Mini Project 6: Report
STAT 6340

Raheel Ahmed
Rsa170130

Section 1

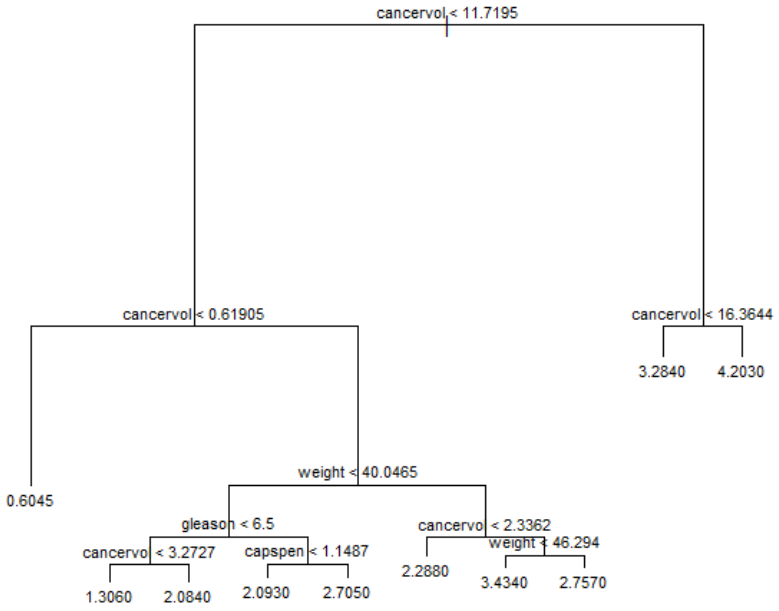
Question 1

a. **Fit a tree to the data. Summarize the results. Display the tree graphically and explicitly describe the regions corresponding to the terminal nodes. Report the test MSE.**

The summary is as follows:

```
Regression tree:
tree(formula = psa ~ ., data = pc_data)
Variables actually used in tree construction:
[1] "cancervol" "weight"    "gleason"   "capspen"
Number of terminal nodes:  10
Residual mean deviance:  0.4022 = 34.99 / 87
Distribution of residuals:
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-1.50300 -0.48790  0.09691  0.00000  0.44560  1.37700
```

The tree visualized is as follows:

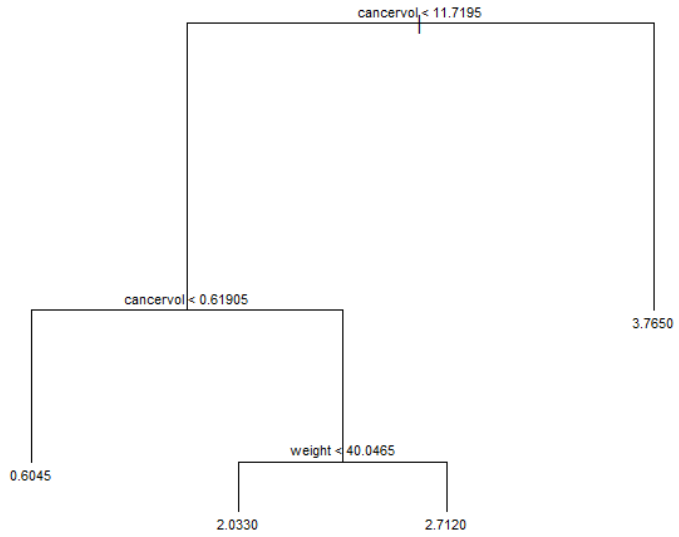


From the summary, we can see that the majority of options are with cancervol < 11.7195 (76 observations) and cancervol > 0.61905 (67 observations). In our case, the majority of these observations are with weight < 40.0465. There are four nodes in this portion of the tree and the criteria are listed below:

a. Gleason < 6.5 and cancervol < 3.2727
b. Gleason < 6.5 and capspen < 1.1487
c. Gleason < 6.5 and cancervol > 3.2727
d. Gleason < 6.5 and capspen > 1.1487

The estimated test MSE via LOOCV was 0.8567983.

b. **Use LOOCV to determine whether pruning is helpful and determine the optimal size for the pruned tree. Compare the best pruned and un-pruned trees. Report estimated test MSE for the best pruned tree. Which predictors seem to be the most important?**
Estimated test MSE via LOOCV was 0.7437038. Our pruned tree was found via LOOCV to have a fairly optimal size of 4 terminal nodes. It is shown below as follows:



The predictors that seem the most important are cancervol and weight, which is similar to our first tree. However, it also seems that this simpler pruned model avoids overcomplexity and yields a higher estimated test accuracy.

c. **Use a bagging approach to analyze the data with B = 1000. Compute the estimated test MSE. Which predictors seem to be the most important?**
Estimated test MSE via LOOCV was 0.6119430. From the importance function and varImpPlot(), we can see that cancervol, weight, and vesinv appear to be the most important predictors for this bagging approach.

d. **Use a random forest approach to analyze the data with B = 1000 and m ≈ p/3. Compute the estimated test MSE. Which predictors seem to be the most important?**
Estimated test MSE via LOOCV was 0.5641470. Again, from the importance function, we can see that cancervol, weight, and vesinv are the top most important functions for this random forest approach as well.

e. **Use a boosting approach to analyze the data with B = 1000, d = 1, and λ = 0.01. Compute the estimated test MSE. Which predictors seem to be the most important?**
Estimated test MSE via LOOCV was 0.6064757. The rel.inf values in the summary of the boosted model show us that cancervol, weight, and vesinv are the most important once again.

f. Compare the results from the various methods. Which method would you recommend? How does your recommendation compare with the method you recommended in the previous project?

The method I would recommend is probably random forest; it has the lowest estimated test MSE found via LOOCV, the random forest approach prevents correlation between trees, and it seemed fairly easy to compute for our dataset. The estimated test MSE for PLS and PCR were both higher at approximately 0.77 than our boosted and random forest variants of decision trees, so I would still use my recommendation over the PCR and/or PLS approach.

Question 2

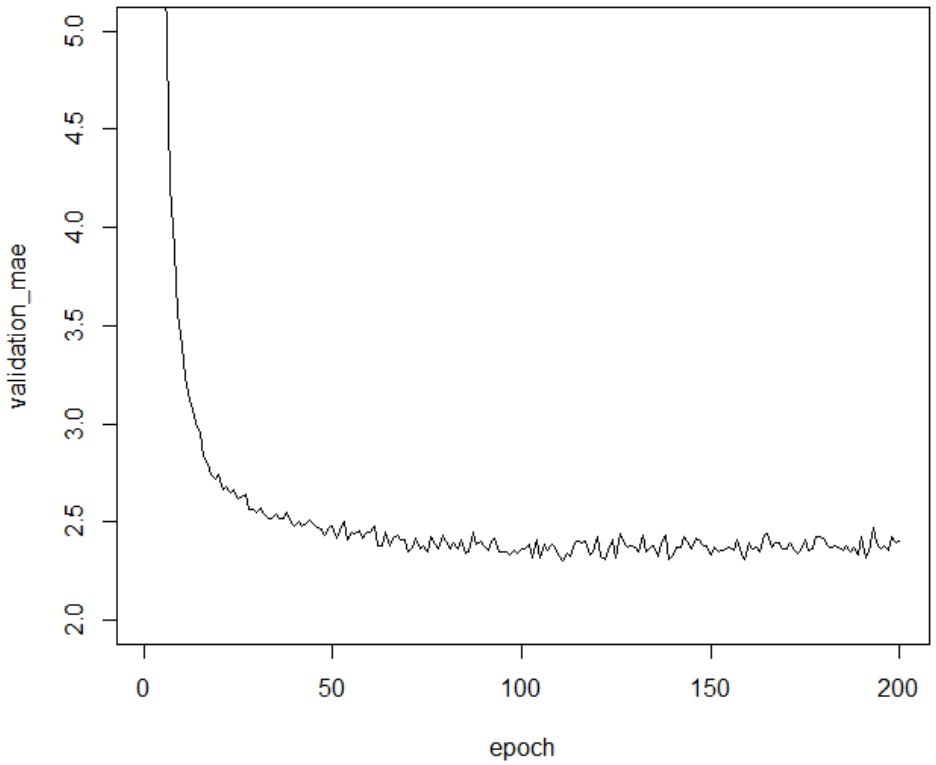
Results of trying models with hidden units in {256, 512}, epochs in {5, 10}, and 1 or 2 hidden layers are shown below. Also, we include some models with L2 regularization with lambda = 0.001 and one with 50% dropout.

	layers	units	epochs	dropout	L2_reg	train_acc	test_acc
2	1	512	10	FALSE	FALSE	0.9978833	0.9835
6	2	512	10	FALSE	FALSE	0.9979666	0.9831
7	2	256	5	FALSE	FALSE	0.9886166	0.9812
4	1	256	10	FALSE	FALSE	0.9953167	0.9807
8	2	256	10	FALSE	FALSE	0.9965667	0.9800
1	1	512	5	FALSE	FALSE	0.9887834	0.9788
5	2	512	5	FALSE	FALSE	0.9915833	0.9786
10	1	512	5	TRUE	FALSE	0.9741333	0.9783
3	1	256	5	FALSE	FALSE	0.9842333	0.9760
9	1	512	5	FALSE	TRUE	0.9712167	0.9683

As we can see the model with 1 layer, 512 hidden units, and 10 epochs without dropout or L2 regularization is the model with the highest test accuracy. As such, I chose this model as my recommended model. Other attempts have yielded similar results to my memory.

Question 3

a. **Fit a neural network model with 2 hidden layers, each with 64 hidden units, and 200 epochs. Make a plot of validation MAE against epoch. Would you recommend early stopping based on this plot? How many epochs would you suggest? Fit a model with the suggested number of epochs. Reports its validation MAE. Use this suggested number of epochs for all the models below.**



I would recommend early stopping based on the plot. I would recommend around 75 epochs. The early stopped model with 75 epochs and the same architecture has a validation MAE of 2.372106.

b. **Fit a neural network model with 1 hidden layer with 128 units. Report its validation MAE.**
   This one's validation MAE was 2.346228.
c. **Add L2 weight regularization to the model with 2 hidden layers, each with 64 hidden units. Report its validation MAE.**
   This one's validation MAE was 2.326746.
d. **Add L2 weight regularization to the model with 1 hidden layer with 128 hidden units. Report its validation MAE.**
   This one's validation MAE was 2.335257.
e. Compare the above models. Which model would you recommend? Compute MAE of the recommended model from the test data. Comment on the results.
   The above models are fairly close in validation MAE, but I will recommend the L2 weight regularized model with 2 layers of 64 hidden units each. It had the lowest validation MAE at 2.326746. The MAE of the recommended model on the test data was 2.644582. This seemed fairly low and a reasonable MAE, leading me to think that this model seems like a balanced, fair model with not much overfitting or underfitting.

Section 2

Question 1

```
library(tree)
library(randomForest)
library(gbm)

# Read in prostate cancer data
pc_data <- read.csv("prostate_cancer.csv")
# Eliminate subject number feature
pc_data <- pc_data[,-1]
# Treat vesinv as a qualitative variable
pc_data$vesinv <- factor(pc_data$vesinv, order=F, levels = c(0, 1))
# Conduct a natural log transformation on the response
# to adjust it's distribution to something more appropriate.
pc_data[, 1] <- log(pc_data[, 1])
hist(pc_data[, 1])

#a
# Create a decision tree with psa as response and the rest as potential
predictors
tree_pc <- tree(psa ~ ., pc_data)
# Print out a summary then visualize the tree made
summary(tree_pc)
plot(tree_pc)
text(tree_pc, pretty = 0, cex = 0.7)

#b
# Find optimal number of nodes via prune.tree function and cross-validation
cv.pc <- cv.tree(tree_pc, FUN = prune.tree, K=nrow(pc_data))
# Plot the deviance against size
plot(cv.pc$size, cv.pc$dev, type = "b")

# Find size at which you have minimum deviance
# Minimum is usually 8 or 9
cv.pc$size[which.min(cv.pc$dev)]
# But as we can see, the deviance with 4 terminal nodes is really close to
# those at higher sizes and thus 4 serves as a great elbow point in my
opinion
cv.pc$dev[cv.pc$size == 8]
cv.pc$dev[cv.pc$size == 4]


# Prune the tree with the elbow in mind, our elbow point is at size=4
prune.pc <- prune.tree(tree_pc, best = 4)
plot(prune.pc)
text(prune.pc, pretty = 0, cex = 0.7)


# c.
# Perform bagging with specified parameters and check importance of
predictors
bag.pc <- randomForest(psa ~ ., data = pc_data, mtry = 7, ntree = 1000,
importance = TRUE)
importance(bag.pc)
```

```r
varImpPlot(bag.pc)

# d.
# Perform random forest with the specified parameters and check importance of
predictors
rf.pc <- randomForest(psa ~ ., data = pc_data, mtry = round(7/3), ntree =
1000, importance = TRUE)
importance(rf.pc)
varImpPlot(rf.pc)

# e.
# Perform boosting with gbm and specified parameters and check importance of
predictors
boost.pc <- gbm(psa ~ ., data = pc_data, distribution = "gaussian",
                    n.trees = 1000, interaction.depth = 1, shrinkage=0.01)
summary(boost.pc)


# Make a function to run LOOCV on all models we want to evaluate
LOOCV_tree <- function(){
  # Set k to number of observations for LOOCV
  k <- nrow(pc_data)
  # Select indices for each fold
  indices <- sample(1:nrow(pc_data))
  folds <- cut(indices, breaks = k, labels = FALSE)
  # Establish structures to store MSE data in
  unpruned_MSEs <- c()
  pruned_MSEs <- c()
  bagged_MSEs <- c()
  rf_MSEs <- c()
  boost_MSEs <- c()

  # Iterate through each fold
  for (i in 1:k){
    # Make validation and training data
    val_indices <- which(folds == i, arr.ind = TRUE)
    val_data <- pc_data[val_indices,]
    train_data <- pc_data[-val_indices,]

    # For each model, compute MSE and store it
    # Base Decision Tree model
    train_tree_pc <- tree(psa ~ ., train_data)
    unpruned_MSE <- (val_data$psa - predict(train_tree_pc, val_data))^2
    unpruned_MSEs <- c(unpruned_MSEs, unpruned_MSE)

    # Pruned Tree model with potentially optimal best number of terminal
nodes
    train_pruned_tree <- prune.tree(train_tree_pc, best=4)
    pruned_MSE <- (val_data$psa - predict(train_pruned_tree, val_data))^2
    pruned_MSEs <- c(pruned_MSEs, pruned_MSE)

    # Bagging model evaluation
    train_bag <- randomForest(psa ~ ., data = train_data, mtry = 7, ntree =
1000, importance = TRUE)
    bagged_MSE <- (val_data$psa - predict(train_bag, newdata = val_data))^2
    bagged_MSEs <- c(bagged_MSEs, bagged_MSE)
```

```
    # Random forest model evaluation
    train_rf <- randomForest(psa ~ ., data = train_data, mtry = round(7/3),
ntree = 1000, importance = TRUE)
    rf_MSE <- (val_data$psa - predict(train_rf, newdata = val_data))^2
    rf_MSEs <- c(rf_MSEs, rf_MSE)

    # Boosted model evaluation
    train_boost <- gbm(psa ~ ., data = train_data, distribution = "gaussian",
                       n.trees = 1000, interaction.depth = 1, shrinkage=0.01)
    boost_MSE <- (val_data$psa - predict(train_boost, newdata = val_data,
n.trees = 1000))^2
    boost_MSEs <- c(boost_MSEs, boost_MSE)
  }
  # Return mean of MSEs per model
  result <- c(
              mean(unpruned_MSEs),
              mean(pruned_MSEs),
              mean(bagged_MSEs),
              mean(rf_MSEs),
              mean(boost_MSEs)
            )
  # Store result in named vector
  return(setNames(result, c("unpruned_est_MSE", "pruned_est_MSE",
"bagged_est_MSE", "rand_forest_est_MSE", "boosted_est_MSE")))
}
# Store and print MSEs
MSEs <- LOOCV_tree()
MSEs
```

Question 2

```
library(keras)

# Get mnist data
mnist <- dataset_mnist()
# Partition training and test images from mnist
train_images <- mnist$train$x
train_labels <- mnist$train$y
test_images <- mnist$test$x
test_labels <- mnist$test$y

# Reshape and scale data where needed
train_images <- array_reshape(train_images, c(60000, 28*28)) # matrix
train_images <- train_images/255 # ensures all values are in [0, 1]
test_images <- array_reshape(test_images, c(10000, 28*28))
test_images <- test_images/255

# Obtain categorical versions of training and test labels
cat_train_labels <- to_categorical(train_labels)
cat_test_labels <- to_categorical(test_labels)

# Store results in a dataframe
results_df <- data.frame(matrix(nrow=0, ncol=7))
colnames(results_df) <- c("layers", "units", "epochs", "dropout", "L2_reg",
"train_acc", "test_acc")
```

```
# Make a function to create different network architectures depending on
parameters
# The function will also compile each network, fit the network on training
data, and evaluate on testing data.
exec_network <- function(num_layers, num_units, num_epochs, dropout, L2_reg){
  # Make models with 2 layers
  if(num_layers == 2){
    # Base type of model with specified number of nodes
    network <- keras_model_sequential() %>%
      layer_dense(units = num_units, activation = "relu", input_shape =
c(28*28)) %>%
      layer_dense(units = num_units, activation = "relu", input_shape =
c(28*28)) %>%
      layer_dense(units = 10, activation = "softmax")
  }
  else{
    # Models with only 1 layer
    if(dropout){
      # Make a model with dropout
      network <- keras_model_sequential() %>%
        layer_dense(units = num_units, activation = "relu", input_shape =
c(28*28)) %>%
        layer_dropout(rate = 0.5) %>%
        layer_dense(units = 10, activation = "softmax")
    }
    else if(L2_reg){
      # Make a model with L2 regularization and lambda = 0.001
      network <- keras_model_sequential() %>%
        layer_dense(units = num_units, activation = "relu", input_shape =
c(28*28),
                    kernel_regularizer = regularizer_l2(0.001)) %>%
        layer_dense(units = 10, activation = "softmax")
    }
    else{
      # Make a model without dropout or L2 regularization
      network <- keras_model_sequential() %>%
        layer_dense(units = num_units, activation = "relu", input_shape =
c(28*28)) %>%
        layer_dense(units = 10, activation = "softmax")
    }
  }
  # Compile the network
  network %>% compile(
    optimizer = "rmsprop",
    loss = "categorical_crossentropy",  # loss function to minimize
    metrics = c("accuracy") # monitor classification accuracy
  )
  # Fit the network on the training data and categorical train labels.
  # This is where number of epochs parameters is passed to the fitting
function
  history <- network %>% fit(train_images, cat_train_labels, epochs =
num_epochs, batch_size = 128, verbose = F)

  # Evaluate the network on test images and the categorical test labels
  metrics <- network %>% evaluate(test_images, cat_test_labels, verbose = F)
```

```
  # Use metrics to report on test accuracy and history to report on training
accuracy
  # Store that data in the results dataframe
  return(data.frame(layers=num_layers, units=num_units, epochs=num_epochs,
                    dropout=dropout, L2_reg=L2_reg,
                    train_acc=history$metrics$accuracy[num_epochs],
test_acc=metrics["accuracy"][[1]]))

}
# Try different permutations of networks with different numbers
# of layers, units, epochs, and dropout and L2 regularization status
results_df <- rbind(results_df, exec_network(num_layers = 1, num_units = 512,
num_epochs = 5, dropout = F, L2_reg = F))
results_df <- rbind(results_df, exec_network(num_layers = 1, num_units = 512,
num_epochs = 10, dropout = F, L2_reg = F))
results_df <- rbind(results_df, exec_network(num_layers = 1, num_units = 256,
num_epochs = 5, dropout = F, L2_reg = F))
results_df <- rbind(results_df, exec_network(num_layers = 1, num_units = 256,
num_epochs = 10, dropout = F, L2_reg = F))

results_df <- rbind(results_df, exec_network(num_layers = 2, num_units = 512,
num_epochs = 5, dropout = F, L2_reg = F))
results_df <- rbind(results_df, exec_network(num_layers = 2, num_units = 512,
num_epochs = 10, dropout = F, L2_reg = F))
results_df <- rbind(results_df, exec_network(num_layers = 2, num_units = 256,
num_epochs = 5, dropout = F, L2_reg = F))
results_df <- rbind(results_df, exec_network(num_layers = 2, num_units = 256,
num_epochs = 10, dropout = F, L2_reg = F))

results_df <- rbind(results_df, exec_network(num_layers = 1, num_units = 512,
num_epochs = 5, dropout = F, L2_reg = T))
results_df <- rbind(results_df, exec_network(num_layers = 1, num_units = 512,
num_epochs = 5, dropout = T, L2_reg = F))

# store results for retrial attempts
saved2_df <- results_df
```

Question 3

```
library(keras)

# Obtain boston dataset information
boston <- dataset_boston_housing()
# Separate boston dataset into train and test data
c(c(train_data, train_targets), c(test_data, test_targets)) %<-% boston

# Obtain mean, stdev, and scale the data
mean <- apply(train_data, 2, mean)
std <- apply(train_data, 2, sd)
train_data <- scale(train_data, center = mean, scale = std)
test_data <- scale(test_data, center = mean, scale = std)

# Specify a function to create a 2 hidden layer model with 64 hidden units
# using ReLU activation and linear 1-node output
build_model <- function(){
```

```
  # specify the model
  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(train_data)[2]) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)
  # compile the model
  model %>% compile(
    optimizer = "rmsprop",
    loss = "mse",
    metrics = c("mae") # mean absolute error
  )
}

# K-fold CV
# Specify 4 folds
k <- 4
# Partition indices and determine folds
indices <- sample(1:nrow(train_data))
folds <- cut(indices, breaks = k, labels = FALSE)
# Supply num epochs and create variable to track histories
num_epochs <- 200
all_mae_histories <- c()
for (i in 1:k){
  cat("Processing fold #", i, "\n")
  # Partition data into validation and training data
  val_indices <- which(folds == i, arr.ind = TRUE) # prepares the validation
data: data from partition #k
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,] # prepares the training
data: data from all other partitions
  partial_train_targets <- train_targets[-val_indices]

  # Use build model function to make architecture and compile
  model <- build_model()

  # Fit and track history on partial training data
  history <- model %>% fit(partial_train_data, partial_train_targets,
                validation_data = list(val_data, val_targets),
                epochs = num_epochs, batch_size = 16,
                verbose = 0) # trains the model in silent mode (verbose = 0)
  # Obtain validation MAE from history
  mae_history <- history$metrics$val_mae
  # Store the MAE data
  all_mae_histories <- rbind(all_mae_histories, mae_history)
}
# Show MAE per epoch
average_mae_history <- data.frame(
  epoch = seq(1:ncol(all_mae_histories)),
  validation_mae = apply(all_mae_histories, 2, mean)
)

# Plot validation MAE against epoch.
# We can see from the results that there is not much improvement after epoch
75
```

```
plot(validation_mae ~ epoch, average_mae_history, ylim = c(2, 5), type ="l")


# a.
# Repeat previous process of 4-fold CV for 2-layer, 64 hidden unit model with
early stopping
all_scores <- c()
for (i in 1:k){
  cat("Processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE) # prepares the validation
data: data from partition #k
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,] # prepares the training
data: data from all other partitions
  partial_train_targets <- train_targets[-val_indices]

  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(train_data)[2]) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)
  # compile the model
  model %>% compile(
    optimizer = "rmsprop",
    loss = "mse",
    metrics = c("mae") # mean absolute error
  )

  opt_history <- model %>% fit(partial_train_data, partial_train_targets,
epochs = 75,
                               batch_size = 16, verbose = 0)
  results <- model %>% evaluate(val_data, val_targets, verbose = 0)
  all_scores <- c(all_scores, results["mae"])
}
# Print mean MAE
mean(all_scores)



# b.
# Repeat previous process of 4-fold CV for 1-layer, 128 hidden unit model
with early stopping
all_scores <- c()
for (i in 1:k){
  cat("Processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE) # prepares the validation
data: data from partition #k
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,] # prepares the training
data: data from all other partitions
  partial_train_targets <- train_targets[-val_indices]
```

```
  model <- keras_model_sequential() %>%
    layer_dense(units = 128, activation = "relu",
                input_shape = dim(train_data)[2]) %>%
    layer_dense(units = 1)
  # compile the model
  model %>% compile(
    optimizer = "rmsprop",
    loss = "mse",
    metrics = c("mae") # mean absolute error
  )
  opt_history <- model %>% fit(partial_train_data, partial_train_targets,
epochs = 75,
                               batch_size = 16, verbose = 0)
  results <- model %>% evaluate(val_data, val_targets, verbose = 0)
  all_scores <- c(all_scores, results["mae"])
}
mean(all_scores)



# c.
# Repeat previous process of 4-fold CV for 2-layer, 64 hidden unit model with
early stopping
# with L2 regularization
all_scores <- c()
for (i in 1:k){
  cat("Processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE) # prepares the validation
data: data from partition #k
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,] # prepares the training
data: data from all other partitions
  partial_train_targets <- train_targets[-val_indices]

  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(train_data)[2],
                kernel_regularizer = regularizer_l2(0.001)) %>%
    layer_dense(units = 64, activation = "relu",
                kernel_regularizer = regularizer_l2(0.001)) %>%
    layer_dense(units = 1)
  # compile the model
  model %>% compile(
    optimizer = "rmsprop",
    loss = "mse",
    metrics = c("mae") # mean absolute error
  )

  opt_history <- model %>% fit(partial_train_data, partial_train_targets,
epochs = 75,
                               batch_size = 16, verbose = 0)
  results <- model %>% evaluate(val_data, val_targets, verbose = 0)
  all_scores <- c(all_scores, results["mae"])
```

```
}
mean(all_scores)


# d.
# Repeat previous process of 4-fold CV for 1-layer, 128 hidden unit model
with early stopping
# with L2 regularization
all_scores <- c()
for (i in 1:k){
  cat("Processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE) # prepares the validation
data: data from partition #k
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,] # prepares the training
data: data from all other partitions
  partial_train_targets <- train_targets[-val_indices]

  model <- keras_model_sequential() %>%
    layer_dense(units = 128, activation = "relu",
                input_shape = dim(train_data)[2],
                kernel_regularizer = regularizer_l2(0.001)) %>%
    layer_dense(units = 1)
  # compile the model
  model %>% compile(
    optimizer = "rmsprop",
    loss = "mse",
    metrics = c("mae") # mean absolute error
  )
  opt_history <- model %>% fit(partial_train_data, partial_train_targets,
epochs = 75,
                               batch_size = 16, verbose = 0)
  results <- model %>% evaluate(val_data, val_targets, verbose = 0)
  all_scores <- c(all_scores, results["mae"])
}
mean(all_scores)



# e.
# Take our considered optimal model and fit it again
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu",
              input_shape = dim(train_data)[2],
              kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_dense(units = 64, activation = "relu",
              kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_dense(units = 1)
# compile the model
model %>% compile(
  optimizer = "rmsprop",
  loss = "mse",
  metrics = c("mae") # mean absolute error
)
```

```
opt_history <- model %>% fit(train_data, train_targets, epochs = 75,
batch_size = 16, verbose = 0)

# Evaluate the model on the test data and print out the results
results <- model %>% evaluate(test_data, test_targets, verbose = 0)
results
```