = levels(test$output)

# multiply with loss matrix to weigh the results as we need to
testPrediction = predict(optimal_tree, newdata = test) %*% LossMatrix
```

```
# Code from tutorial 2
testPredictionI = apply(testPrediction, MARGIN=1, FUN = which.min)
Pred = levels(test$output)[testPredictionI]

confusion_matrix = table(test$output, Pred)

accuracy=1-missclass(Pred, test$output) #accuracy = 0.8910351

precision = confusion_matrix[2,2]/(confusion_matrix[2,2] + confusion_matrix
    [1,2])
recall = confusion_matrix[2,2]/(confusion_matrix[2,2] + confusion_matrix[2,1])

F1 = 2* (precision*recall)/(precision+recall) #F1 = 0,486205 increased =>
    model is better at detecting the positive class.


  ##### Execise 6 ######

optimal_dt_predict = predict(optimal_tree, newdata = test, type="vector")

logistic_regression = glm(output~., train, family = "binomial")
prob_logistic_regression = predict(logistic_regression, newdata=test, type="
    response")

TPR_DT = numeric(20)
FPR_DT = numeric(20)

TPR_LR = numeric(20)
FPR_LR = numeric(20)
index = 1
for (i in seq(0.05, 0.95, 0.05)) {
  #Decision tree
  y_hat = ifelse(optimal_dt_predict[,2] > i, "yes", "no")
  cm_DT = table(
    factor(test$output, levels = c("no", "yes")),
    factor(y_hat, levels = c("no", "yes"))
  )


  FPR_DT[index] = cm_DT[1,2]/(cm_DT[1,2]+ cm_DT[1, 1]) #FP/(FP+TN)
  TPR_DT[index] = cm_DT[2,2]/(cm_DT[2,2]+ cm_DT[2, 1]) #TP/(TP+FN)


  #Logistic regression
  pred_LR = ifelse(prob_logistic_regression > i, "yes", "no")
  cm_LR = table(
    factor(test$output, levels=c("no","yes")),
    factor(pred_LR, levels=c("no","yes"))
  )
```

```
  FPR_LR[index] = cm_LR[1,2]/(cm_LR[1,2]+ cm_LR[1, 1]) #FP/(FP+)
  TPR_LR[index] = cm_LR[2,2]/(cm_LR[2,2]+ cm_LR[2, 1]) #TP/(TP+FN)

  index = index + 1

}

# Plot Decision Tree
plot(FPR_DT, TPR_DT, type="l", col="red", lwd=2,
     xlab="False Positive Rate", ylab="True Positive Rate",
     xlim=c(0,1), ylim=c(0,1),
     main="ROC Curves")

# Add Logistic Regression
lines(FPR_LR, TPR_LR, col="blue", lwd=2)


# Legend
legend("bottomright", legend=c("Decision Tree","Logistic Regression"),
       col=c("red","blue"), lty=1, pch=c(16,17))
```

A.3 Full R code for assignment 3

```
communities = read.csv("communities.csv", header = TRUE)

#Question 1

#extract the crimerate
crime_rate <- communities[,101]


library(caret)
# names(communities) gives ViolentCrimesPerPop index 101
scaler = preProcess(communities[,-101], method = c("center", "scale"))
communities_scaled <- predict(scaler, communities[,-101])

#calculate the covariance matrix
cov <- cov(communities_scaled)
pca_results_1 <- eigen(cov) #solving equation C x u = lambda * u
eigen_values <- pca_results_1$values

total_variance <- sum(eigen_values) #100 after scaling

variance_proportion <- eigen_values/total_variance

cumulative_variance <- cumsum(eigen_values/total_variance)

#Amount of components needed for 95% of the variance
```

```
components_for_95_percent <- which(cumulative_variance >= 0.95)[1]

#The proportion of PC1
variance_proportion[1]

#The proportion of PC2
variance_proportion[2]

#Question 2 (Oral defence)

#Trace plot of loading scores
pca_results_2 = princomp(communities_scaled)
loading_scores_sorted <- sort(pca_results_2$loadings, decreasing=TRUE)
plot(pca_results_2$loadings[,1], main="Trace Plot of PC1 Loading scores", ylab
    ="Loading Score")



sorted_pc1_loadings <- sort(pca_results_2$loadings[,1], decreasing = TRUE)
barplot(sorted_pc1_loadings,
        main = "PC1 Feature Contributions (Loadings)",
        xlab = "Features",
        ylab = "PC1 Loading Score",
        las = 2,       # rotates x-axis labels vertically
        cex.names = 0.5)

sorted_pc1_loadings <- sort(abs(pca_results_2$loadings[,1]), decreasing = TRUE
    )
barplot(sorted_pc1_loadings,
        main = "PC1 Feature Contributions (Loadings)",
        xlab = "Features",
        ylab = "|PC1 Loading Score|",
        las = 2,       # rotates x-axis labels vertically
        cex.names = 0.5)

#No not many varibles have a notable contribution or dominates PC1,
#the greater values are around 0.15 and 0.18 in absolute magnitude


#set some values (loading, absolute loading, sort them etc)
pc1_loadings <- pca_results_2$loadings[,1]
abs_pc1_loadings <- abs(pc1_loadings)
sorted_pc1_loadings <- sort(abs_pc1_loadings, decreasing=TRUE)

top_5_contributing_features = sorted_pc1_loadings[1:5]
print(top_5_contributing_features)
#greatest contributing features are: medFamInc, medIncome PctKids2Par
    pctWInvInc, PctPopUnderPov
```

```r
# medFamInc (Median Family Income)
# medIncome (Median Income)
# PctKids2Par (Percentage of Kids with Two Parents)
# pctWInvInc (Percentage with Investment Income)
# PctPopUnderPov (Percentage of Population Under Poverty)
# one could easily make the case that these features all are correlated to the
    crimerate.


library(ggplot2)

scores_df <- data.frame(
  PC1 = pca_results_2$scores[,1],
  PC2 = pca_results_2$scores[,2],
  crime_rate = crime_rate
)

ggplot(scores_df, aes(x=PC1, y=PC2, color=crime_rate))  +
         geom_point(alpha=0.7, size=3) +
         scale_color_gradient(low="blue", high="red")+
         labs(title="PCA Scores Colored by Violent Crime Rate",
              x = "Principal Component 1 (PC1)",
              y = "Principal Component 2 (PC2)",
              color = "Violent Crimes per Pop") +
         theme_minimal()

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

model_scaler = preProcess(train, method = c("center", "scale"))
train_scaled <- predict(scaler, train)
test_scaled <- predict(scaler, test)

model <- lm(train_scaled$ViolentCrimesPerPop~., train_scaled)
preds_train <- predict(model, train_scaled)
preds_test <- predict(model, test_scaled)

MSE_train <- mean((train_scaled$ViolentCrimesPerPop - preds_train)^2)
MSE_test <- mean((test_scaled$ViolentCrimesPerPop - preds_test)^2)
# MSE_train 0.01406865
# MSE_test 0.02171593
# Slightly overfitted


#TASK 4
```

```r
# Prepare the data matrices
X_train <- as.matrix(train_scaled[, -which(names(train_scaled) == "
    ViolentCrimesPerPop")])
X_test <- as.matrix(test_scaled[, -which(names(test_scaled) == "
    ViolentCrimesPerPop")])
y_train <- train_scaled$ViolentCrimesPerPop
y_test <- test_scaled$ViolentCrimesPerPop

# Initialize storage vectors
train_errors <- c()
test_errors <- c()

# Cost function that tracks errors at each iteration
cost_function <- function(theta) {
  # Training error (this is what BFGS minimizes)
  predictions_train <- X_train %*% theta
  mse_train <- mean((y_train - predictions_train)^2)

  # Test error
  predictions_test <- X_test %*% theta
  mse_test <- mean((y_test - predictions_test)^2)

  # Store both errors using global assignment
  train_errors <<- c(train_errors, mse_train)
  test_errors <<- c(test_errors, mse_test)

  # Return training error for optimization
  return(mse_train)
}

# Initialize theta to zero vector
theta_init <- rep(0, ncol(X_train))

# Run BFGS optimization
opt_result <- optim(
  par = theta_init,
  fn = cost_function,
  method = "BFGS"
)

# Find optimal iteration (minimum test error - early stopping criterion)
optimal_iteration <- which.min(test_errors)

cat("Total iterations:", length(train_errors), "\n")
cat("Optimal iteration (early stopping):", 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")

# Plot errors (discarding first 500 iterations as suggested)
```

```
start_iter <- 500
plot_data <- data.frame(
  iteration = start_iter:length(train_errors),
  train_error = train_errors[start_iter:length(train_errors)],
  test_error = test_errors[start_iter:length(test_errors)]
)

plot(plot_data$iteration, plot_data$train_error,
     type = "l", col = "black", lwd = 2,
     xlab = "Iteration Number",
     ylab = "Mean Squared Error",
     main = "Training and Test Errors vs Iteration Number",
     ylim = range(c(plot_data$train_error, plot_data$test_error)))

lines(plot_data$iteration, plot_data$test_error, col = "blue", lwd = 2)

# Add vertical line at optimal iteration
abline(v = optimal_iteration, col = "red", lty = 2, lwd = 2)

legend("topright",
       legend = c("Training Error", "Test Error", "Optimal (Early Stopping)"),
       col = c("black", "blue", "red"),
       lty = c(1, 1, 2),
       lwd = 2)

# Compare with Task 3 results
cat("\n=== Comparison with Task 3 ===\n")
cat("Task 3 - Training MSE:", mse_train, "\n")
cat("Task 3 - Test MSE:", mse_test, "\n")
cat("Task 4 (optimal) - Training MSE:", train_errors[optimal_iteration], "\n")
cat("Task 4 (optimal) - Test MSE:", test_errors[optimal_iteration], "\n")

# Alternative visualization with ggplot2
library(ggplot2)

plot_data_long <- data.frame(
  iteration = rep(start_iter:length(train_errors), 2),
  error = c(train_errors[start_iter:length(train_errors)],
            test_errors[start_iter:length(test_er rors)]),
  type = rep(c("Training", "Test"), each = length(start_iter:length(train_
      errors)))
)

ggplot(plot_data_long, aes(x = iteration, y = error, color = type)) +
  geom_line(size = 1) +
  geom_vline(xintercept = optimal_iteration, linetype = "dashed",
             color = "red", size = 1) +
  labs(title = "Training and Test Errors During BFGS Optimization",
       x = "Iteration Number",
       y = "Mean Squared Error",
```

```
      color = "Dataset") +
  theme_minimal() +
  annotate("text", x = optimal_iteration, y = max(plot_data_long$error) * 0.9,
           label = paste("Optimal:", optimal_iteration),
           color = "red", hjust = -0.1)
```