Computer lab 2, group 29

Machine Learning, TDDE01

Oscar Hoffmann, oscho091
Martin Sörme, masor844
Hjalmar Öhman, hjaoh082

1. Statement of contribution

Martin Sörme was responsible for assignment 1. In the second assignment, Hjalmar Öhman was responsible and Oscar Hoffmann was responsible for assignment 3. For each assignment this included the code writing and analysis. However, all members discussed the problems and took help from the other members in the tasks during the coding. After finishing the code writing and analysis, all three members sat down and discussed the problems and presented the solutions. This was done to ensure that all members understood the code, solutions and analyses for the tasks.

2. Assignments

2.1 Explicit regularization

2.1.1 TASK 1

Underlying probabilistic model:

Coefficients:

	Estimate	Std. Error	t value	Pr(>|t|)	
(Intercept)	-1.815e+01	5.488e+00	-3.306	0.01628	*
Channel1	2.653e+04	1.126e+04	2.357	0.05649	.
Channel2	-5.871e+04	3.493e+04	-1.681	0.14385	
Channel3	1.154e+05	7.373e+04	1.565	0.16852	
Channel4	-2.432e+05	1.175e+05	-2.070	0.08387	.
Channel5	3.026e+05	1.193e+05	2.536	0.04430	*
Channel6	-2.365e+05	8.160e+04	-2.898	0.02741	*
Channel7	1.090e+05	3.169e+04	3.440	0.01380	*
Channel8	-6.054e+04	1.508e+04	-4.015	0.00700	**
…					
Channel100	-1.206e+04	4.264e+03	-2.828	0.03006	**

Misclassification rate train: 0.00570911701090834
Misclassification rate test: 722.429419336971

An analysis based on this misclassification rate is that there seems to be overfitting of the training data, leading to a high misclassification rate for the test data. The overall quality of the model is quite low since it has a high misclassification rate for the testing data.

2.1.2 TASK 2

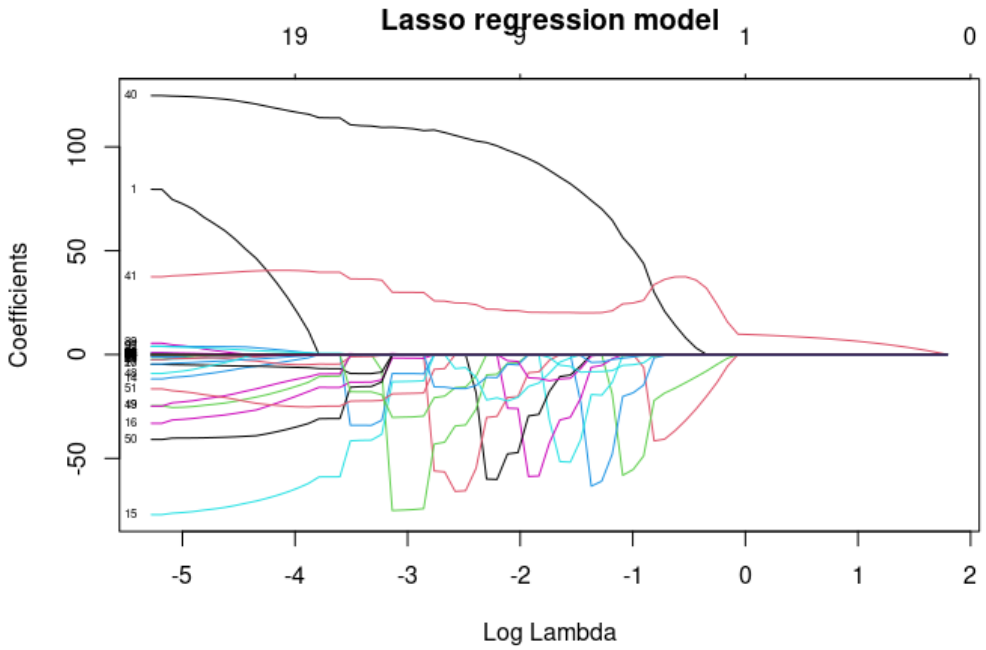
The cost function that should be optimized in this scenario is the following:

$$cost = argmin\{\frac{1}{n}\sum_{i=1}^{n}(Fat_i - \theta_0 - \theta_1 x_{1i} - \ \dots \ - \theta_{100i} x_{100i})^2 + \lambda \sum_{j=1}^{p} abs|\theta_j|\}$$

All of the channels are used in this function, so the channels 2-99 are also included, but not shown to simplify the visualization of the model.

2.1.3 TASK 3

```
################################### TASK 3 ###################################

## LASSO regression

x = as.matrix(train_data %>% select(-Fat))
y = as.matrix(train_data %>% select(Fat))

lasso = glmnet(x, y, alpha = 1, family = "gaussian")
plot(lasso, xvar = "lambda", label = TRUE, main="Lasso regression model")
```



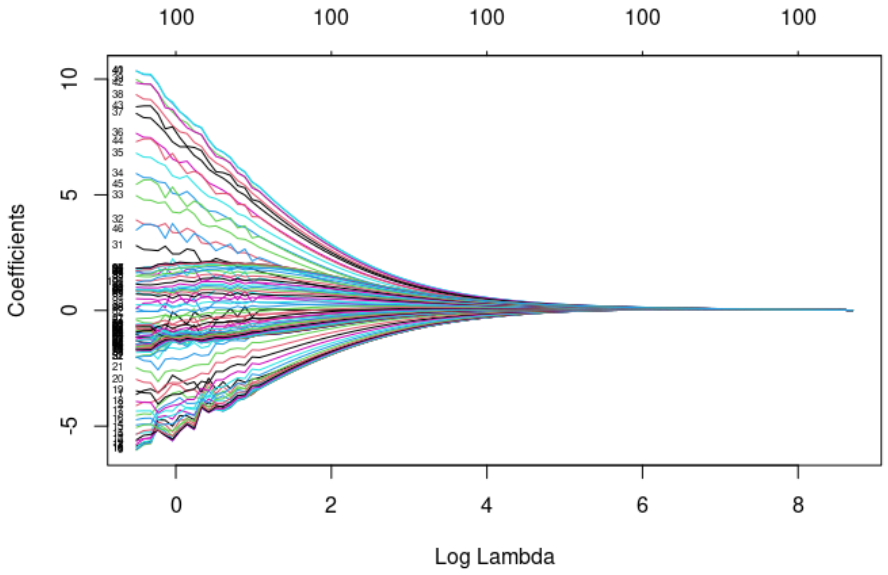
We can see from this plot that as log(λ) increases, the number of coefficients used decreases. This is due to the penalty becoming larger and therefore forcing coefficients to be zero.

What value of the penalty factor can be chosen if we want to select a model with only three features?

When interpreting the plot, we can observe that if we only want three features, we could implement a log lambda of about -0.3 (Ahead of when coefficient 40 is 0 but before when the red and green line turns to 0). Implementing this lambda would lead to only three features being used in the Lasso model.

2.1.4 TASK 4

```
##################################### TASK 4 #####################################

## Ridge regression

ridge = glmnet(x, y, alpha = 0, family="gaussian")
plot(ridge, xvar = "lambda", label = TRUE, main="Ridge regression")
```



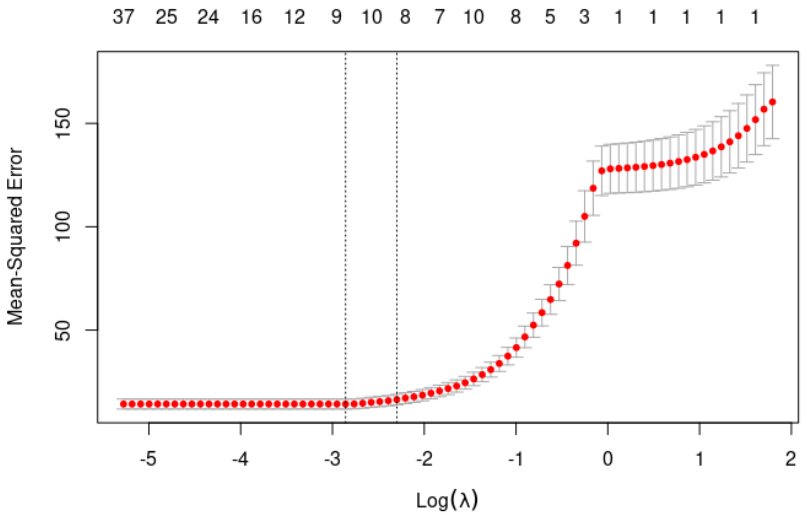
A conclusion that could be drawn from this is that Ridge Regression with L2 Regularization shrinks the coefficients towards zero when the log lambda increases but it is never exactly zero whereas the Lasso Regression with L1 Regularization forces the coefficients to exactly zero. Lasso effectively does feature selection, whereas Ridge just shrinks some parameters. Ridge regression is also harder to interpret, since it retains all of the features. Lasso on the other hand uses feature selection making it more interpretable by highlighting the important features as determined by the model.

2.1.5 TASK 5

Compute optimal LASSO model

```
cross_validation = cv.glmnet(x, y, alpha=1, family="gaussian")
opt_lambda = cross_validation$lamda.min
```

Plot showing dependence of the CV score on log(λ)



The cross-validation score seems to increase as log(λ) increases. It increases heavily from between -2 to 1 and then seems to stagnate.

Optimal λ and variables chosen

The optimal lambda is: 0.05733535

```
coef(cross_validation, s="lambda.min")
```

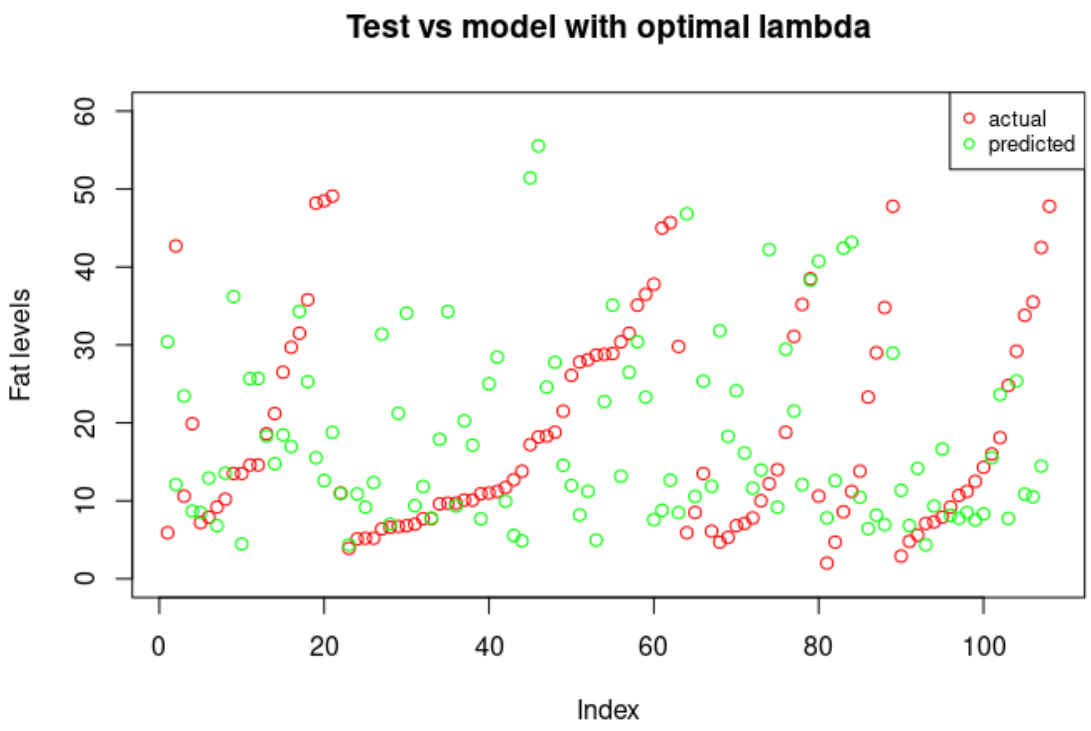
From this, we can see the channels used in the model, these are channels 52, 51, 41, 50, 16, 15, 14, 13 are chosen. 8 in total.

Does the information displayed in the plot suggest that the optimal λ value results in a statistically significantly better prediction than log(λ) = -4?

No, based on the graph it does not suggest that a value of -4 is significantly better since it seems that the mean-squared error for both is roughly the same. The change in mean-squared error can not be visually observed to be different.

Create a scatter plot of the original test versus predicted values for the model corresponding to optimal lambda and comment on whether the model predictions are good

```
# Cross validation prediction
crossval_predict = predict(cross_validation, newx = x, s = opt_lambda)

plot(test_data$Fat, col="red", ylim = c(0,60), ylab="Fat levels", main = "Test vs model with optimal lambda")
points(crossval_predict, col = "green")
legend("topright", c("actual", "predicted"), col = c("red", "green"), pch = 1, cex = 0.8)
```



The predictions from the model seem to have quite a high degree of error. There is no clear pattern where the model successfully predicts the actual fat values and it seems to be quite random or sporadic. However, it is with mentioning that some of the values are predicted quite accurately, but there is no clear way of seeing a pattern of correct predictions based on this graph.

2.2 Decision trees and logistic regression for bank marketing

2.2.1 TASK 1

Split the data according to instructions, as done in lecture slides.

2.2.2 TASK 2

Report the misclassification rates for the training and validation data.

Training
default: 0.1048
minsize: 0.1048
mindev: 0.0936
Validation
default: 0.1092
minsize: 0.1092
mindev: 0.1118

Which model is the best one among these three?

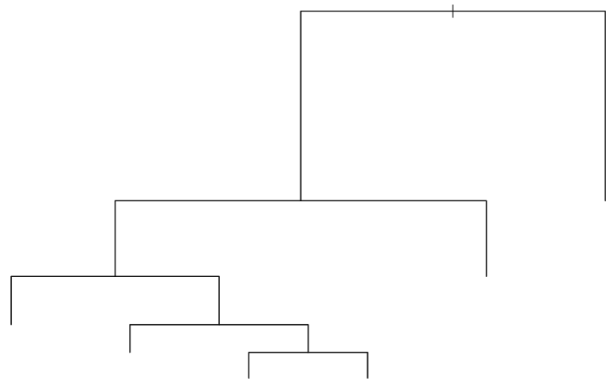
Default and node gets the same missclass on unseen data (valid), so equally good. mindev performs the worst because it is overfitted.

Report how changing the deviance and node size affected the size of the trees and explain why.

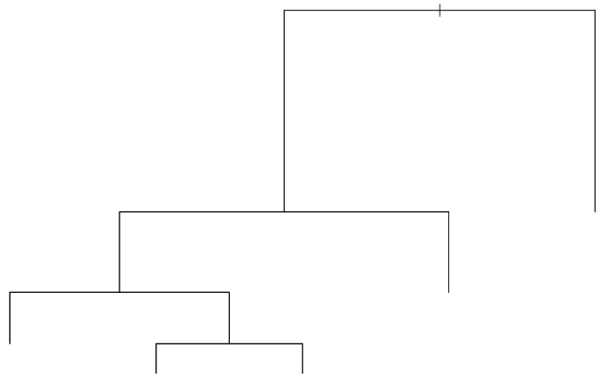
Restriction of each node containing a minimum amount of 7000 data points (default 10) made the tree smaller, since one branching was therefore not allowed to be done.

Restriction of each branching needing a mindev of 0.0005 (default 0.01) to occur led to a very large overfitted tree because lots of branching could be done.

Default



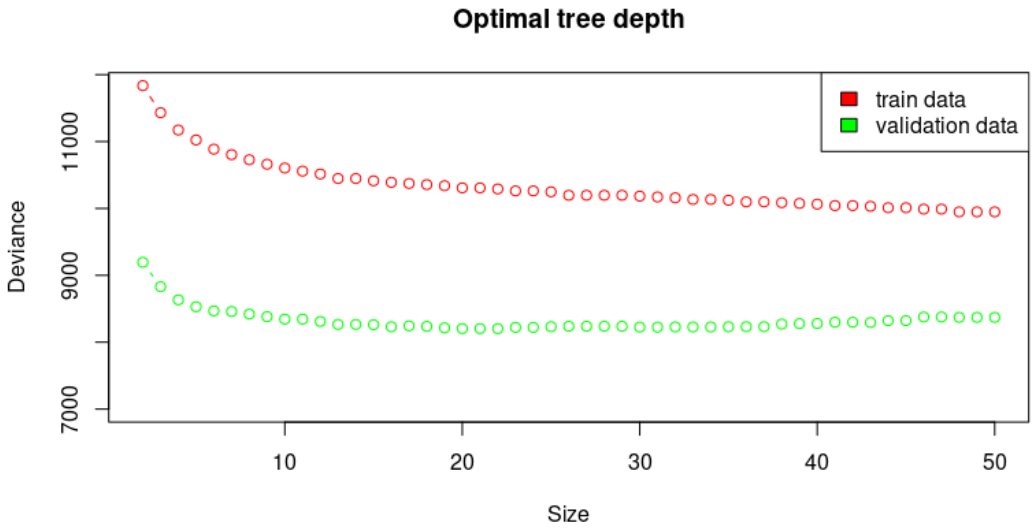
minsize



mindev

2.2.3 TASK 3

Present a graph of the dependence of deviances for the training and the validation data on the number of leaves



Interpret this graph in terms of bias-variance tradeoff.

It performs better on unseen data (validation lower deviance), indicating high bias and low variance.

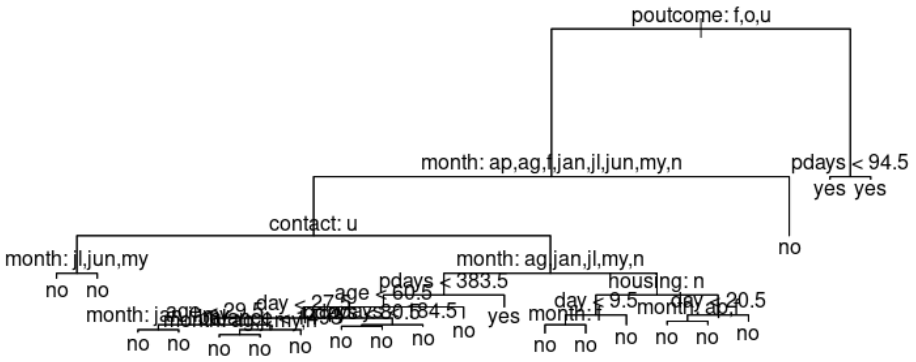
Train's deviance decreases with more leaves, indicating high bias and low variance.

High Variance (Overfitting): The model is too complex, fitting the training data too closely and not generalizing well to new data.

High Bias (Underfitting): The model is too simple and does not capture the underlying patterns in the data.

Report the optimal amount of leaves and which variables seem to be most important for decision making in this tree.

Optimal leaves are 47 for training data and 21 for validation data.



poutcome and *month* are variables most important, since they are branched on first. The other 2nd layer branching to the right covered in text is also poutcome.

Interpret the information provided by the tree structure (not everything but most important findings).

poutcome not being *f*, *o*, or *u* important to predict *y*. Same for *month* not being *april, august, february, january, july, june, may* or *november*.

2.2.4 TASK 4

Estimate the confusion matrix, accuracy and F1 score.

test_predictions		
	no	yes
no	11812	167
yes	1294	291

Accuracy: 0.8922884
F1: 0.2848752

Comment whether the model has a good predictive power and which of the measures (accuracy or F1-score) should be preferred here.

Googling gives “an F1 score of **0.7 or higher** is often considered good”, so 0.304 indicates a bad predictive power.
F1-score is preferred here, since the data is heavily imbalanced.The amount of “no” is a lot higher. Just guessing “no” gives good accuracy, since it's more frequently occurring than “yes”, which the model seems to do.

2.2.5 TASK 5

Report the confusion matrix for the test data.

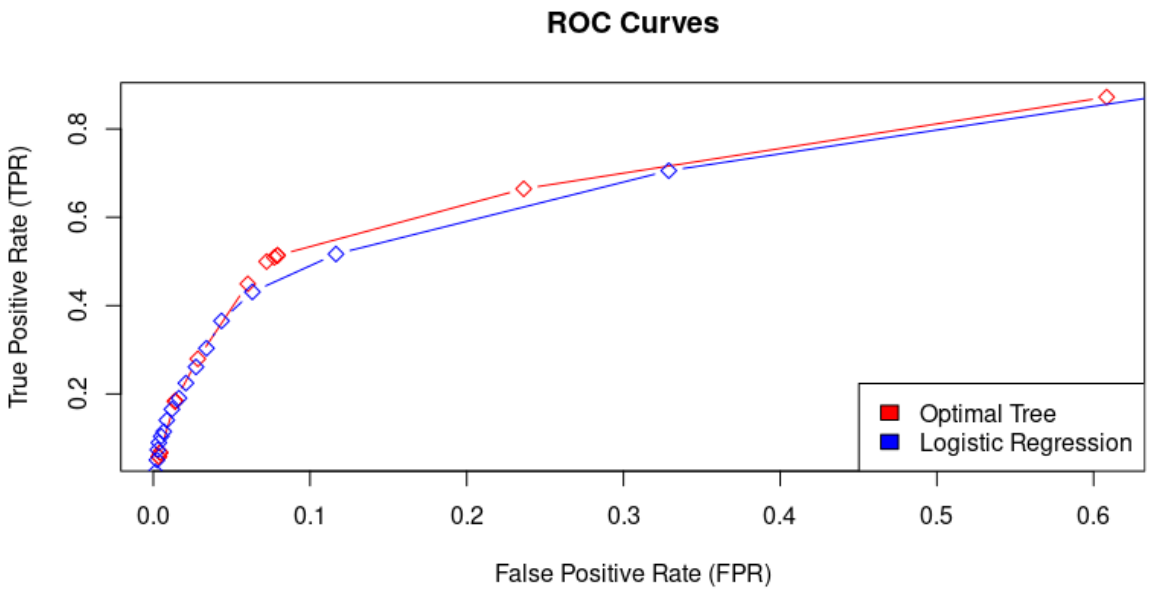
	predictions_with_loss	
	no	yes
no	11030	949
yes	771	814

Compare the results with the results from step 4 and discuss how the rates has changed and why.

About 5 times as many yes predictions.
F1 Score: 0.4862605 (vs 0.2848752)
Accuracy: 0.8731937 (vs 0.8922884)
Lower accuracy, since predicting “no” is not as safe of a bet as earlier.
Higher F1 Score, because recall rate significantly increased (0.51 vs 0.18) while precision decreased by a lower amount (0.46 vs 0.64). *f1_score <- (2 * precision * recall_rate) / (precision + recall_rate)*

2.2.6 TASK 6

Compute the TPR and FPR values for the two models and plot the corresponding ROC curves. Conclusion?



AUC for Optimal Tree is slightly larger, so therefore performing slightly better. TPR and FPR do not take into account imbalanced classes.

Why precision recall curve could be a better option here?
Precision recall curve would account for imbalance class, therefore still be an option.

2.3 Principal components and implicit regularization

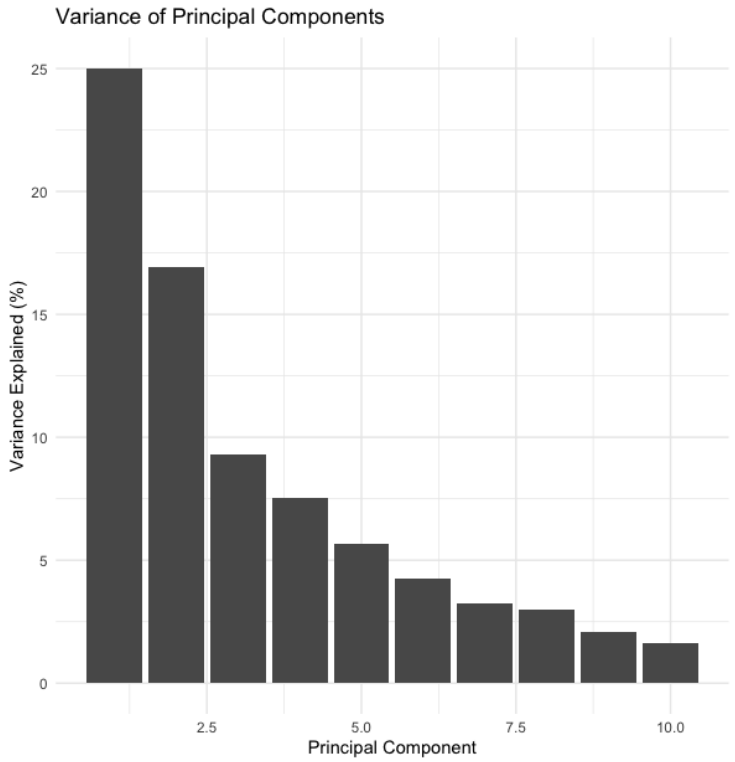
2.3.1 TASK 1

```
############################## Task 1 ##########################################
#Scale the data and manually implement PCA

#Reading the dataset and excluding the target variable ViolentCrimesPerPop for PCA
communities_data = read.csv("communities.csv")
VCPP_excluded = communities_data[,1:100]

#Scaling the data
scaler = preProcess(VCPP_excluded)
data_scaled = predict(scaler, VCPP_excluded)

#Calculating the covariance matrix of the scaled data and computing the eigenvalues
cov_matrix = cov(data_scaled)
eigen_values = eigen(cov_matrix) #The values vector is sorted in decreasing order

#Calculating the cumulative proportion of variance explained by each PC
#Using the match() function to find how many components are needed to obtain 95% of the va
cumulative_variance = cumsum(eigen_values$values)
match(TRUE, cumulative_variance >= 95) #35 PCs
print(cumulative_variance) #printing the cumulative_variance vector to verify the step abo
print(eigen_values$values[1]) #25.01699
print(eigen_values$values[2]) #16.93597

#Plotting the variance of the 10 first componentents in a barplot for visualization purpos
PCA = data.frame(PCA = 1:10, Variance = (eigen_values$values)[1:10])
ggplot(PCA, aes(x = PCA, y = Variance)) + geom_bar(stat = "identity") + labs(title = "Vari
```

The first task was to import the data from the “communities.csv” file and scale all variables except the target variable “ViolentCrimesPerPop”. This was done as in the lecture slides. After that we implemented PCA manually by calculating the covariance matrix and the eigenvalues. The eigenvalues express the amount of variance captured by each principal component and the eigenvector indicates the direction of each principal component since the elements in the vector represent the coefficients of the linear combination. In PCA, we often look for the largest eigenvalues and their corresponding eigenvectors because they represent the directions where the data has the most variance.

Report how many components are needed to obtain at least 95% of variance in the data.
To do this the eigenvalues of each principal component were cumulatively added and saved in the vector “cumulative_variance” using the cumsum() function. The match() function was then used with the logical condition in the code to find the index of the element where the sum was >= 95. This calculation showed that 35 components were needed to obtain at least 95% of the variance in the data. To verify that the match() function worked correctly the cumulative_variance vector was also printed and the number of components were counted manually.

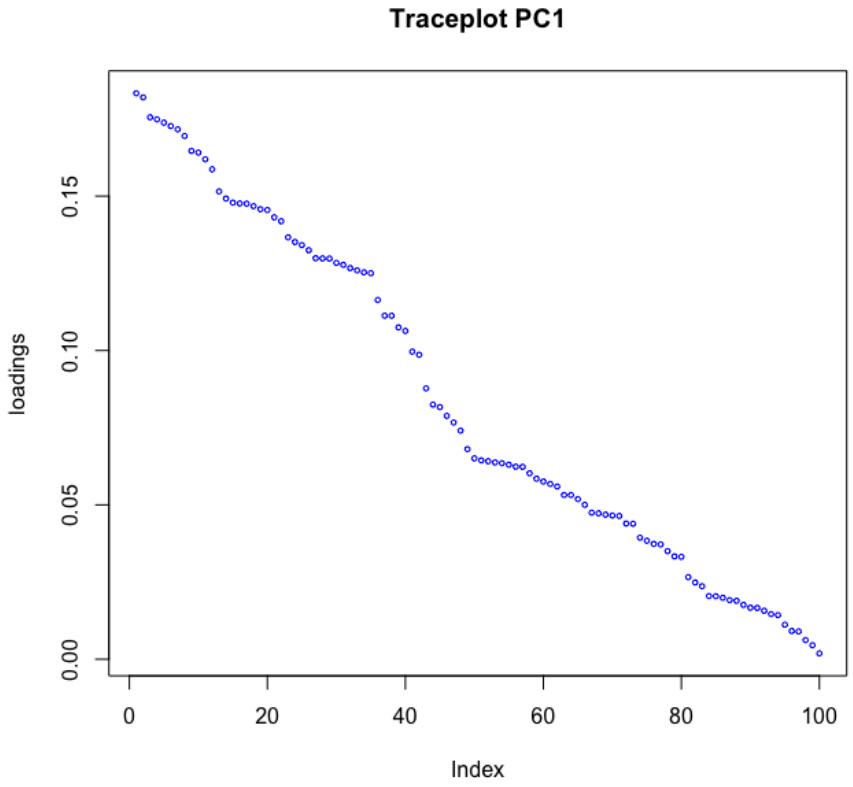
What is the proportion of variation explained by each of the first two principal components?
To answer this we simply had to print the first eigenvalues. This resulted in 25.01699 ~ 25% for the first principal component and 16.93597 ~17% for the second principal component. To visualize this in a nice way we also created a bar plot that shows the percentage of variance contained in the 10 first principal components which can be seen below.



2.3.2 TASK 2

```
############################## Task 2 ##########################################
#Repeat PCA using princomp(), make trace plot of the first principal component

PCA_res = princomp(data_scaled)
#Sorting the absolute values of the first PC's loadings in descending order
PC1_sorted = sort(abs(PCA_res$loadings[,1]), decreasing = TRUE)
plot(PC1_sorted, main="Traceplot PC1", col = "blue", cex = 0.5, ylab = "loadings")
print(head(PC1_sorted, 5)) #The 5 features that contribute mostly

#Plotting the PC scores
Violent_Crimes = communities_data$ViolentCrimesPerPop
ggplot(data.frame(PC1 = PCA_res$scores[,1], PC2 = PCA_res$scores[,2]), aes(x=PC1, y=PC2)) + geom_point(aes(co
```

In the second task we repeated the PCA analysis with the princomp() function as instructed. We then made a traceplot by first sorting the absolute values of the loadings for PC1 in decreasing order and then using the plot() function. The traceplot can be seen below.



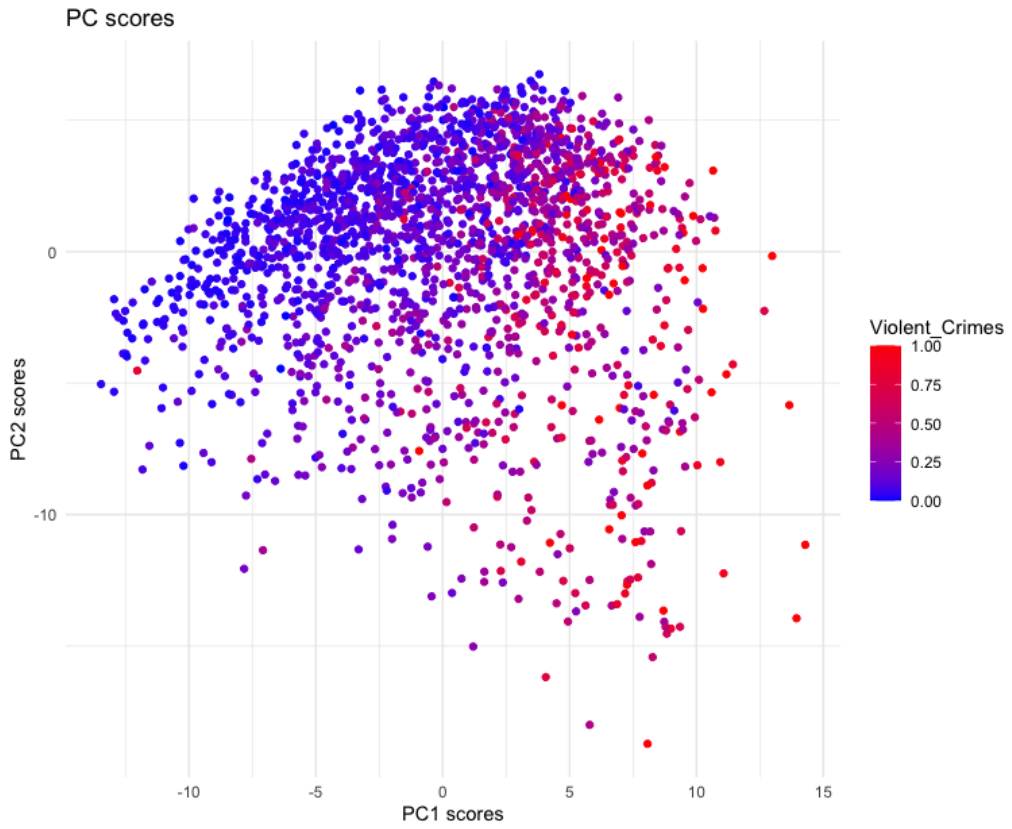
Do many features have a notable contribution to this component?
The plot clearly shows that many variables have a notable contribution to PC1. This is because many have a high absolute value meaning that they strongly influence PC1.

Report which 5 features contribute mostly to the first principal component.
To see this we printed the head of the PC1_sorted vector with 5 elements, the result was the following:

medFamInc	**medIncome**	**PctKids2Par**	**pctWInvInc**	**PctPopUnderPov**
0.1833080	0.1819830	0.1755423	0.1748683	0.1737978

Comment whether these features have anything in common and whether they may have a logical relationship to the crime level.
These features are economic factors or have a relation to family economy. It is fair to assume that the economy and the cime level have a correlation and that the crime levels are often higher when the income is lower.

Finally we used ggplot2 to plot the PC scores in the coordinate system PC1 and PC2 with a color gradient defined in the ViolentCrimesPerPop variable, which can be seen below.



The plot shows more red dots for high values of PC1 scores and more blue dots for lower values of PC1 scores. This means that violentCrimesPerPop are more frequent for higher values, confirming the correlation mentioned above. When it comes to PC2 the scores seem to be somewhat evenly distributed around 0, meaning that the PC2 scores do not seem to affect violentCrimesPerPop in any significant way. In other words, PC1 has a logical relationship to crime level while PC2 does not.

2.3.3 TASK 3

```
############################## Task 3 ##########################################
#Splitting the data into training and test

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

#Scaling the data
scaler = preProcess(train)
train_scaled = predict(scaler, train)
test_scaled = predict(scaler, test)

#linear regression
fit1=lm(ViolentCrimesPerPop ~ ., data=train_scaled)
summary(fit1)

#Predicting ViolentCrimesPerPop using the linear model
predicted_train = predict(fit1, train_scaled)
predicted_test = predict(fit1,test_scaled)

#Calculate Mean Squared Error for training and test data
mse_train = mean((train_scaled$ViolentCrimesPerPop - predicted_train)^2)
mse_test = mean((test_scaled$ViolentCrimesPerPop - predicted_test)^2)

print(paste("Training MSE: ", mse_train)) #~0.275
print(paste("Test MSE: ", mse_test)) #~0.425
```

In the third task we divided the dataset into training and test (50/50) and scaled it again, this time with the target variable included as per the instruction. Then we created a linear regression model using the lm() function to predict the ViolentCrimesPerPop variable. To evaluate the quality of the model, training and test MSE was calculated which gave the following result.

Training MSE: 0.275207137480974
Test MSE: 0.424801137490899

The test MSE is higher than the training MSE, which is expected, as models tend to perform better on data they have seen compared to new data. The test MSE is notably higher than the training MSE, but not excessively so. This suggests that while the model has learned to predict the training data reasonably well, its predictions are less accurate when applied to the test data. This indicates a slightly overfitted model and while the model shows some generalization capability, there is room for improvement.

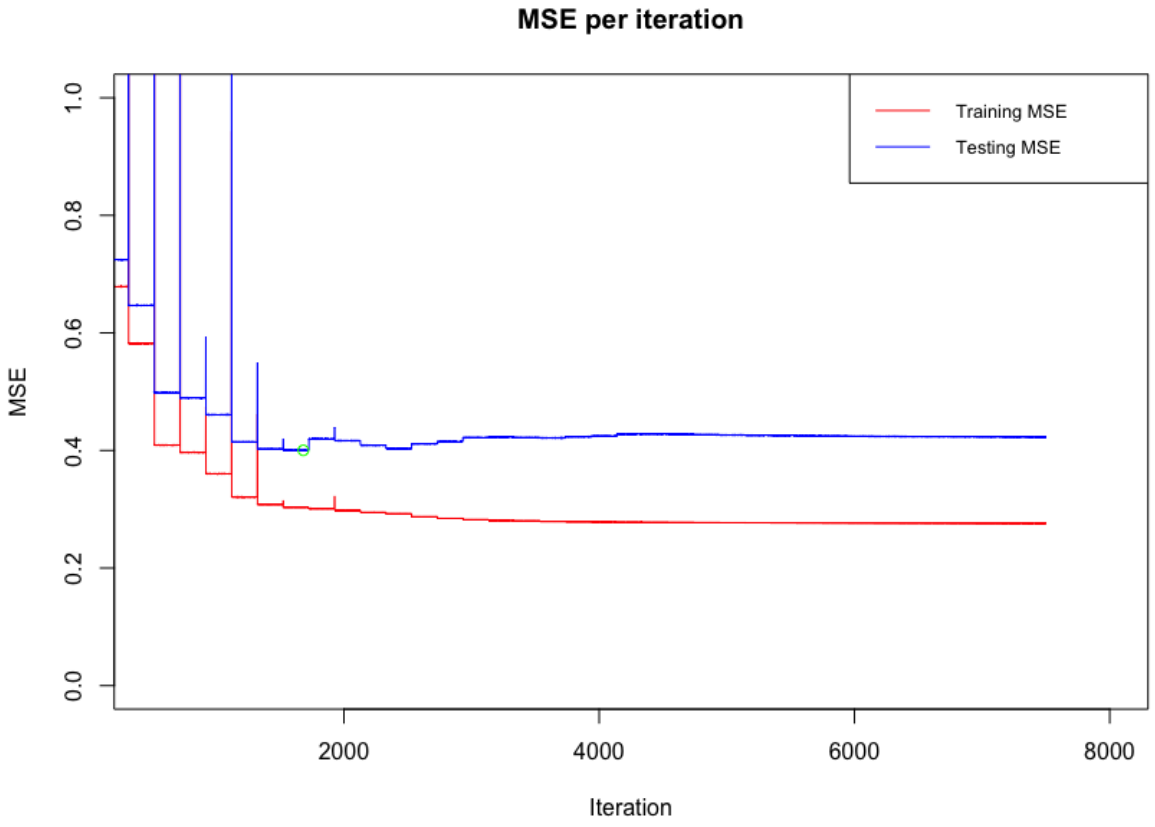
2.3.4 TASK 4

```
############################## Task 4 ##########################################
#Some global variable used in calculations below
#Empty vectors for storing MSE values to be plotted
MSE_training = c()
MSE_testing = c()
#scaled training params, VCPP excluded
x_train = as.matrix(train_scaled[,-101])
x_test = as.matrix(test_scaled[,-101])
#Target variable
target_train = train_scaled$ViolentCrimesPerPop
target_test = test_scaled$ViolentCrimesPerPop

#Implementation of cost function
cost_function = function(theta) {
  predictions_train = x_train %*% theta
  predictions_test = x_test %*% theta

  mse_train = mean((target_train - predictions_train)^2)
  mse_test = mean((target_test - predictions_test)^2)
  #Saving the MSE values to the vector
  MSE_training <<- c(MSE_training, mse_train)
  MSE_testing <<- c(MSE_testing, mse_test)
  return(mse_train)
}

#starting at theta = 0
theta = rep(0, ncol(x_train))
#Using the optim() fucntion to optimize the cost
opt_result = optim(par = theta, fn = cost_function, method = "BFGS")
```

In the last task we implemented a cost function for linear regression. One hint was to only store the errors, which is why we created two empty vectors first. The cost function took theta as a parameter and computed the train and test MSE for different theta values. For each iteration these were saved in the empty vectors created in the beginning. Theta was initialized as a vector of 100 zeros (one for each parameter) as per the instruction. Then the optim() function was used to optimize the cost function.

```
#takes the minimum MSE testing value to see which iteration that is optimal
min_test = which.min(MSE_testing)
min_train = which.min(MSE_training)

#printing the optimal training and test MSE
print(MSE_testing[min_test])
print(MSE_training[min_train])
#the index for the iteration numner for optimal test MSE
print(min_test)

#defining start and endpoints for the graph
startpoint = 500
endpoint = 8000
#Plotting the training MSE
plot(MSE_training[startpoint:endpoint], type = "l", col = "red", lwd = 1, ylim = c(0, 1), ›
#Adding the testing MSE line
lines(MSE_testing[startpoint:endpoint], col = "blue", lwd = 1)
#Marking the min test MSE, adding startpoint as offset
points(min_test - startpoint, MSE_testing[min_test], col = "green")
#Adding a legend to the plot
legend("topright", legend = c("Training MSE", "Testing MSE"), col = c("red", "blue"), lwd =
```

The next step was to plot a graph showing the MSE errors for both test and train for every iteration. To do this we first used the which.min() function to get the index (iteration) of the smallest MSE for test and training. We then defined a startpoint at 500 based on the hint in the instruction. The endpoint was derived by plotting the full graph and then choosing a suitable value. After that we used the plot() function to plot both test and training as lines. We also added a point to mark the minimum test MSE and a legend to the top right corner for explanation. Below is the graph that the code resulted in.



We were also asked to comment which iteration number that is optimal according to the early stopping criterion. This was found by printing the “min_test” variable, which resulted in the iteration **2182**. The optimal iteration number is also marked in the graph with the green dot. In the graph it is easy to see that the performance on the test data slightly worsens and flattens out after iteration 2500 approximately. By the definition of the early stopping criterion the training process should be halted at this point which confirms that the iteration 2182 is the best according to the criterion.

Compute the training and test error in the optimal model, compare them with results in step 3 and make conclusions.

This was done by printing the MSE corresponding to the best iteration in the graph for both test and training and gave the following result:

Training MSE: 0.2752213

Test MSE: 0.4002329

From the linear regression model in 3.3 the following MSE:

Training MSE: 0.275207137480974
Test MSE: 0.424801137490899

Here we can see that the training errors are almost identical and that the test errors are close in 3.3 and 3.4. One conclusion that can be drawn from this is that using the optim() function to optimze theta has led to a modest improvement in the model's generalization, as seen by a slightly reduced test MSE from 0.4248 to 0.4002, while maintaining a comparable training MSE.

Appendix

—----------Assignment 1—----------

```
########### Libraries #############
library(glmnet)
library(dplyr)
library(caret)

# Clear global env.
rm(list = ls())


############### Read data and divide it into test and train ###############
set.seed(12345)
df <- read.csv('tecator.csv', header=TRUE)
data <- (df)[2:102] #Relevant columns

n = dim(data)[1]
id = sample(1:n, floor(n * 0.5)) # Take a random sample
train_data = data[id, ]
test_data = data[-id, ]


######################################## TASK 1
########################################

# Model
fit = lm(Fat ~ . ,data = train_data)
summary(fit)

# Predictions
prediction_train = predict(fit, train_data)
prediction_test = predict(fit, test_data)
```

```
# Calculating errors, Mean Squared Error
mse_train = mean((train_data$Fat - prediction_train)^2)
mse_test = mean((test_data$Fat - prediction_test)^2)

print(paste("Train MSE: ", mse_train)) # 0.00570911701090834
print(paste("Test MSE: ", mse_test)) # 722.429419336971


###################################### TASK 2
######################################

# ...

###################################### TASK 3
######################################

## LASSO regression

x = as.matrix(train_data %>% select(-Fat))
y = as.matrix(train_data %>% select(Fat))

lasso = glmnet(x, y, alpha = 1, family = "gaussian")
plot(lasso, xvar = "lambda", label = TRUE, main="Lasso regression model")

###################################### TASK 4
######################################

## Ridge regression

ridge = glmnet(x, y, alpha = 0, family="gaussian")
plot(ridge, xvar = "lambda", label = TRUE, main="Ridge regression")

###################################### TASK 5
######################################

cross_validation = cv.glmnet(x, y, alpha=1, family="gaussian")
plot(cross_validation)
opt_lambda = cross_validation$lambda.min # 0.05744535


coef(cross_validation, s = "lambda.min")
summary(cross_validation)
# Channels 52, 51, 41, 40, 16, 15, 14, 13. 8 total

# Cross validation prediction
crossval_predict = predict(cross_validation, newx = x, s = opt_lambda)
```

```
plot(test_data$Fat, col="red", ylim = c(0,60), ylab="Fat levels", main = "Test vs model with
optimal lambda")
points(crossval_predict, col = "green")
legend("topright", c("actual", "predicted"), col = c("red", "green"), pch = 1, cex = 0.8)


##################################### EXTRA
#####################################

# Testing the channels derived from LASSO with linear regression model
fit_new <- lm(Fat ~ Channel52 + Channel51 + Channel41 + Channel40 + Channel16 +
Channel15 + Channel14 + Channel13, data = train_data)
ptrain_new = predict(fit_new, train_data)
ptest_new = predict(fit_new, test_data)

mse_train_new = mean((train_data$Fat - ptrain_new)^2)
mse_test_new = mean((test_data$Fat - ptest_new)^2)

print(paste("Train MSE: ", mse_train_new)) # 5.10650377834861
print(paste("Test MSE: ", mse_test_new)) # 15.7160188358832
```

—----------Assignment 2—----------

```
library(tree)

#### TASK 1 ####
rm(list = ls())

data = read.csv2("bank-full.csv", stringsAsFactors = TRUE)
data = data[, -12]

n=dim(data)[1]

set.seed(12345)
id=sample(1:n, floor(n*0.4))
train_data=data[id,]

id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.3))
validation_data=data[id2,]

id3=setdiff(id1,id2)
test_data=data[id3,]

rm(data, id, id1, id2, id3, n)
```

```
#### TASK 2 ####

# Decision Tree with default settings
default_tree <- tree(y ~ ., data = train_data)
plot(default_tree)

# Decision Tree with the smallest allowed node size (7000)
node_size_tree <- tree(y ~ ., data = train_data, control = tree.control(nrow(train_data),
minsize = 7000))
plot(node_size_tree)

# Decision Tree with minimum deviance set to 0.0005
deviance_tree <- tree(y ~ ., data = train_data, control = tree.control(nrow(train_data), mindev
= 0.0005))
plot(deviance_tree)

# Extract misclassification rates
default_missclass = summary(default_tree)$misclass
node_size_missclass = summary(node_size_tree)$misclass
deviance_missclass = summary(deviance_tree)$misclass

# Misclassification rates for training data
train_missclass_rates <- list(
  default = default_missclass[1] / default_missclass[2],
  node_size = node_size_missclass[1] / node_size_missclass[2],
  deviance = deviance_missclass[1] / deviance_missclass[2]
)

# Predictions on validation set
pred_default <- predict(default_tree, newdata = validation_data, type = "class")
pred_node_size <- predict(node_size_tree, newdata = validation_data, type = "class")
pred_deviance <- predict(deviance_tree, newdata = validation_data, type = "class")

# Misclassification rates for validation data
validation_missclass_rates <- data.frame(
  default = mean(pred_default != validation_data$y),
  node_size = mean(pred_node_size != validation_data$y),
  deviance = mean(pred_deviance != validation_data$y)
)


#### TASK 3 ####

# Initialize vectors to store training and validation deviances
trainScore <- rep(0, 50)
validScore <- rep(0, 50)
```

```
# Find optimal number of leaves. 2:50 because just one node gives error for prediciton
for (i in 2:50) {
  # Prune the tree to the specified number of leaves
  pruned_tree <- prune.tree(deviance_tree, best = i)

  # Check if the pruned tree is not a leaf node. i=1 gives error in predict
  pred_pruned_tree <- predict(pruned_tree, newdata = validation_data, type = "tree")

  # Calculate deviances for training and validation data
  trainScore[i] <- deviance(pruned_tree)
  validScore[i] <- deviance(pred_pruned_tree)
}

# Visualize with plot
plot(2:50, trainScore[2:50], type = "b", col = "red", ylim = c(7000, max(trainScore,
validScore)),
          main = "Optimal tree depth", ylab = "Deviance", xlab = "Size")
points(2:50, validScore[2:50], type = "b", col = "green")
legend("topright", c("train data", "validation data"), fill = c("red", "green"))

# Optimal number of leaves.
optimal_leaves_train <- which.min(trainScore[-1])
optimal_leaves_valid <- which.min(validScore[-1])
optimal_leaves <- data.frame(train = optimal_leaves_train, valid = optimal_leaves_valid)

# Optimal tree
optimal_tree <- tree(y ~ ., data = train_data, control = tree.control(nrow(train_data), mindev =
0.0005, minsize = optimal_leaves_valid))

# Prune the tree
pruned_optimal_tree <- prune.tree(optimal_tree, best = optimal_leaves_valid)

# Display pruned tree structure
plot(pruned_optimal_tree)
text(pruned_optimal_tree, pretty=1)

#### TASK 4 ####
# Predictions on the test set using the optimal model
test_predictions <- predict(pruned_optimal_tree, newdata = test_data, type = "class")

# Confusion Matrix
confusion_matrix <- table(test_data$y, test_predictions)

# Calculate Accuracy
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)

# Calculate precision TP/(TP+FP)
precision <- confusion_matrix[2,2]/(confusion_matrix[2,2] + confusion_matrix[1,2])
```

```
# Calculate Recall Rate TP/P
recall_rate <- confusion_matrix[2,2]/sum(confusion_matrix[2,])

# Calculate F1 Score
f1_score <- (2 * precision * recall_rate) / (precision + recall_rate)

# Print Results
cat("Confusion Matrix:\n", confusion_matrix)
cat("\nAccuracy for the optimal model:\n", accuracy)
cat("\nRecall Rate for the optimal model:\n", recall_rate)
cat("\nF1 Score for the optimal model:\n", f1_score)

#### TASK 5 ####

# Decision tree classification with Loss matrix:
loss_matrix <- matrix(c(0, 1, 5, 0), byrow = TRUE, nrow = 2)

# Predictions using the pruned optimal tree
probabilities <- predict(pruned_optimal_tree, newdata = test_data)

# loss om den gissar yes när det är no är P(no|x) * 1 enligt matris
# loss om den gissar no när det är yes är P(yes|x) * 5 enligt matris
# Matrix multiplication (p(no|x)*1 p(yes|x)*5)
losses <- probabilities %*% loss_matrix

# Find column index with lowest value along each row (max of negative)
lowest_indices <- max.col(-losses)

# Convert to levels
predictions_with_loss <- levels(test_data$y)[lowest_indices]

# Confusion matrix with loss matrix
confusion_matrix_loss <- table(test_data$y, predictions_with_loss)
confusion_matrix_loss

# Calculate Accuracy
accuracy_loss <- sum(diag(confusion_matrix_loss)) / sum(confusion_matrix_loss)

# Calculate precision TP/(TP+FP)
precision_loss <- confusion_matrix_loss[2,2]/(confusion_matrix_loss[2,2] +
confusion_matrix_loss[1,2])

# Calculate Recall Rate TP/P
recall_rate_loss <- confusion_matrix_loss[2,2]/sum(confusion_matrix_loss[2,])

# Calculate F1 Score
f1_score_loss <- (2 * precision_loss * recall_rate_loss) / (precision_loss + recall_rate_loss)
```

```
# Print Results
cat("Confusion Matrix with Loss Matrix:\n", confusion_matrix_loss)
cat("\nAccuracy for the model with Loss Matrix:\n", accuracy_loss)
cat("\nF1 Score for the model with Loss Matrix:\n", f1_score_loss)

#### TASK 6 ####

# Logistic regression model
logistic_model <- glm(y ~ ., data = train_data, family = "binomial")

# Initialize vectors for TPR and FPR
tpr_tree <- rep()
fpr_tree <- rep()
tpr_logistic <- rep()
fpr_logistic <- rep()

# Computing ROC curves for the optimal tree and logistic regression model
for (pi in seq(from = 0.05, to = 0.95, by = 0.05)) {

  # Optimal tree predictions
  pred_tree <- ifelse(predict(pruned_optimal_tree, newdata = test_data, type = "vector")[, 2] >
pi, "yes", "no")
  cm_tree <- table(pred_tree, test_data$y)

  #pred_tree    no       yes
  #       no  TN FN
  #       yes FP TP

  #Sometimes model never predicts "yes", giving no tpr
  if (nrow(cm_tree) >= 2) {
          #TPR = TP / TP + FN
          tpr_tree <- c(tpr_tree, cm_tree[2, 2] / sum(cm_tree[, 2]))
          #FPR =  FP / FP + TN
          fpr_tree <- c(fpr_tree, cm_tree[2, 1] / sum(cm_tree[, 1]))
  }


  # Logistic regression predictions
  pred_logistic <- ifelse(predict(logistic_model, newdata = test_data, type = "response") > pi,
"yes", "no")
  cm_logistic <- table(pred_logistic, test_data$y)

  if (nrow(cm_logistic) >= 2) {
          tpr_logistic <- c(tpr_logistic, cm_logistic[2, 2] / sum(cm_logistic[, 2]))
          fpr_logistic <- c(fpr_logistic, cm_logistic[2, 1] / sum(cm_logistic[, 1]))
  }
}
```

```
# Plotting ROC curves
plot(fpr_tree, tpr_tree, pch = 5, type = "b", col = "red", main = "ROC Curves", xlab = "False Positive Rate (FPR)", ylab = "True Positive Rate (TPR)")
points(fpr_logistic, tpr_logistic, pch = 5, type = "b", col = "blue")
legend("bottomright", c("Optimal Tree", "Logistic Regression"), fill = c("red", "blue"))
```

——----------Assignment 3—----------

```
#libraries
library(caret)
library(ggplot2)

#Clear global environment
rm(list = ls())

################################ Task 1
##########################################
#Scale the data and manually implement PCA

#Reading the dataset and excluding the target variable ViolentCrimesPerPop for PCA
communities_data = read.csv("communities.csv")
VCPP_excluded = communities_data[,1:100]

#Scaling the data
scaler = preProcess(VCPP_excluded)
data_scaled = predict(scaler, VCPP_excluded)

#Calculating the covariance matrix of the scaled data and computing the eigenvalues
cov_matrix = cov(data_scaled)
eigen_values = eigen(cov_matrix) #The values vector is sorted in decreasing order

#Calculating the cumulative proportion of variance explained by each PC
#Using the match() function to find how many components are needed to obtain 95% of the variance
cumulative_variance = cumsum(eigen_values$values)
match(TRUE, cumulative_variance >= 95) #35 PCs
print(cumulative_variance) #printing the cumulative_variance vector to verify the step above
print(eigen_values$values[1]) #25.01699
print(eigen_values$values[2]) #16.93597

#Plotting the variance of the 10 first componentents in a barplot for visualization purposes
PCA = data.frame(PCA = 1:10, Variance = (eigen_values$values)[1:10])
```

```
ggplot(PCA, aes(x = PCA, y = Variance)) + geom_bar(stat = "identity") + labs(title =
"Variance of Principal Components", x = "Principal Component", y = "Variance Explained
(%)") + theme_minimal()

############################### Task 2
#########################################
#Repeat PCA using princomp(), make trace plot of the first principal component

PCA_res = princomp(data_scaled)
#Sorting the absolute values of the first PC's loadings in descending order
PC1_sorted = sort(abs(PCA_res$loadings[,1]), decreasing = TRUE)
plot(PC1_sorted, main="Traceplot PC1", col = "blue", cex = 0.5, ylab = "loadings")
print(head(PC1_sorted, 5)) #The 5 features that contribute mostly

#Plotting the PC scores
Violent_Crimes = communities_data$ViolentCrimesPerPop
ggplot(data.frame(PC1 = PCA_res$scores[,1], PC2 = PCA_res$scores[,2]), aes(x=PC1,
y=PC2)) + geom_point(aes(color = Violent_Crimes)) + scale_color_gradient(low = "blue",
high = "red") + ggtitle("PC scores") + xlab("PC1 scores") +  ylab("PC2 scores") +
theme_minimal()

############################### Task 3
#########################################

#Splitting the data into training and test
set.seed(12345)
n=nrow(communities_data)
id=sample(1:n, floor(n*0.5))
train=communities_data[id,]
test=communities_data[-id,]

#Scaling the data
scaler = preProcess(train)
train_scaled = predict(scaler, train)
test_scaled = predict(scaler, test)

#linear regression
fit1=lm(ViolentCrimesPerPop ~ ., data=train_scaled)
summary(fit1)

#Predicting ViolentCrimesPerPop using the linear model
predicted_train = predict(fit1, train_scaled)
predicted_test = predict(fit1,test_scaled)

#Calculate Mean Squared Error for training and test data
mse_train = mean((train_scaled$ViolentCrimesPerPop - predicted_train)^2)
mse_test = mean((test_scaled$ViolentCrimesPerPop - predicted_test)^2)
```

```
print(paste("Training MSE: ", mse_train)) #~0.275
print(paste("Test MSE: ", mse_test)) #~0.425

################################ Task 4
##########################################
#Some global variable used in calculations below
#Empty vectors for storing MSE values to be plotted
MSE_training = c()
MSE_testing = c()
#scaled training params, VCPP excluded
x_train = as.matrix(train_scaled[,-101])
x_test = as.matrix(test_scaled[,-101])
#Target variable
target_train = train_scaled$ViolentCrimesPerPop
target_test = test_scaled$ViolentCrimesPerPop

#Implementation of cost function
cost_function = function(theta) {
  predictions_train = x_train %*% theta
  predictions_test = x_test %*% theta

  mse_train = mean((target_train - predictions_train)^2)
  mse_test = mean((target_test - predictions_test)^2)
  #Saving the MSE values to the vector
  MSE_training <<- c(MSE_training, mse_train)
  MSE_testing <<- c(MSE_testing, mse_test)
  return(mse_train)
}

#starting at theta = 0
theta = rep(0, ncol(x_train))
#Using the optim() fucntion to optimize the cost
opt_result = optim(par = theta, fn = cost_function, method = "BFGS")

#takes the minimum MSE testing value to see which iteration that is optimal
min_test = which.min(MSE_testing)
min_train = which.min(MSE_training)

#printing the optimal training and test MSE
print(MSE_testing[min_test])
print(MSE_training[min_train])
#the index for the iteration numner for optimal test MSE
print(min_test)

#defining start and endpoints for the graph
startpoint = 500
endpoint = 8000
#Plotting the training MSE
```

```
plot(MSE_training[startpoint:endpoint], type = "l", col = "red", lwd = 1, ylim = c(0, 1), xlim =
c(startpoint, endpoint), xlab = "Iteration", ylab = "MSE", main = "MSE per iteration")
#Adding the testing MSE line
lines(MSE_testing[startpoint:endpoint], col = "blue", lwd = 1)
#Marking the min test MSE, adding startpoint as offset
points(min_test - startpoint, MSE_testing[min_test], col = "green")
#Adding a legend to the plot
legend("topright", legend = c("Training MSE", "Testing MSE"), col = c("red", "blue"), lwd = 1,
cex = 0.8)
```